Accelerating the Digital Journey from Legacy Systems to MODERN MICROSERVICES

By Zeev Avidan
& Hans Otharsson

Accelerating the Digital Journey from Legacy Systems to Modern Microservices

Zeev Avidan
Hans Otharsson

ISBN-13: 978-1987762822

Table of Contents

About the Authors

Zeev Avidan - Zeev has decades of experience with legacy systems and legacy system integration starting with his days as a mainframe systems architect in 1996 where he also designed and implemented Service Oriented Architecture (SOA). From there Zeev worked on mainframe integration for SRL and founded a services company with special emphasis on integration and performance for the IBM i series (AS/400). In 2011, Zeev began business consulting to financial and governmental institutions on legacy systems and integration, and also teaches courses on these subjects and uses his deep background in the subject area to evaluate the technology and potential of start-ups for Venture Capital firms. He joined OpenLegacy in 2014 and is currently Chief Product Officer.

While the interest in microservices is fairly new and growing, Zeev has been involved with the concept of microservices since the early days of EAI and SOA. During the past 20 years he has seen first-hand the pitfalls and challenges with integration and legacy systems and believes microservices are a way to simplify and solve issues from the past.

Hans Otharsson - Hans is a global leader in legacy transformation programs with decades of experience developing, enhancing, maintaining, troubleshooting and transforming so called 'legacy applications' and associated environments. His global journey has taken him into countless environments and business scenarios, where his "straight to the point" approach has enabled him to bring true change and business driven transformation to his clients. His ability to quickly assess a situation and determine if an organization can bring value – and offer suggestions for other alternatives if needed - has made him a trusted advisor to numerous global organizations. Hans has many years' experience with 'legacy modernization' in senior executive roles at Consist Software Solutions, Ateras and also he founded ModernWiser, a consultancy helping organizations understand their legacy modernization options. At Software AG, Hans was responsible for all Professional Services Sales & Delivery in North America and Canada.

In his current role as Chief Operating Officer at OpenLegacy, Hans is responsible for corporate operations and client success. These dual roles truly capitalize on Hans' strengths of quality delivery, a customer first mentality, and solid industry experience - all of which are truly echoed in his mantra of "we measure our success on our client's success."

Introduction

The question answered by this book is this: “How do you accelerate delivery of innovative digital services from monolithic legacy technologies in a way that doesn’t add more complexity and layers?” In short, how can IT deliver on the demands of the business?

To answer that question, we will describe the latest technologies and approaches involving application programming interfaces (APIs) and microservices, and end with many examples of how they have worked for other organizations (chapter 8). Like the bank that delivered a new payment processing system 50% faster than other typical mainframe projects. Or the insurance company who could finally compete in online price quote comparison engines.

Our discussion is somewhat technical but mindfully written to clarify and demystify these concepts for *both* the IT and business audiences. Our goal is to facilitate meaningful conversations around these topics, building a bridge between these groups based on common goals and understanding.

While innovation and time-to-market have always been important, the millennial market adds a new sense of urgency. They are digitally impatient. They will not accept traditional banking

processes. They won't visit a branch, fill out a form, then wait for days and check their mail. Yet in many banks in many parts of the world, this is still the onboarding process. If you can't offer digital banking services via a mobile device, millennials will download an app from a competitor and be on their way.

If you can't offer digital banking services via a mobile device, millennials will download an app from a competitor and be on their way.

Demands set upon technology grow in the same escalated scale as the capabilities of technology. When technology becomes faster, consumers want things faster. As a result, most organizations find their business leaders requiring faster innovation, while the technical leaders still face many of the same technical challenges.

These days, a "modern microservice" is a fundamental aspect of legacy system integration. It is about reducing and bypassing layers. It is about rapidly accessing the system of record and avoiding the unnecessary middle layer. It is the realization of what we hoped Services Oriented Architecture (SOA) would have brought us decades ago. And, it is what we envisioned when we initially built those legacy back-end systems of record. In short, the modern microservice is the right approach that finds itself in the unique position of being in the right place at the right time.

However, these days, there seems to be a lot of confusion around microservices. Common questions include things as basic as “how are they different than APIs” and “wait a minute, why does that vendor definition sound different than this one over here?”

Truth is, it’s easy to be confused. Microservices have changed a lot over the years, and vendors refer to microservices in a wide variety of ways. So, what are they, should you care, and, if so, what can you do about it?

First of all, we will be talking a lot about “legacy systems,” “legacy monoliths,” and “application monoliths”. An accurate definition of these legacy environments would be systems developed in the past that still provide business value. They were built using the best available technology at the time, but often don’t meet today’s integration requirements and business needs. They are viewed as complex silos of tightly coupled business logic that require numerous abstraction layers to reduce and demystify so modern DevOps resources can simply access those business systems.

> Unlike typical application monoliths, microservices are small independently deployed services focused on a specific business function.

Unlike typical application monoliths (Chapter 1), microservices are small independently deployed services focused on a specific business function. Programs written in the microservices style are popular because they are easy to understand, develop, and test. Developing an application made up of individual services means that different teams can work in parallel. This makes it faster and easier to release a collection of related microservices.

Unfortunately, many IT departments with legacy systems, an ESB, or SOA still struggle to be as agile as needed in this ever-increasing global and digitized world.

You may find yourselves asking, “But, wait a minute—weren’t ESBs and SOA supposed to do that?” Yes, and in many cases they succeeded. Unfortunately, most IT departments with legacy systems, an ESB, or SOA still struggle to be as agile as needed in this ever-increasing global and digitized world. In many ways, microservices provide the benefit of SOA, while also removing many of the disadvantages (Chapter Two).

Microservices align well with agile processes that support continuous development and delivery. These aspects are needed in businesses where frequent data and program updates are required (Chapter Three). They are a natural fit for DevOps.

In the original use-case, microservices were an effort to push modularity to a new level. They operated primarily as behind-the-scenes components. Today's modern microservices are frequently customer-facing, highly integrated services used to create previously impossible combinations of application functionality. In large part, they leverage the capabilities of API contracts to interface with a variety of back-end systems. Most importantly, modern microservices can bypass layers of existing complexity and be implemented far faster than earlier approaches (Chapter Six).

This book is for any IT, DevOps, or business leader in an organization who is considering microservices as a way to quickly and efficiently leverage legacy data in modern technologies. May your digital journey lead you to IT efficiency, faster cycles, greater scalability and competitive differentiation.

May your digital journey lead you to IT efficiency, faster cycles, greater scalability and competitive differentiation.

Chapter One:

From Application Monolith to Microservices

If your organization can't innovate fast enough to satisfy business and competitive demands, consider a microservices approach to your application monoliths. If you're asking yourself, "what's an application monolith?" you're probably running one right now.

Typically, an IT department develops an application — usually a single core application or a number of related applications — and combines all the elements into one system as shown in Figure 1.

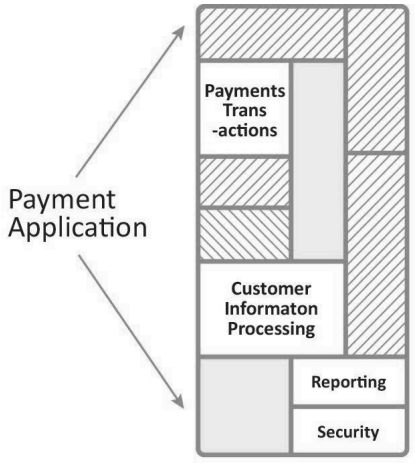

Figure 1. An example of an application monolith, where multiple elements and functions are combined into one application.

For example, functionality is mixed together in one big application; you might have a payment application that would include the customer information, payment transactions and additional functions such as reporting and security. Sometimes this

approach has benefits, and some applications are simpler to develop in this way. On the other hand, monoliths create a lot of inefficiency and code-usability challenges.

Slow Releases – The Enemy of Velocity

While this monolithic approach may have worked in the past, the world has changed. Most organizations have numerous development teams, all trying to create, update, and test their code before release (Figure 2).

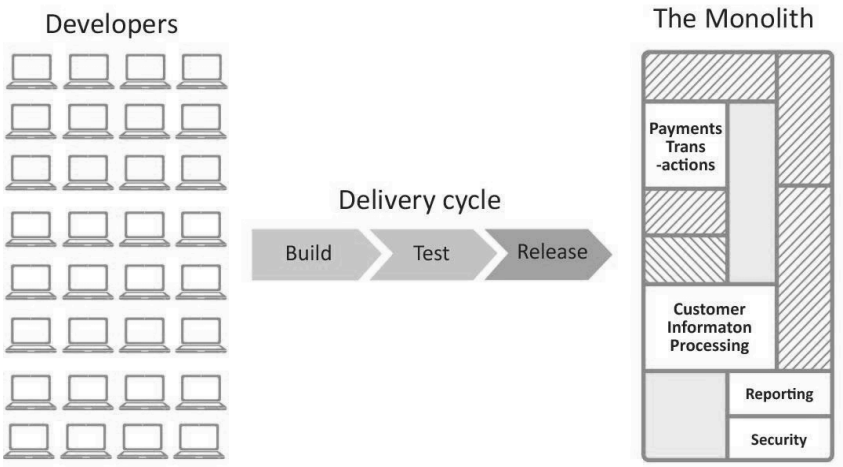

Figure 2. Application monoliths are mostly incompatible with today's Agile and DevOps environments. The build, test and release cycles are usually far longer than what is required for velocity and scale.

Releasing to production once or twice a year used to be the norm, but it no longer serves most businesses in today's fast-paced global

economy. Businesses rely on Lean and Agile approaches when they need faster release cycles — yet, the application monolith is the enemy of such velocity.

Problems with Shared Code and Data

An application can be monolithic if you have software coupling, such as shared code or shared data (see Figure 3).

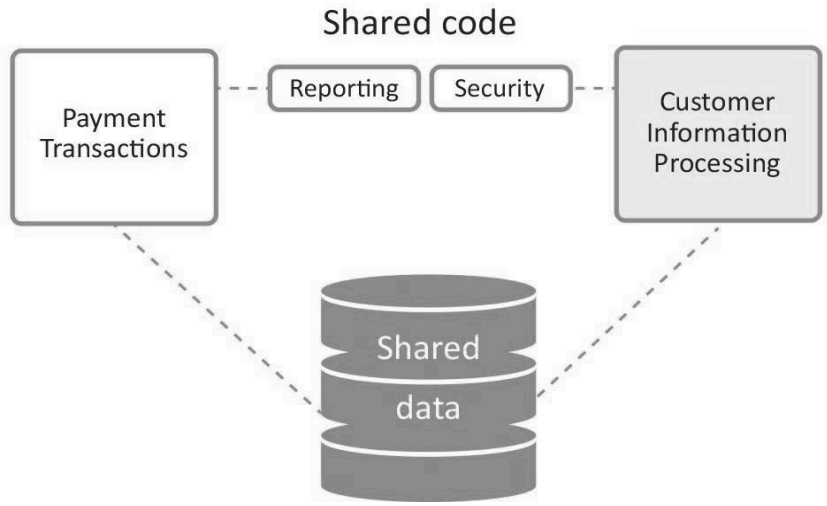

Figure 3. In a typical application monolith, different modules may share the same code or data. Therefore, any change to that code or data requires testing of the entire monolith.

Shared code means central routine data is shared through the application by many modules. For example, a credit card company likely has a central digit validation routine that checks the validity of the digits on a credit card. Nearly every module in the

application can use this routine — whether it is the approval process, the alert system, or the customer statement — because everything needs to check that the card number is valid. Consequently, if you make a change to this central routine, then you have made a change to each and every function of the application. If something goes wrong with this module, then the application fails.

This inter-connected “house of cards” is precisely why most organizations find it difficult to innovate with legacy systems.

Shared data poses a similar problem. For example, customer information is used by many components of the monolithic application. Sharing both code and data results in very strong coupling. Although it is simpler to maintain one code base or one database, it is harder to make and manage changes.

Since change can be unpredictable, companies need extensive and time-consuming testing — often taking months — to make sure that code changes are stable enough to go into production. In fact, this is such a huge concern that some organizations will avoid fixing bugs because they are concerned about the aftermath.

This inter-connected ‘house of cards’ is precisely why most organizations find it difficult to innovate with legacy systems.

No Scalability

To improve scalability, companies have tried to break the monolith into major application components that run on different server images or platforms. However, many applications are so coupled with code and data that they cannot easily be modernized. It is not possible to deploy one component of application functionality in one place and another piece of functionality somewhere else. This puts a cap on scalability.

Scaling issues, lack of agility, long release cycles, slow innovation, complex changes, and risk management are all well-known challenges of the monolith (Figure 4).

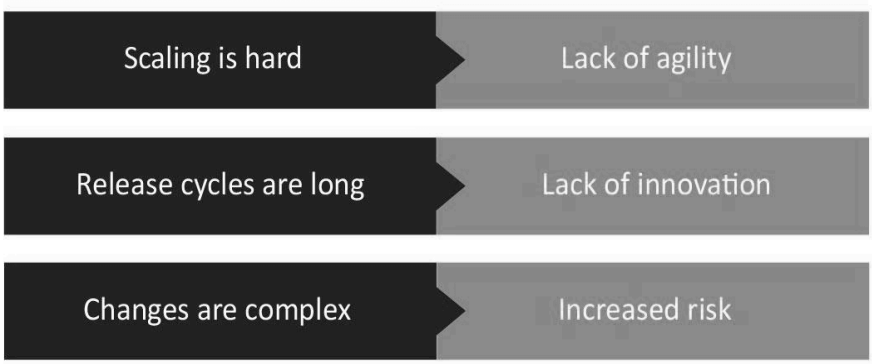

Figure 4. Typical challenges of the legacy application monolith.

What is a Microservice?

Microservices are an application architectural style that is a little more than a decade old and interest and adoption has been growing rapidly (Figure 5).

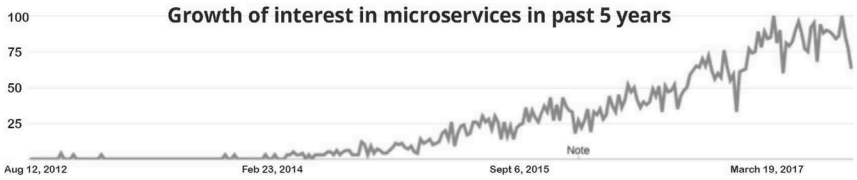

Figure 5. Interest in microservices architecture has grown rapidly since 2012 as a way to solve common problems with application monoliths. Source: Google Trends

The idea of microservices emerged from experiences with service-oriented architecture (SOA). Unlike SOA, microservices structure an application as a collection of independent services that are narrow in scope and communicate using protocols that are very efficient. Boiled down to their essence, microservices are a formal architectural style for decoupling business functions from a monolith. When you also encapsulate an Application Programming Interface (APIs) in the microservice, you avoid many pitfalls of monolithic designs.

For example, data and processes are often locked up in current systems of record, such as the customer information and credit

card verification process utilized in your application monoliths. Microservices include APIs that are purposefully limited in their functionality. For example, a payment system would not exist inside of a microservice. Instead, the payment system would be comprised of a mesh of microservices – one to get customer details, one to transfer money, and so on. Each service is loosely coupled, allowing the overall application to function even if one service went down, allowing different services to exist on different servers and different clouds, or allowing different components to scale independently (Figure 6).

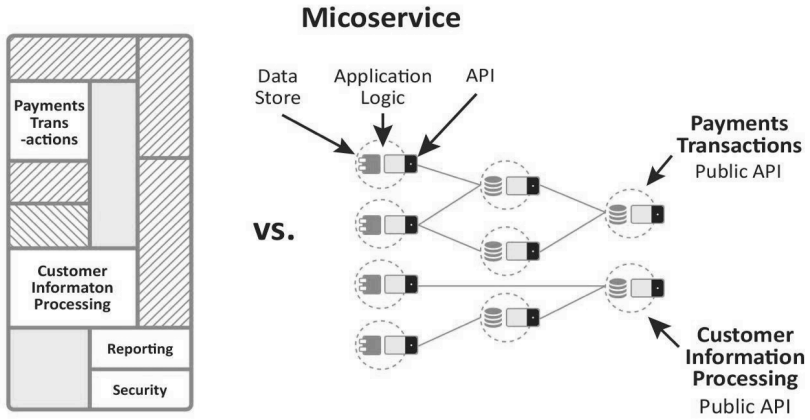

Figure. 6 Whereas a typical application monolith is tightly coupled, microservices decouple independent business functions into separate services so that changes to any one function will not interfere with the other functions.

Breaking Up Is Necessary Sometimes

Breaking up an application into different smaller microservices improves modularity and makes the application easier to understand, develop, and test. It also enables parallel development by allowing small autonomous teams to develop and deploy their services independently. Lastly, microservices allow the function of an individual service to emerge through continuous improvement supporting continuous delivery and deployment.

Microservices Speed Development on Legacy Applications

The major benefit of microservices isn't just the speed of development, it's the concurrency of development. Developers can work on different parts of the same application at the same time. This has tremendous benefits for companies working with legacy applications.

Modern application development demands a speedy pace, but working with legacy applications is usually slow and laborious. Microservices remove that burden. It is possible to expose business functions via a microservice, if you consider the legacy application as a data store, where you encapsulate the application logic (business functions) within an API. In this manner, it's possible to use the COBOL output of a legacy application in a format that's understandable by a REST API.

> The major benefit of microservices isn't just the speed of development, it's the concurrency of development.

Developers can create other microservices representing features related to the legacy application and connect them to the legacy application without writing or modifying any legacy code. In this way, it's possible for one development team to work on speedily adding new features to a legacy application, and another team to work on maintaining the legacy codebase. All the while, neither team needs to communicate or slow down their work based on the demands on the other. Microservices enable rapid development for projects involving legacy applications.

Monitoring Microservices

Before the microservices era, it was common for developers to create a single monolith featuring hundreds of APIs, and then run those APIs in a single container. The drawback to this has been discussed above – if one single component went down, the entire application would follow. On the other hand, one (extremely faint) upside to this was that it was relatively easy to monitor the condition of the application, since it was all in one place.

In the era of modern microservices, it's possible to have an application that's comprised of hundreds of services. Some of these services may not be in the same server, or in the same cloud.

While the resulting application will be extremely fault-tolerant, it's worth asking how to monitor your microservices before switching to that approach. Consider finding a software vendor or solution provider, such as OpenLegacy, that's equipped to do analytics and monitoring on hundreds of microservices running in parallel.

With these benefits in mind, it is no surprise that leading industry analysts are suggesting organizations take a closer look at microservices. Forrester writes that microservices have an important role in the future of solution architecture[1], highlighting faster software delivery, greater operational resilience and scalability and better solution maintainability as the main benefits. Gartner indicates that microservices architecture enables unprecedented agility and scalability.[2]

[1] *Forrester, Microservices Have An Important Role In The Future Of Solution Architecture, July 2015*

[2] *Gartner, Innovation Insight for Microservices, January 2017*

Chapter Two:

Microservices & Service Oriented Architecture (SOA)

When discussing the advantages of a microservices architecture compared to monolithic applications, it's easy to wonder why no one took this approach in the first place. After all, upwards of 90% of enterprise applications in the world today are monolithic.

At the time they were created, monolithic design was simply the best approach available to meet technology and business needs. However, as business needs have evolved and the demand for greater agility has intensified, developers and IT teams have been tasked with breaking free of these monolithic architectures.

The resulting approach, at least in recent years, has been a service-oriented architecture (SOA). However, while SOAs solved many problems, they didn't solve them all. The term "microservices" is a subset of SOA terminology (Figure 1).

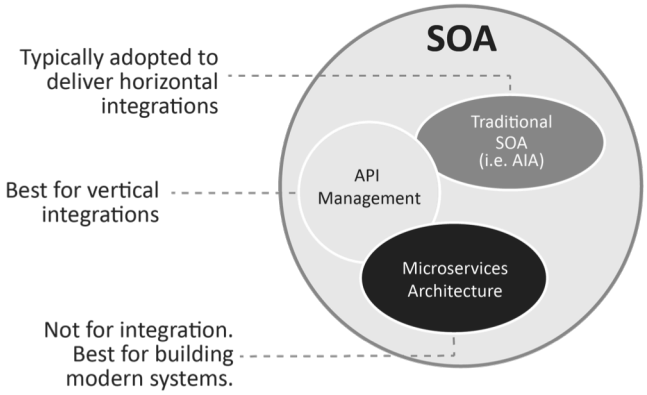

Figure 1. Microservices and APIs can be thought of as a subset of Service Oriented Architecture.

The SOA Integration Challenge

The promise of SOA initiatives was to extend the reach of core business functions while reducing the internal expenses and complexity that grew alongside monoliths. SOAs would achieve these goals by breaking the core functions of a monolith into web services using protocols like simple object access protocol (SOAP) and eXtensible markup language (XML).

SOAP was built for universal application communications. Because it's based on XML, SOAs designed with SOAP could, in theory, be used to create an agnostic integration layer. Rather than struggling to piece together various proprietary systems, these

protocols would open monoliths on different operating systems so they could work together.

An agnostic integration layer would let system administrators connect pieces of a monolith to an enterprise service bus (ESB) to achieve an agile, plug-and-play SOA. In some cases, this approach succeeded. There's a great deal of so-called "legacy SOA" out there in the world which still provides value. In these cases, however, the value that SOA provides is in behind-the-scenes server-to-server communication that mostly aids developers. When it comes to serving customers – and their rapidly changing demands – SOA isn't always up to the task.

The problem is that the ESB oversees the messages that achieve service integration to reach their destination. This communication isn't as simple as the SOA vendors may have promised.

When it comes to serving customers – and their rapidly changing demands – SOA isn't always up to the task.

Take, for example, an SOA integration approach with a core banking application. This involves a message going from your core application on your mainframe to a branch office server. If the business logic states a message is only relevant for one business day, administrators must decide whether it must be

moved to a queue, logged, or disregarded when the day passes. This is a common scenario for core banking applications in an SOA, but how well does it work?

In this example (and others like it), the ESB must know whether a business day has passed or not to make the right decision about where to send the message. This means the integration requires an algorithm. Even if it's a simple algorithm, the ESB can't simply rely on a universal rule for sending messages between the monolith and external integration.

When administrators introduce new business logic into the monolith integration, it turns what was supposed to be an agnostic service layer into a new application layer thus increasing complexity. Instead of simply integrating the monolith into a new digital service, the enterprise ends up with a larger monolith — one that includes the mainframe and all the new integration stacks that have been added. Despite the promises of SOA, the integration problems result in increased time spent on maintenance, greater complexity of code and software, and the continued growth (not elimination) of monolithic applications. The first rule of effective integration is "smart end-points and dumb pipes." Building logic and layers into the service layer breaks that rule, adds to the overall complexity, and adds another legacy application to your portfolio.

Trying to optimize the SOA approach will only result in larger monoliths. Taking a new approach with microservices architectures will help realize the original promise of SOAs (Figure 2).

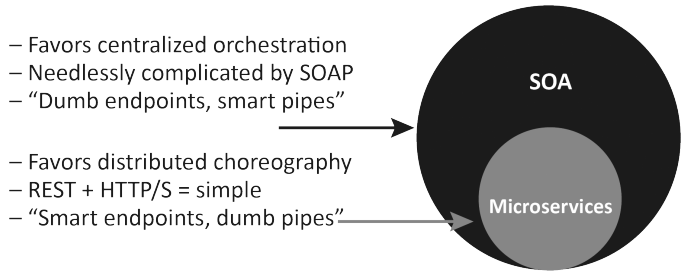

Figure 2. Microservices deliver on the original promise of SOA.

Improving SOAs with Microservices

For two decades, CIOs have tried to transition away from monoliths by taking traditional approaches to integration, only to find they've doubled down on legacy investments. All of these integration stacks end up coupled to legacy systems and only result in more work for the IT team — and a less agile organization.

The integration step of SOA was never intended to include business logic. Trying to force business logic into this approach

leads to workarounds and extra effort just to achieve a less-than-ideal result.

The key factor here is how to incorporate microservices without adverse effects. Luckily, microservices architecture can be forged from SOAs by introducing new principles.

One of these principles is context mapping. When replacing an existing SOA, teams must consider the size and scope of the new microservice and apply proper contextual boundaries. For example, SOAs that typically sought broad integration with digital services should be broken down into smaller domains to simplify operation.

Another key principle is the idea of a shared-nothing architecture. Too many SOA integrations create sprawling dependencies that create complexity in the tech stack. Microservices avoid these cross-service dependencies. For those looking to move to microservices from an existing SOA, it’s important to look at the list of dependencies and work toward standalone functionality.

Ultimately, the goal should be to refactor monoliths in a way that shifts the IT stack toward microservices. If you take the right approach, you can make the most of both microservices and APIs within your SOA.

Chapter Three:

Microservices vs. APIs

We've already explained how to differentiate SOAs from microservices, but differentiating APIs from microservices is a whole other question. Businesses are asking themselves the question, "will microservices allow us to create more valuable products?" They're also asking if it's necessary to master APIs before pursuing a microservices approach. Perhaps that is not even the best question. Both CEOs and technology buyers will use microservices and API terminology interchangeably. The difference is not simply an academic question, however (Figure 1).

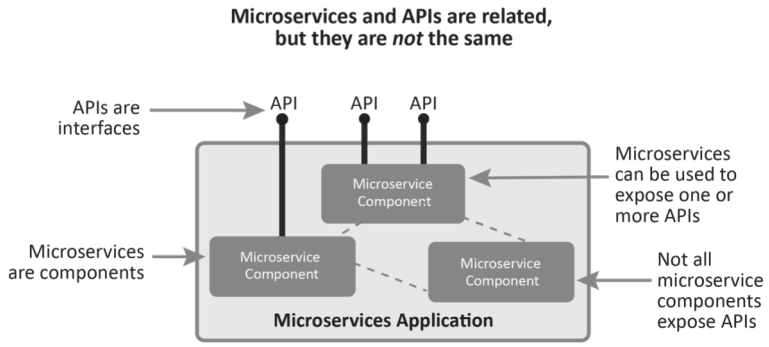

Figure 1. All microservices include APIs, but not all API's are microservices.

The difference between microservices and APIs is important for businesses that are beginning to plan their participation in a more digital era. You can implement microservices without exposing APIs, and you can create APIs without using microservices. The option you choose depends largely on your business needs.

So let's try to be clear here on the differences and similarities of APIs and Microservices. The fundamental concept behind a microservices architecture is to break up your application(s) into many small services. Each of these services will typically have its own:

- Distinctive business-related responsibility
- Execution Process
- Database
- Versioning
- API
- UI (User Interface)

The fundamental concept behind a microservices architecture is to break up your application(s) into many small services.

A microservice should expose a well-defined API (Figure 2).

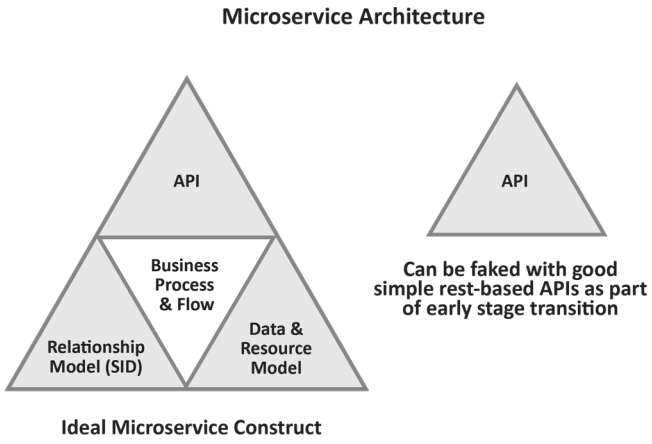

Figure 2. There are times when an API is sufficient – and preferred – and times when a microservice is better. A microservice will expose an API, but also include other elements.

Microservices is the way you want to architect your solution, while the API is what the consumer sees, keeping in mind you can expose an API without a microservices architecture.

A rule of thumb to follow is to keep your services small and have lots of small services versus building larger services. Then you need to understand that a microservice can use a representational state transfer (REST), message queue or any other method to communicate with one another, so REST is orthogonal (independent) to the topic of microservices (Table 1).

Table 1. When to choose microservices, APIs or both.

API	Microservices	Application Status
		Your application is slow to change and difficult to integrate with. Time to market will be slow, and this application is not truly an asset to your Digital Transformation Initiative.
		Your application is still slow to change, but it is now easy to integrate with. Meaning your time to market, your ability to access this legacy application is easy and fast. This application now can be an effective contributor to your Digital Transformation journey.
		Integration nirvana – your application is easy to change, rapid to integrate. Most importantly your ability to add on top of your legacy application new products and services is optimal.

Microservices a Streamlined Architecture for APIs

From a developer perspective, microservices are an approach that enhances the performance, and thus the value, of the APIs and applications they create. Remember that at its heart, an API is still just a contract. The API receives a certain input and delivers a certain output. Depending on whether the API is built with microservices architecture, that output will arrive in a certain way. Microservices impose a set of rules on APIs that make them simpler, more modular, and more functional.

Microservice	**API**
One function	Many functions
One data store	Many data stores
Simple communications	Complicated pipes

One Function vs. Many Functions

A standard API might be built to behave several ways. For instance, it might accept one kind of input, such as a customer phone number, and it could return a customer's ID and address. If you give the same API a customer ID, it might return their credit card and phone number, with further different outputs depending on what other information it gets.

Microservices strictly limit the kind of information that an API can return. A microservices API will only return one or two pieces of information depending on the input. A different microservice

handles anything else. The reason for this rests on the way that microservices relate to their data stores.

One Data Store vs. Many Data Stores

For an ordinary API to return so many kinds of data, the component implementing it might have to connect to many different data stores. This is where APIs can begin to slow down the development process. If one of those connected data stores is updated, it may change the information that gets returned, and vice versa. If more than one API is connected to the same data store, any changes will affect both.

Microservices return only one or two kinds of data, and the best practice is to connect them to only a single data store. This way, a development team needs only test a single API before pushing changes into production. This eliminates cross-functional disputes and prevents duplication of efforts.

Simple Communications vs. Complicated Pipes

Once their information is retrieved, APIs need to do something with it. Sometimes this means just transmitting the information to the end user, but more often this means it's handed off to another automated system. In an ordinary application, the API can call any data store or any other API as determined by the underlying application logic. This necessarily makes the application more tightly tied together, less modular, and more monolithic.

In a microservices architecture, the application logic tied to a particular API can only call on other APIs. This simplified communication prevents the application from breaking when a microservice is changed or removed. This added flexibility further simplifies and streamlines the development process.

An ordinary API implementation stands by itself. It can accept any commands, connect to any database, and make calls to any other application or service. This seeming flexibility, however, can actually make applications more monolithic, less flexible, and more difficult to work with. By contrast, microservices encapsulate APIs with a set of rules and components that ultimately liberates applications from monolithic constraints.

A simple API in a microservices framework can be created in one day using proven technologies.

In summary, companies are beginning to realize that simply having APIs doesn't make them an especially innovative or forward-thinking organization. If it takes four months to create and implement an API, then there's no inherent velocity in development. Microservices change the conversation. A simple API in a microservices framework can be created in a day using proven technologies.

Depending on their requirements, companies that adopt microservices right now will have an enormous first-mover advantage over companies that rely on traditional APIs. Microservices adopters will be able to build applications with more intrinsic value and add features faster than their competitors. In other words, if you haven't considered microservices already, be prepared for that to change very soon.

Chapter Four:

The Anatomy of Microservices Architecture

A microservice is built in a specific way that incorporates three parts: The API itself, an application logic unit, and a data store (Figure 1). Microservices architecture allows for some variation within this blueprint while adhering to the following precepts.

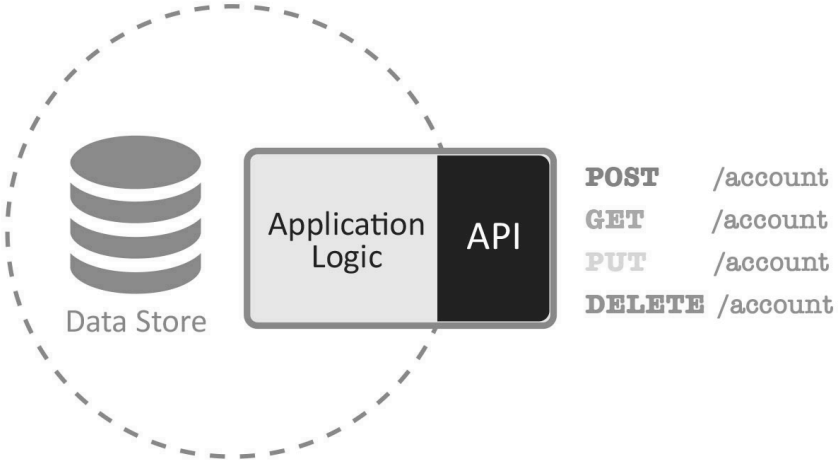

Figure 1. A microservice includes three parts: a data store, application logic, and an API.

Microservices Anatomy: The API

Every microservice includes an API, but that API has to be written in a specific way. The microservices architecture favors extremely narrow and specific tasks, and so the embedded API will usually have just one role, representing a public contract. For example, the

contract for an agent desktop might involve an API that provides a customer's name, telephone number, and address when given their customer ID.

> The microservices architecture favors extremely narrow and specific tasks, and so the embedded API will usually have just one role, representing a public contract.

Some APIs may have an expanded role – say that the customer forgets their ID, so the agent can retrieve it by providing the customer phone number instead. Going much beyond this usually represents bad practice, however.

In addition, the API itself should retain past functions when given new ones. In other words, the development team may decide to improve the functionality of their API, but they can't replace it. Doing otherwise may break workflows or cause cascading failures.

Microservices Anatomy: Application Logic

In microservices, the application or business logic component adds a measure of intelligence to the API. For example, imagine an ecommerce platform built on microservices. A customer clicks on "place an order". This signals one microservice to check that the customer creating the order has a valid account, billing

information, and address. Once this is finished, the microservice needs to pass on a certain amount of information:

- Was the application able to validate the customer?
- If no, why not?
- If yes, call the next microservice in sequence to create an order.

Microservices Anatomy: Data Store

Best practices for microservices architecture holds that microservices never share data – a microservice encapsulates its data store, and other APIs will never call that data store directly. This lets developers make changes to their data stores without creating affecting other microservices, greatly speeding up the development and testing process.

Another cardinal rule of data storage in microservices is to use the database or databases which best suit the use case of the microservice itself. This allows developers, for example, to use both a SQL and a NoSQL database within the same application, retaining the benefits of NoSQL without giving up ACID (atomicity, consistency, isolation, durability) transactions and other positive aspects of relational databases. This is mirrored on the API side, with polyglot programming enabling application logic and APIs to be written in whichever languages add the correct blend of functionality and efficiency.

Microservices are Like A Mini-Application

In summary, a microservice can be thought of as incorporating the aspects of three-tiered client-server architecture commonly used in web applications such as ecommerce: a presentation layer, a business layer, and a data layer.

- **The Presentation layer** - which is the mechanism that is used to frame the communication in a three-tier architecture. The API, in the Microservices paradigm, is the contract which defines the mechanism of communication.

- **The Business Layer** - functions the same within a 3-tiered architecture and a Microservice architecture. It coordinates the start and end of tasks and activities along with managing the communication with other services.

- **The Data Layer** - manages the packaging and exposure of the information (data) to the logic of the Business Layer, which in turn is passed to the API (Presentation Contract)

Although the three-tier application contains layers of horizontal separation, microservices add vertical dividers, exposing themselves to other microservices only through the presentation logic layer. This is the major anatomical difference between

microservices and monolithic applications. By closing themselves off from the rest of an application, microservices can be administered by a single team over their entire lifetime, eliminating the possibility misplaced business logic, duplicated efforts, and other negatives.

Chapter Five:

Microservices Best Practices

Microservices approaches have evolved to become far less complex than they were a few years ago. Instead of modules preoccupied with connectivity and message passing, modern microservices are data applications that can be created more easily than in the past, then used stand-alone or combined into applications using APIs. They make important data and processes available in new ways without disrupting the systems of record that they access.

When you invest in modern microservices, you should expect scaling with ease, rapid release cycles, simpler changes, increased agility, faster innovation and lower risk. These motivations are behind these seven rules of microservices:

1. Bypass layers where possible
2. Access only public APIs
3. Use the right tool for the job
4. Secure all levels
5. Be good citizens yet have great police
6. This is not just about technology
7. Automate everything

Rule Number 1: Bypass Layers Where Possible

Many readers may perceive or have first-hand experience of integration microservices as being complicated to create. Whereas in the past microservices creation required navigating through the complex layers of your existing architecture (Figure 1), modern microservices bypass these architectures to connect directly with your monolithic legacy systems, such as mainframes, midrange systems, and databases (Figure 2).

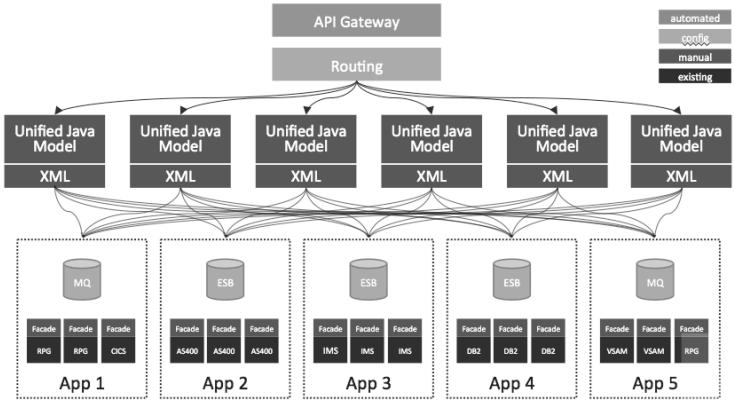

Figure 1. Typically, integration microservices have been hard to create, largely due to complex, manual effort.

Some software (such as OpenLegacy) consumes the logic of the legacy system so that the best method of connecting to the system can be determined. Then at run-time, that information is used with pre-built connectors to automatically connect with the legacy system, and bypass as many layers as possible (see Rule Number 3 – Polyglot Back-End).

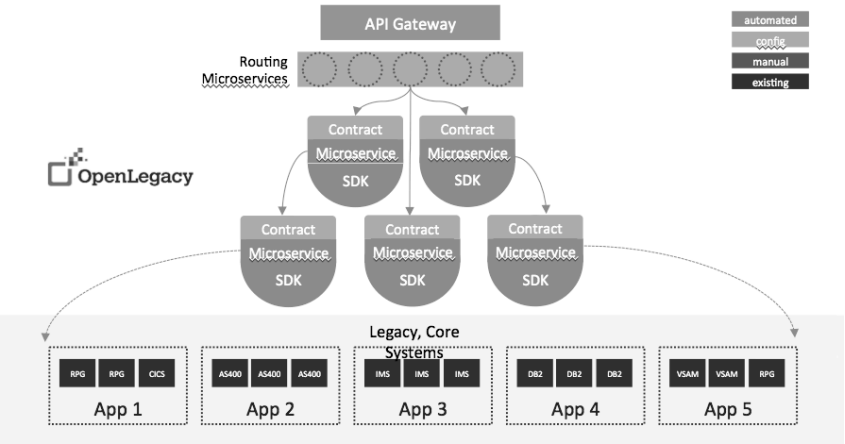

Figure 2. Some vendors (such as OpenLegacy) bypass architectural layers to connect directly with existing monolithic, legacy systems. Furthermore, many manual processes are automated thereby simplifying and speeding the process.

Microservices can be created very quickly and without complexity, and without special programming skills such as COBOL or RPG, or invasive changes to underlying systems.

Since the value of microservices can't be achieved without first creating them, this point can't be overstated. In our experience, the biggest hurdle for organizations adopting microservices or APIs for legacy monoliths is the time and effort for creation.

Rule Number 2: Access Only Public APIs

Delivering new business applications using Public APIs is fundamentally changing how software is created and delivered to

the market. The public API, built with REST or SOAP, has a contract governing its access, so the rule is that you must only interact with the contract of the microservice. This is important because for a microservice to be coupled or combined, it has to be utilized in a specific way to preserve its modular aspect. When using the service contract, you avoid problems that could arise from reading the service's database or message queue directly. If you bypass the contract, you are dependent on the physical attributes of the service and can run into problems when there is a change of the code in the microservice (Figure 3).

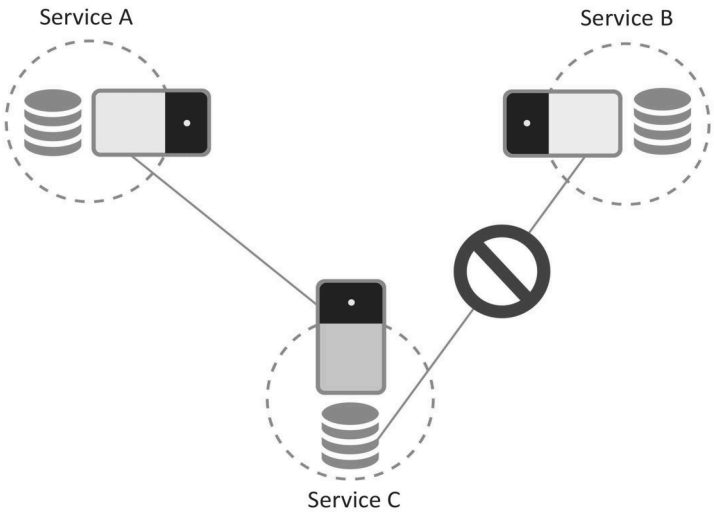

Figure 3. Only interact with the API contract of the microservice. Otherwise, you may run into problems if there is a change in the microservice code.

How does a microservice change and evolve? There is a process for changing and growing the microservice. This is handled through versions of the contract, so organizations can have some applications that use one version of the contract whereas other applications will use a different contract.

For example, microservices can break up an application into functional components, which are combined with other microservices to create an external API. By creating this kind of "function network", these combinations of microservices utilize a SDK to access the legacy system like a mainframe. These microservices are used to create new business applications. The new contracts they use are not on the mainframe, but rather inside the microservice itself that accesses the mainframe.

Having a function network makes it easy to change, enhance and add new business logic. Modern microservices create a layer of protection and abstraction on top of the legacy system. At the same time, they give users a lot of flexibility and agility without changing the legacy application.

Rule Number 3: Use the Right Tool for the Job

Many organizations are starting to adopt the best-tool-for-the-job approach by developing an integrated collection of supported products and technology. This may include a "must use" list where there is one main tool that you must use to address a specific need.

There is also an “available list”, which is a list of approved alternative solutions. Programmers can pick and choose whichever tools or product best fits their needs — SQL database, sequential file system or in-memory data system. This is known as polyglot persistence (Figure 4).

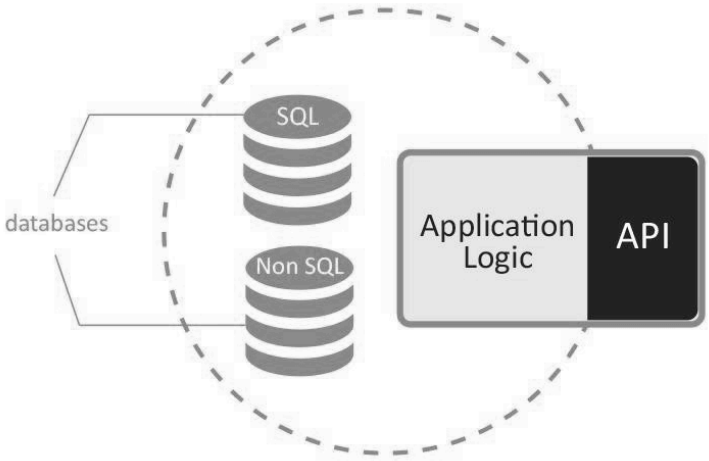

Figure 4. Polyglot persistence enables developers to choose the right tool for the job.

Making the right choice with a programming language is just as important as choosing the right data-management option. Choosing the best language to complete the task is known as polyglot programming. Just because an organization has made a commitment to a specific programming language should not mean that programmers must do every procedure in that language as long as they properly invoke the microservices using its contracts. For example, in some applications, even though Java might be the

company standard, C or javascript may be the best choice (Figure 5).

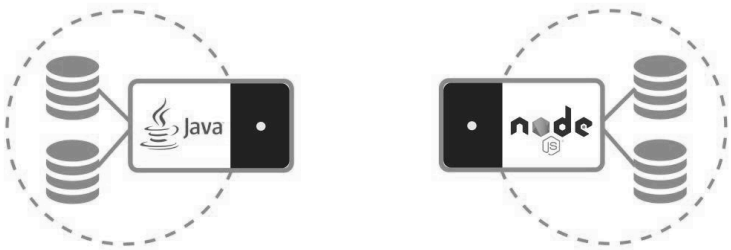

Figure 5. Polyglot programming means programmers can choose the right programming language for the job.

The polyglot backend rounds out the trifecta of polyglot architectures. This allows microservices to flexibly access one or more back-ends in the microservice depending on the needs of the application being developed (as described in rule number one). The polyglot backend does not take the form of an integration layer, but instead is a pure microservice implementation without microservices washing – the practice of selling a product as a microservice when it doesn't actually fit microservices characteristics (Figure 6).

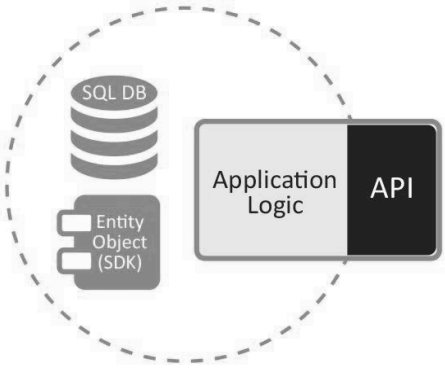

Figure 6. Polyglot backend capability is a powerful feature because the back-end can be a variety of sources like a mainframe, midrange system or relational database on a distributed platform. Support for diverse back-ends makes it possible to create microservices with significant ability to integrate data and processes from previously disparate sources.

Rule 4: Secure All Levels

Since microservices are separate runnable units, they do not enjoy some of the benefits of a security framework, which are shared by all components in a monolithic application. Because there is often not a shared security mechanism, developers compensate for this shortfall by running the microservice behind an API gateway[1] which supplies a lot of functionality while also playing the role of a firewall.

When an API gateway is implemented, many organizations assume that everything behind the API gateway is secure. Once a cyber threat penetrates the gateway, there are no additional security mechanisms to challenge it. Microservices require an in-depth defense that is not limited to one layer of security. Communications between microservices should be protected using encrypted SSL.[2] oAuth[3] should be used for user identity and access control. It is important to properly handle JSON.[4] This protocol replaced XML but has weak data typing capabilities. JSON has limited features to help with data validation, which means that JSON has vulnerabilities that must be addressed by logic in the microservice (Figure 7).

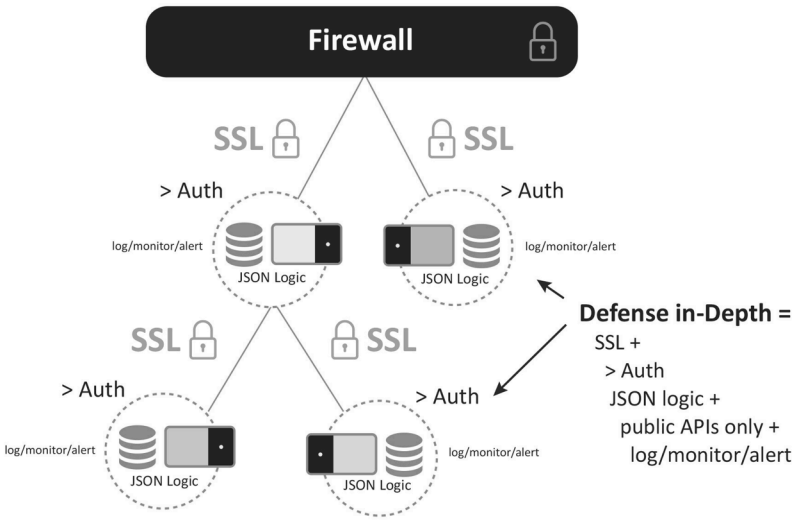

Figure 7. Due to their unique nature, microservices require multiple levels of security.

Another secure-all-level strategy is never to allow microservices to run on the public network because it typically is not secure. Additionally, programmers need to embrace certain guidelines, such as logging every significant event, implanting automated monitoring, and generating alerts when needed. Programmers don't need to reinvent the wheel in terms of their security, development, and systems management solutions. Instead, they should use trusted tools and frameworks with which they are already familiar. Examples include:

- Apache Log4j - A fast, reliable, and flexible logging framework written in Java
- oAuth - Open protocol to allow secure authorization for REST APIs, web, mobile and desktop applications
- EhCache – Widely-used open source Java distributed cache engine
- Angular - Open source development platform for web applications
- Freemarker - Open source, Java-based template engine. Templates are a structured format, created by Freemarker, where programmers enter data when generating entities

In addition to the other security considerations discussed — firewall, SSL, JSON, use of a public network, logging, monitoring and alerting – some vendors (such as OpenLegacy) have a feature called in-service security. This adds another secure layer to every

microservice in an application by building upon LDAP and oAuth authentication. That layer provides data-structure security by restricting access to specific API fields according to the security level. It adds data-content security by restricting access to data values according to the security level.

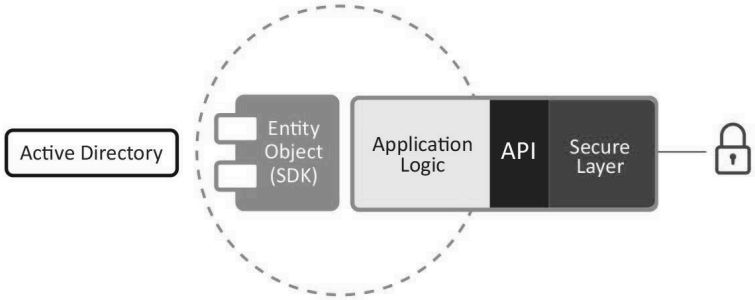

Figure 8. Security needs to be implemented at the service level.

Rule 5: Be Good Citizens, Yet Have Great Police

Being a good citizen of the microservice ecosystem means that when you write a new microservice that uses the contract of another microservice, you should be aware of the current usage of that microservice. You should approach the team supporting that microservice to tell them of your plans and needs. If you plan to use it extensively, you might impact their SLA[5] and they will have to take steps that might include increasing pool data size or enabling caching[6] (Figure 9).

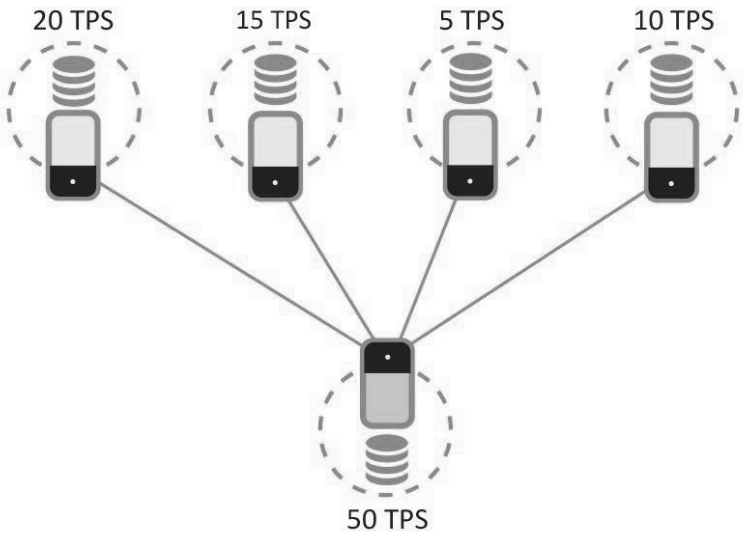

Figure 9. Being a good citizen means that developers should work together when their microservices are used together as part of new business function for the company.

You need to be a good citizen, but you also need great policing tools. You need to measure SLAs, collect logs & traces and throttle[7] unruly workloads. It is also important to collect both internal metrics ('which services were involved and what is their response times?') and user-experience metrics ('how long from click to data?').

When your microservice interfaces with mainframe applications and data, you should take actions to protect the monolith. Not only does each microservice keep logs and provide tracing ability, but also there is a caching mechanism that improves the performance

of searches by keeping the most frequently used data in memory. When too many requests are sent to a host, some software can specifically throttle access to the monolith. The throttling feature can limit the number of requests that a single client can send to the application APIs per time unit (Figure 10).

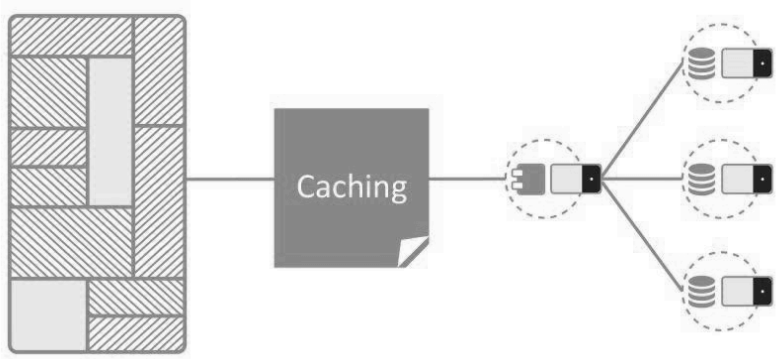

Figure 10. Both caching and throttling are tools used to minimize the impact of the microservice on the monolith.

Rule 6: It's Not Just About Technology

Before developing microservices, consider the organization of your team. Microservices thrive when you organize as a cross-functional team[8] where each team has any number of different skills and these teams are as self-sufficient as possible. Martin Fowler says that "siloed functional teams lead to siloed application architectures" (Figure 11). Cross-functional teams work better with microservices APIs because they are organized around the capabilities they are creating and managing (Figure 12).

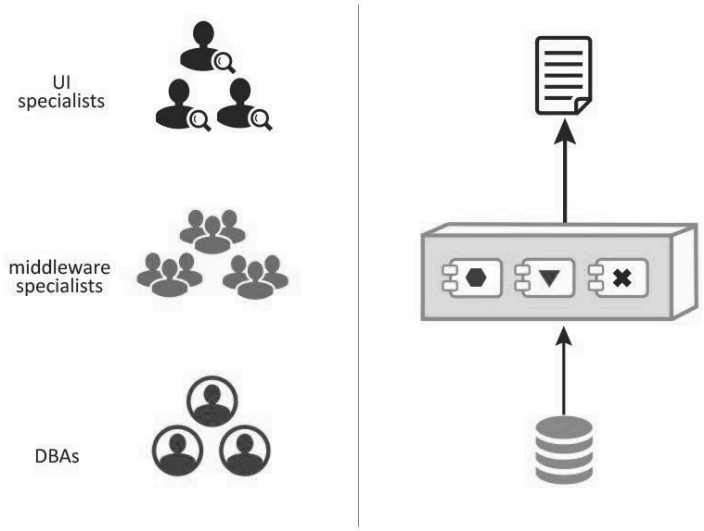

Figure 11. Siloed functional teams lead to siloed application architectures.

Before developing microservices APIs, consider the skill set of your team. Java is an important skill to have on the team, but other language and middleware skills are also needed. For example, some tools automatically generate microservices in Java, and Java skills are required only when you want to change the standard output of a microservices API. Surprisingly, because the legacy application and data access are handled automatically, the microservices implementation team does not have to include legacy skills. This means that the number of cross-functional teams is not limited to the number of legacy-skilled people and the labor pool is therefore much bigger.

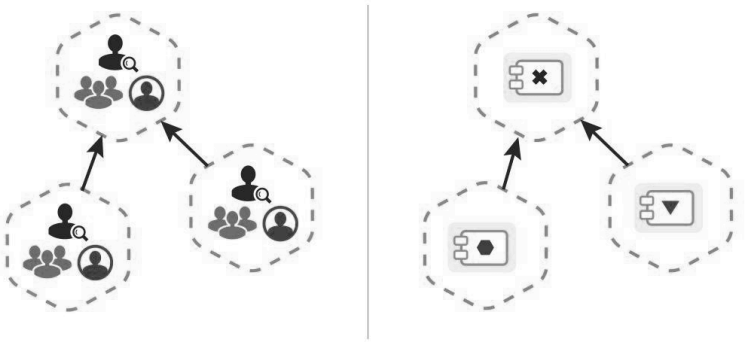

Figure 12. It is important to organize teams around business functions they develop and deploy as microservices so the development can be fast and responsive.

Agile approaches[9] to development are a natural fit for microservices. The cross-functional team can rapidly create the public APIs that result from their microservices. DevOps[10] is also a good fit for microservices as both developers and support technicians belong to the same cross-functional team. Being on one team also breaks down any organizational barriers that inhibit the use of common tools and procedures.

Rule 7: Automate Everything

Automation is a proven way to improve quality and lower risk in IT. Gartner[11] writes that automation is the next IT frontier. Testing is a good candidate for automation because, during microservice development, manually repeating the necessary tests is costly and time-consuming. Automated software testing can reduce the time

to run repetitive tests from days to hours, providing more comprehensive and consistent results.

Automation leads to a much more robust IT environment as well as a greater opportunity to change software in a low-risk way. Some microservices vendors are inherently more automated than others. For example, some vendors make it possible for the deployable units - the actual microservices themselves - to be standard Java entities. There is nothing proprietary about them — they are exactly what an experienced Java programmer would create.

Additionally, with these types of microservices, solutions like Jenkins and Maven are available for you to use. With Jenkins, you can automate many parts of the software development process by making use of continuous integration and continuous delivery steps. With Maven you can manage a project's build, reporting, and documentation. Jenkins and Maven are just examples. When choosing a microservices vendor or tool, look for a solution where no proprietary deployment, testing, or versioning solutions are needed, and you are encouraged to use whatever best-practices open-source solution you find best (Table 1).

Table 1. Sample of the Technology Stack of the Tools that are used to create, manage and deploy microservices (using OpenLegacy as an example).

Tool	Description
Spring	Spring is an application framework used to build simple, portable, fast and flexible JVM-based systems and applications.
SonarQube	SonarQube, a continuous code quality tool, provides the capability to show the health of an application as well as to highlight issues that may have been newly introduced.
Jenkins	Jenkins is an open source automation server that provides hundreds of plugins to support building, deploying and automating any project.
Maven	Apache Maven is a software project management and comprehension tool that can be used for building and managing any Java-based project.
Git	Git is an open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Sonatype Nexus	Sonatype Nexus is a tool to organize, store, and distribute software components.
Cloud Foundry	Cloud Foundry is an open source cloud application platform for developing and deploying enterprise cloud applications. It automates, scales and manages cloud apps throughout their lifecycle.

It is important to use a standard technology stack because the technologies are tested. There is also a good deal of expertise and documentation available on how to set them up properly to ensure security and high performance. Certain development situations may call for a faster or more robust database. Or there may be contractual or regulatory requirements for certain types of hardware, operating systems, and server software. These cases are rare, and most microservices function well on a standard technology stack.

The best practices in this chapter are common sense measure, but some will require cultural and technology changes in the IT organization. “Access only public APIs” and “secure all levels” focus on prudent measures to protect data resources. “Using the right tool for the job” and “this is not just about technology” likely require organizational change to be impactful. “Be good citizens

yet have great police" and "automate everything" focus on a mix of organization and technology measures that are easy to implement by following IT procedures.

[1] *API gateway – API management, Wikipedia*

[2] *SSL – What is SSL?, SSL.com*

[3] *oAuth – oAuth, TechTarget.com*

[4] *JSON – JSON, JSON.org*

[5] *SLAs – Service-Level Agreement, Wikipedia*

[6] *Caching _ Cache (Computing, Wikipedia*

[7] *Throttle – Throttling Process (Computing), Wikipedia*

[8] *Cross functional team – Want to Develop Great Microservices? Reorganize Your Team, TechBeacon*

[9] *Agile approaches – Agile In a Nutshell*

[10] *DevOps – The AgileAdmin, What is DevOps*

[11] *Gartner – Automation: The Next Frontier for IT, May 2016*

Chapter Six:

Microservices Cost/Benefit

We are focusing on how microservices can better enable legacy applications to be assets -- not speed bumps -- in your digital journey. As such, microservices can benefit both the IT and business organizations.

Despite the benefits of a microservices strategy, it's important to be sober-minded about pursuing it, because adopting a microservices architecture is similar to the adoption of any relatively new software discipline. If you are building and deploying microservices, you will need the appropriate environment and staff. By no means is this an extensive or surprising list, but, here are a few things to consider when considering microservices, similar to any new technology or methodology.

Development of Microservices Architecture

Determining the cost & value of microservices projects isn't wildly different from other projects, but there are unique factors to consider. Here, for example, are just a few of the "getting started" expenses that might be incurred:

1. **Personnel Costs:** Not all developers will be familiar with the microservices architecture.

2. **Organizational Expenses:** Microservices architecture performs best when administered by small, cross-functional teams.
3. **Tools:** Containerization and other supporting technologies.

Rule of Thumb – Depending on where you are starting from, the "Getting Started" costs might be budget concern, but the downstream benefits will be significant. Adoption of a microservices architecture will quickly defray those costs by returning large amounts of business and technical value.

Maintainability & Ongoing Operational Costs

The first part of value generation comes in the form of maintenance advantages. Let's assume that you're starting out by running application monoliths, as opposed to green-field development. Maintaining these applications takes time, because they are built out of interlocking dependencies.

For example, imagine that there's an outage in a monolithic application – the login manager fails. Every other part of the application hangs on the login manager, so when it is down, it's all down. It's difficult to support a growing number of customers with an application that behaves like this, and while there are workarounds, such as failover services and instancing, they tend to be expensive.

> The reductions in maintenance expenses alone should be enough to pay for the up-front cost of microservices within a few years.

The time it takes to test, update, and maintain application monoliths means that maintenance has become a huge part of the traditional IT budget. A sample healthcare IT budget from Gartner shows that 70% of budget expenditures[1] come from simply running the business – increasing to 73% in 2017. This leaves little left over for innovation.

Rule of Thumb – A microservices architecture with fewer application dependencies and simple APIs, will immediately reduce the time and money spent on application maintenance. Application maintenance expense savings has proven to be more than enough to cover the "getting started" costs within a few years.

The Marriage of Quality and Speed

The dependencies (speed bumps) inherent in any monolithic application will inhibit innovation. Application monoliths don't tend to play well with newer development techniques – such as Agile and DevOps – that emphasize speed. Any update that's made to one part of the application will be reflected in other parts, so any update will need to be tested thoroughly.

There are automated testing tools designed to mitigate this problem, but like the solutions designed to mitigate failures in monoliths, they're expensive and hard to scale.

Microservices, on the other hand, let developers increase the speed of their development without sacrificing quality. This results in a competitive advantage – they will be able to refine their application faster than those who haven't yet adopted a microservices strategy. External customers and vendors will build up loyalty to these applications, while internal end users will become more productive.

QUALITY

Here's how it works: DevOps, Agile, and other modern development practices rely heavily on automated testing. The idea is to give developers or QA personnel the ability to set up several test environments in just a few clicks, and then let an automated testing program (e.g. Jenkins) handle most of the effort. Done correctly, microservices should require zero change to your legacy applications, thereby limiting the need for the time consuming and costly exercise of monolithic application testing.

Microservices make for a much cleaner testing process. They're built simpler, so it's easier to review their code. As a result, it's also simpler to perform unit tests. By definition, microservices are small and simple and quick and easy to write, therefore they are equally easy to test.

SPEED

The value of speed is different for every organization, but one can easily appreciate the benefit of a 90% increase in delivered services per year, or being able to push out 20 new services every five weeks.

Rule of Thumb: Speed of Development + Quality of Development = Competitive Advantage. For example:

- When an insurance organization leverages microservices to compete in the large insurance quote comparison engines, you're part of a fast-growing digital channel used by millions of shoppers.
- A bank that can offer mobile bill-pay and mobile deposits as a result of microservices is now able to capture younger generations of new banking clients who can offer lifelong value and add millions in deposits.

Walk, Then Run

Find a partner that will work with you on a pressing and compelling business use case, where a microservice architecture can bring immediate value. Define the success criteria of a low-risk proof of concept that will give you the ability to envision "what is possible", and the data points to confidently assess the potential and cost/value benchmarks necessary for you to begin your digital journey.

[1] *Gartner IT Budget: Enterprise Comparison Tool, March 2017*

Chapter Seven:

Tips for Getting Started with Microservices

The primary thing to take into consideration is to avoid diving into a “silver bullet” scenario - there is no one answer or approach for all. With any microservices strategy, there are countless options and alternatives. You need first to assess the business problems you are looking to solve, where you see your market going, how do your clients want to interact with you, what might be the operational issues, and what are the support issue impacts, and so on.

Based on this walkthrough, you might realize that your best approach to Digital Transformation might be evolutionary or revolutionary - either track will most likely require a combination of in-house resources/development, open source technology, service providers, and technology vendors.

It is inadvisable to begin a microservices project – or indeed any software development project – without first having a detailed plan as to how it should be accomplished. Microservices, however, might represent a greater departure from development norms than other forms of software architecture. This is because developing microservices doesn’t just mean learning new programing

frameworks – it means fundamentally reorganizing the functional units of development within an organization.

Let's cover two major microservice focus areas: The first is foundational – how to lay the groundwork for microservices within an organization. The second is procedural – the best ways that we have found for companies to create, build, and implement specific microservices from the ground up. With these blueprints in hand, companies will have a much bigger chance to bring their microservices approach off the drawing board and into reality.

Before You Get Started: Preconditions for Microservices

Research shows that many companies who attempt to replace entrenched legacy architectures with modern digital infrastructure will fail – usually after having spent a large amount of money over a long period of time. Microservices, on the other hand, are supposed to be the exact opposite – quick, cheap, and successful. To guarantee success, however, companies need to fulfill several preconditions:

First Precondition: Technological Enablement

One key challenge for organizations is the imperative to take the output from legacy software infrastructure built in the 1980s and translate it into an input that the latest model of mobile phone can understand. Microservices can help facilitate this process, but one key to this puzzle is knowing that making the legacy backend more

available to mobile and browser-based users will increase the workload on an already sensitive infrastructure.

Another question concerns the future. Microservices – and the technologies that support them – aren't static. Creating a microservices infrastructure means anticipating ongoing trends in information technology while avoiding mission creep. In other words, developers should create a microservices architecture that's upgradable, but one that also sidesteps the inevitability of a complex middleware stack.

The idea that all microservices should be designed with a common set of standards goes hand-in-hand with the idea that all microservices should be future-proof.

Second Precondition: Standards-Based Approach

The idea that all microservices should be designed with a common set of standards goes hand-in-hand with the idea that all microservices should be future-proof. Many third-party developers (which are commonly employed to create microservices architecture) don't value this approach, and as a result, they create frameworks which are blind to the realities of programming for legacy infrastructure. When no standards are applied to a

microservice, it will be difficult, if not impossible, to re-use and maintain them in additional applications.

Third Precondition: Gearing for Speed

It's easy for a development team to create a microservice quickly, but the fact that it's easy depends mostly on the team, not the microservice itself. A team that can create microservices quickly needs to be armed with the right technology and an appropriate set of standards, but that's not all.

Conway's Law says that organizations create software that aids the structure of their organization. Microservices are discrete and independently deployable, so an organization that creates microservices must be comprised of small teams that can work on independent projects with loose organization from the top down. Therefore, a microservices architecture complements an Agile or DevOps framework.

Agile, DevOps and Microservices

Approximately 66% of companies use Agile already as of 2016, but the usage of Agile is far from comprehensive on an industry-by-industry basis. Organizations that regularly grapple with legacy hardware and software will often have difficulty adapting the swift pace of Agile to the slow reality of developing for legacy systems.

- J.P. Morgan Chase – the world's 10th largest bank – only began Agile adoption as of 2015.

- Many healthcare organizations still consider Agile as an equivalent to Waterfall.
- Large government organizations still wrestle with the concept of adopting Agile methods to legacy hardware.

Many of the organizations that use Agile in theory may not actually be using Agile in actual practice. Agile is best defined simply as an organization's ability to achieve a high rate of change. If an organization still deploys releases only once every four months, then they're not actually Agile.

Adding microservices can help organizations achieve Agile in full. Microservices are designed to be small and flexible, so when Agile teams start working on microservices projects, their overall velocity increases. This can be enough to take a development organization from a monthly release cadence to a weekly release cadence – in other words, to transform an organization into one that achieves Agile in practice, as opposed to in name only.

Microservices/API software vendors that enable an organization to achieve scale and velocity with legacy-based applications can be considered a DevOps enabler. For example, OpenLegacy:

1) Automatically generate APIs/microservices from legacy applications in a fraction of the time, and with open standards

2) Generate standard Java objects that can be deployed as digital services for mobile, the Web, or in the cloud
3) Compatible with all standard DevOps tools and are easily incorporated into the DevOps ecosystem
4) Accommodate test-driven design by generating automated test units
5) Enable deployment by generating code that can be deployed like any other – nothing proprietary
6) Management console can be used to control both APIs and microservices, scale up or down, which aligns well with DevOps processes

Actualizing Microservices Once Preconditions are Met

Once an organization has the technological and structural prerequisites that can support a much faster development cycle, it's time to begin to create microservices in earnest. This is a three-phase process – discovering a need for a microservice, creating the microservice, and putting it into production. For example, at OpenLegacy, this can be summarized as Discover, Build, and Realize (Figure 1).

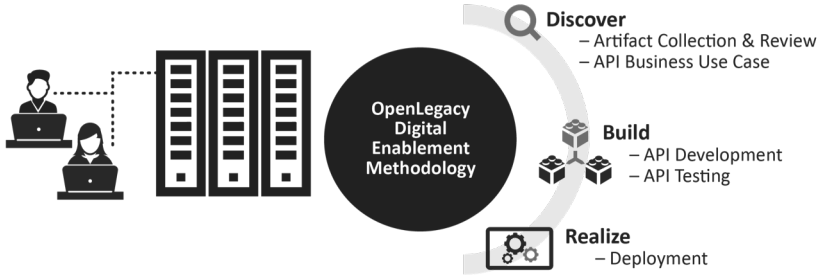

Figure 1. Once preconditions are met, one approach to getting started involves the Discover, Build and Realize approach.

Stage One: Discover

"Discover," in this case means documentation. Work doesn't begin until developers can articulate the detailed logic that underpins their microservice in a text document, describing what it's for, how it works, and whether there are any loopholes.

As one example, imagine a microservice that queries hospital records when patients and doctors want to look up their test results. The discovery document might include:

- Users: In this case, patients who wish to look up their own records, and doctors who wish to do the same.
- Description: This application lets users look up their hospital records. To do so, it must tie together a number of systems, such as a web portal, a record library, and/or a mainframe.
- Sequence: User authenticates via the web portal, navigates to the records page, and selects a record, which triggers the

microservices and provides a response back to the customer.

- Precondition: User must authenticate as a patient or a doctor, must be authorized to view records, and must have records to view.
- Postcondition: The application serves a medical record, which is downloaded as a PDF. If the microservice is unable to do so, it supplies error codes for the edification of the user.

The discovery document is the lynchpin of the microservice itself. Many errors, inconsistencies, and shortcomings of applications can be traced to their origins on paper, independent of their code. The bullet points above represent the briefest sketch of an actual discovery document – the real version should be thoroughly checked for loopholes.

Stage Two: Build

The "Build" phase of microservices development encompasses both writing and testing code. By their nature, microservices are mean to be created quickly and tested quickly, often using automated testing tools. Testing a microservice will generally be more complicated than writing it. Once the microservice is written, at least three phases of testing are required:

1. **Development and Build-Time Testing**
   Development testing is performed by a developer during

and immediately after the microservice is written. For microservices, these typically take the form of unit tests. Build time tests, on the other hand, will take the form of static analysis, with different test tools aimed at different parts of the microservice.

2. **QA Testing**

   QA testing for microservices involves positive/negative testing several areas. For example, there's functionality – does the application work as intended? There's also security and authorization – who can use the microservice, and how does it react to unauthorized users? Lastly, testing is performed on the business logic of the application to gauge application performance under a number of use-case scenarios.

3. **PTE Testing**

   Public Test Environment (PTE) Testing simulates the performance of a microservice in a production environment. Essential questions include how the application performs under normal traffic, whether it can perform well under unusual traffic loads, and how its performance translates into an SLA.

Stage 3: Realize

Once discovery and building have been completed, all that's left to do is put the microservice into production. This is very easy to do – possibly the easiest step in the discover-build-realize loop.

Microservices are designed to be easy to deploy by their very nature.

Because microservices are mini-applications, they can be designed and put into production without extensive collaboration between teams. Putting a new microservice into production doesn't run the risk of breaking another part of the application.

Therefore, the realization of a microservice is about more than just production. Rather, it asks developers to reflect on the lessons learned from building and testing an application. Incorporating these lessons is an aspect of continuous improvement. Creating your first microservice may have been an unexpectedly easy process. Incorporating the lessons from building it means that your second microservice will be even faster and easier still.

Put It All Together: Start Small & Think Big

For microservices evangelists within large organizations, the challenge isn't laying the groundwork, committing to institutional change, or programming new services. Rather, the challenge is getting buy-in from principals in order to get those changes started.

Microservices are designed to be easy to deploy by their very nature.

From the perspective of a single individual, this task can seem daunting, but there are a number of concrete steps that even just one person can take to move the needle.

Without proof – proof that the microservices concept is workable – there's no way that decision makers will commit to large-scale institutional change. Therefore, the evangelist's job is to create proof, ideally by committing to a successful small-scale microservices project. Fortunately, in a large organization with a lot of legacy infrastructure, there's ample opportunity for one person or a small team of people to pilot a workable microservice.

Imagine a specific function – something small, convenient, and preferably not mission-critical – that could be improved if it were augmented with a microservice. Take that function through the design, build, and realize process above. This will represent the test case for a microservices approach. If it works, that achievement will serve as a toehold: you can scale from there.

In most organizations, creating the culture that can support microservices – and then microservices themselves – is iterative. The best approach is to start with a small success and grow from there. Although the starting point is small, and the journey is long, the result will be a more streamlined and focused business unit that can finally meet the demands of an increasingly digitized economy.

One Last Thing: Choosing a Microservices Vendor

A monolith to microservices strategy is going to involve technology impacts, process impacts and corporate culture impacts - it is easy to state that one is Agile and flexible versus actually being Agile and flexible. So pick a vendor who will be a partner, one that has a proven track record of understanding where you are, what you have and where you want to go.

Vendor Technology

Not every microservices product is created equal, and some don't even fulfill the strict requirements of being a microservice. This is what Gartner calls "microservices washing" – the act of selling a product with superficial microservices labeling that doesn't conform to the definition of a microservice.

Up until fairly recently, SOAs and ESBs were the primary integration strategies for customer-facing applications. As this architecture lost some momentum, many vendors felt the need to pivot to a new architecture such as microservices.

In theory, this pivot is supposed to look like an idealized microservices strategy – a layer of microservices that exposes features to customers and partners, a layer that connects these features to business units, and a third layer that connects business units to legacy applications (Figure 1).

In practice, however, these companies never quite figured out how to get rid of the SOA architecture underlying their products. The result is usually something that looks like an ESB encapsulated by a microservices wrapper.

For example, imagine a microservice that exposes customer information as an API. The API needs to transmit all customer data, but only to some agents, due to compliance rules. This microservice contains an ESB. When it receives a request for customer data, the ESB contacts the mainframe, but not directly. First it needs to contact another microservice that will check whether or not the requesting agent is authorized to receive that data.

This violates good architectural design principles for microservices. It produces a multi-layer of dependency due to its integration, essentially creating a monolith. If one part breaks, the entire service goes down (Figure 2).

APIs need to connect to legacy mainframes in ways that are simple, intelligent, and modern. For example, OpenLegacy makes this possible by creating prebuilt connectors that interface with legacy applications in order to create a Java Object, with output that’s readable by a standard REST API, a browser-based web-page, or even an SOA web service (Figure 3).

The result is a tool that any developer can use without needing to learn or modify the legacy source code (e.g. COBOL). Instead of worrying about lengthy integration periods or legacy code, developers can create APIs and implement robust new features in just hours – more than fast enough to meet the increasing pace of customer demand.

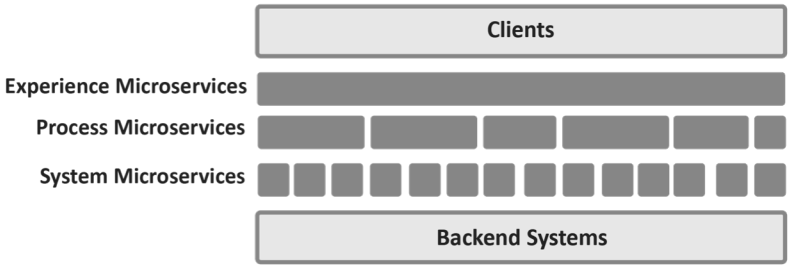

Figure 1. According to Gartner, organizations who over-state how their microservices work are engaged in so-called "microservices washing." The diagram above often represents how they say it works.

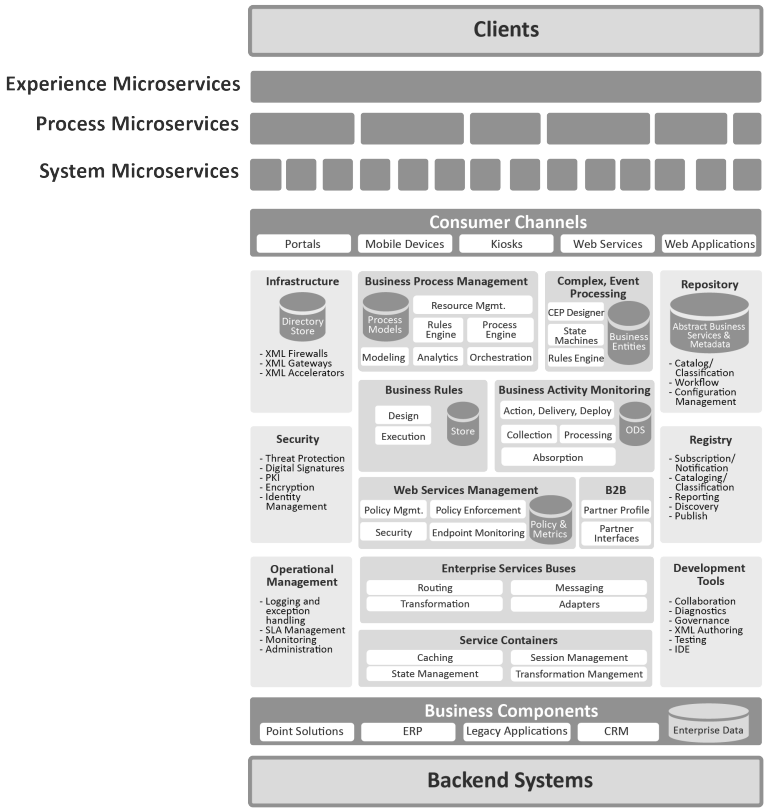

Figure 2. In a "microservices washing" scenario, the actual microservices architecture is far more complicated than described. This diagram is often how it really works.

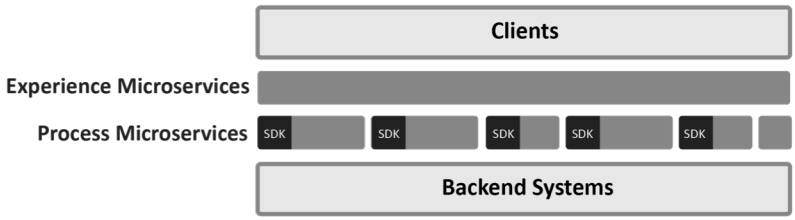

Figure 3. For example, OpenLegacy's "modern microservices" remove layers and complexity to accelerate innovation with legacy monoliths. This is how it actually works.

Vendor Checklist & Considerations

Partner or a Vendor	A partner should understand more than one aspect of your business challenges. A partner cares about what you care about. You will have issues, but a measure of a partner is how you resolve and walk through those issues together. A partner is transparent with their clients and quick to respond to questions. A partner will own their mistake if they make a mistake.
Subscription Model	It is all about flexibility, minimizing upfront risk and minimal capital investment. In a subscription model, subscribers are always up to date or can access the latest versions, are in line with the steady adoption towards cloud computing and software as a service, and can be time-boxed to accommodate current project timeframes and usage needs.
Support	Look for a defined support process and

Model	methodology. Understand the standard and premium terms around Response Time, Mitigation Time, and Resolution Time. Look into the enhancement request process, developer's community, knowledge center, user groups and primary point of contact philosophy.
Onboarding	How do you adopt this new technology, what is the training curriculum and can it be customized to your specific needs? What is available from a self-learning process, access and content within the Knowledge Center? Does your partner provide Quick Starts and On the Job Mentoring Services?
Open Standards	Why Open Standards? One could also ask why should you re-invent the wheel. Open Standards enable broader adoption with the community along with reducing the biggest barrier to adoption which is cost. Open Standards eliminate unnecessary barriers, implementation assumptions are reduced, and open standards increase adoption and cooperation amongst teams. Open Standards invites cooperation and thinking outside the box -- all pushing toward innovation.
Underlying Architecture	Does your partner practice what they preach -- are they using industry standard tools within their architecture? Are they flexible and adaptable to new or other technologies? Do they provide you with access to the people that can explain their underlying architecture and why they selected certain standards? Most importantly do they provide you with a clear and definitive Platform Roadmap?

Core Focus	Identify a partner who understands your business/technical problem. It might not be all your problems, but be sure they at least understand a specific business pain point, and are capable of solving that pain and do it very well. The ideal partner won't pretend to do more than what they can actually do, but are honest and eager to do the thing they do very well.

[1] *Gartner Blog Network, What A Microservice is Not, January, 2017*

Chapter Eight:

The Business Impact of Microservices & APIs

The benefits of microservices and APIs extend far beyond the IT department, and it's important to keep the business side of the organization more involved in the conversation.

The bottom line is that if you need to innovate faster yet your IT teams say that a project is "complex, expensive and time-consuming" because of your organization's "legacy systems," then microservices may be the answer.

The irony of legacy systems is that they tend to impact some of the oldest, most successful companies in the world. Legacy monoliths, like IBM, UNISYS, HP, Digital, Siemens, Honeywell, Tandem, Stratus and others began positively impacting business operations in the 1950's. Data storage and access methods, like VSAM, IDMS, IMS, DB2, Oracle, and ADABAS proved to be revolutionary in the ability to store and facilitate rapid access to data through mission and business-critical applications written in COBOL, Assembler, Fortran, ADS, RPG, and Natural.

Therefore, it is often the most established companies - the big brand names - that have the greatest difficulties remaining on the "cutting edge" within their industries unless they have found ways to accelerate innovation with their legacy monoliths. Some organizations have become so accustomed to the status quo, that they do not even think it is possible to reduce the backlog and speed delivery of innovative digital channels and applications.

When you have legacy systems *and* must cater to customers and prospects that are heavily reliant on digital services delivered via mobile or Web, you need to find a way to close that gap between old technology and modern demands.

When you have legacy systems *and* must cater to customers and prospects that are heavily reliant on digital services delivered via mobile or Web, you need to find a way to close that gap between old technology and modern demands.

We don't intend to sound too promotional, but we can speak most confidently about the business use-cases we have personally been involved in over the years in our roles at OpenLegacy.

One of the most amazing aspects of our work is that even the largest organizations in the world haven't been able to solve their legacy system challenges using in-house development, middleware technology, or SOA/ESB initiatives. All of these investments, while excellent choices at the time, are often not able to keep up with the demands of the new "digital economy" -- even after throwing hundreds of people, millions of dollars and years into the effort.

The business side of the organization may not care one bit about how the technology works: they just want to be competitive, nimble and build new or enhanced revenue channels. The IT side of the organization cares about reducing the complexity of accessing and leveraging their systems. Somewhere in the intersection of these two goals is where "modern microservices" become part of the conversation.

Very often, our approach to microservices reduces microservice creation from weeks to minutes or hours, and project deployment from months to weeks.

Although microservices can benefit any business with legacy systems, two industries that have an abundance of legacy technology and digital-demanding consumers are insurance and banking. So, let's start there.

Optimizing Agent Portal Efficiency

A major publicly traded insurance company developed an "agent portal" which exposed business processes for its workers. The organization wished to add more services to the portal, but the portal was based on an IBM i application. This meant that adding more services took months at a time— and in the meantime, the company's competitors were gaining.[3]

They were able to use an API connector which automatically scans and parses green screens in order to take the output from their IBM i application and place it into the context of their portal. No COBOL modification or re-write was required, and the first new service was in place within hours. As a result, their agents were able to rapidly understand the new interface and perform job responsibilities more productively.

Online Insurance Quote Comparison Services

Insurance aggregators have become a critical way for companies to get their price quotes in front of consumers. According to Price Waterhouse Cooper (PWC), 71% of insurance customers use digital research before buying a policy, and 68% of insurance customers were willing to download and use an application from their insurance provider. Unfortunately, one company found that their AS/400-powered application was unable to serve quotes to aggregation services. Even after six months of development, it still

took up to three seconds to deliver quotes – long enough that most aggregator services still refused to accept their input.[4]

This insurance company was able to use an API that defined web services on top of AS/400 transactions. This let them extend the reach of their legacy systems into the cloud. They were rapidly able to serve quotes to browser-based applications in just 300 milliseconds and are now included in all major insurance aggregators.

Improving Internal Staff Efficiencies

Before: Although an insurance company was able to modernize most of their offerings, their auto insurance offering was left as a legacy application. As a result, agents were forced to switch between a web browser and an antiquated "green screen," which impacted productivity and responsiveness.

After: This insurance company was able to expose a service from their AS/400 claim management system that presented all the reports related to a specific claim within the main auto insurance web applications. The initial proof-of-concept was completed in just five days. In production mode, insurance agents were able to realize time-savings of up to 30%.

“OpenLegacy let us connect our IBM i and AS/400 applications to our insurance agent portal without changing our COBOL applications, which would have been a huge, expensive headache. We couldn’t believe OpenLegacy was able to conform to all of our security, performance, and design constraints - and do so within days.” Insurance Services IT Director

“OpenLegacy let us connect our IBM i and AS/400 applications to our insurance agent portal without changing our COBOL applications, which would have been a huge, expensive headache. We couldn’t believe OpenLegacy was able to conform to all of our security, performance, and design constraints - and do so within days.”

Accelerating Bank Innovation By 50%

Before: Despite wanting to be an innovative, “FinTech-ready” bank, A major bank in Latin America was stifled by post-merger legacy systems spread across two countries and some of the highest operating costs of any bank in the region. Mainframe programming using COBOL was done in Columbia, Java programming was done Panama, and the infrastructure was maintained by a third-party global systems integrator. Excessive

complexity delayed time-to-market for new mainframe-based products and services, often requiring six months or more for deployment. Among their top priorities was a Payment Processing service for their commercial clients.

After: The bank's "digital journey" is based on OpenLegacy's microservice-enabled API integration and management software. The first project included training, creation, and deployment of 12 new APIs in just 8-9 weeks by only four Java developers. Total time for deployment of their new Payment Processing service was 90 days, 50% faster than typical mainframe projects. The bank considered these timeframes exceptional considering they also switched from IBM BlueMix to Amazon Lambda mid-way through then project. Furthermore, the service runs about 20-30% faster than those previously built without microservices, and a DevOps stress-test concluded that, together, OpenLegacy and Lambda far exceeded their goals by handling 60,000 concurrent requests.

Bank Creates Six Global APIs in Two Weeks

Before: A top ten global bank's demand for digital, global services led to a backlog of 100+ foundational APIs necessary to build new applications and customer experiences.

Nearly 200 developers have been working on the project for over a year, using a popular product for API gateway and orchestration. The product did not specialize in legacy (core) applications running on AS/400 and mainframe platforms, and despite its popularity, came short of addressing the bank's needs.

Specifically, the product could not generate APIs exposing RPG programs, but could only manage and expose existing APIs. Hence, the bank's developers had to write additional AS/400 code in RPG in order to expose functionality. In other cases, they had to change existing code, an invasive practice for applications that have been in place, tried and tested, for many years. Beyond just coding new features, there was substantial time invested in testing and regional certification and customization which slowed things down even further. Ironically, a tool that was supposed to shorten development time and reduce efforts ended up creating additional manual effort.

Another critical requirement for the success of the API project was creating a Global API: A unified API with the same end-user experience no matter the country, region, or underlying technical environment. From a business standpoint, this would require the extraction of common logic and functionality that can serve as a single launch point for new products and services that can be built and consistently rolled out in a unified, global manner.

After: OpenLegacy's API integration consultants worked out an alternative architecture, showing it was possible to eliminate all the intermediate layers and the AS/400 channel logic. Instead, the bank could directly expose the AS/400 transactions and move it to a Java application where the bank develops new applications and new logic. Now the bank can run and orchestrate transactions outside their legacy environment. This was made possible thanks to OpenLegacy's simplified product architecture with a minimal number of layers, and direct (straight-line) connectivity to the CBS transaction.

As a result, six key global APIs were created in just two weeks, and the new approach enabled the bank to accelerate "omnichannel" innovations for mobile Web or cloud to serve customers wherever, and whenever they choose. Furthermore, the simplicity and automation enabled the APIs to impressive 7X performance improvements.

"Within two days, OpenLegacy created APIs for standard CBS transactions, including payments and other financial products, compared to the previous 5-7 weeks using the prior API orchestration product and existing IT architecture." Executive, Top Ten Bank

> “Within two days, OpenLegacy created APIs for standard CBS transactions, including payments and other financial products, compared to the previous 5-7 weeks using the prior API orchestration product and existing IT architecture.”

These are a few examples of just a few industries, and there are many more (www.openlegacy.com/case-studies). As established organizations all over the world face the growing digital economy and the growing influence of the millennials and their digital impatience, there has never been a better time to consider microservices and APIs.

Regardless of the vendor and approach you choose, may *your* digital journey lead you to IT efficiency, faster cycles, greater scalability and competitive differentiation.

About OpenLegacy

OpenLegacy helps organizations accelerate their speed of innovation with legacy systems by quickly launching digital services for the web, mobile and cloud in days or weeks versus months. Our microservice-enabled API software accelerates innovation with legacy systems. We automate API creation, deployment, testing and management from core applications, mainframes and databases. We simplify projects by avoiding complex architectures, and we improve both staff efficiency and digital service performance. Together, business and IT teams can quickly, easily and securely meet consumer, partner or employee demands for digital services without modernizing or replacing core systems, and without special programming skills or invasive changes to existing systems and architectures. Learn more at OpenLegacy at www.openlegacy.com