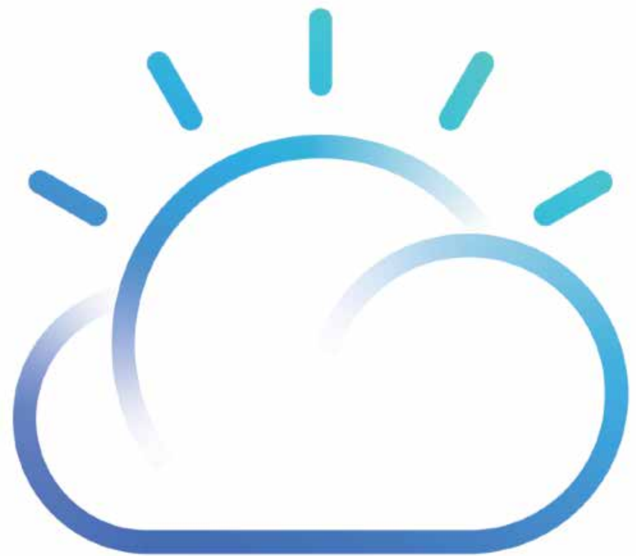
IBM **Cloud** IBM

Microservices point of view guide

Understanding microservices



Roland Barcia – IBM Distinguished Engineer, CTO Microservices, NYC IBM Cloud Garage
Kyle Brown – IBM Distinguished Engineer, CTO Cloud Architecture
Richard Osowski – IBM Senior Technical Staff Member, Microservices Adoption

Contents

Overview
3

Solve business problems with microservices
10

Migrate monolithic applications
10

Microservices architecture
12

Microservices capabilities
12

DevOps
13

Microservices compute options
13

Infrastructure services
14

Application architecture
15

Microservices stack
17

Application
17

Microservices fabric
17

DevOps
18

Container management
18

Resiliency in microservices-based architectures
19

High availability and disaster recovery patterns for microservices
19

Active/Passive
21

Active/Standby
21

Active/Active
21

Implementing microservices projects
22

Overview

In a microservices application architectural style, an application is composed of many discrete, network-connected components, termed *microservices*. Microservices architectural style is an evolution of SOA (Services Oriented Architecture) architectural style. Applications built using SOA services tend to become focused on technical integration issues and the level of the implemented services are often fine-grained technical application programming interfaces (APIs). In contrast, the microservices approach implements clear business capabilities through large-grained business APIs.

The biggest difference between the two approaches is how they are deployed. For many years, applications have been packaged in a monolithic fashion, a team of developers construct one large application that does everything required for a business need. Once built, the application is deployed multiple times across a farm of application servers. In the microservices architectural style, developers independently build and package several smaller applications that each implement parts of the application.

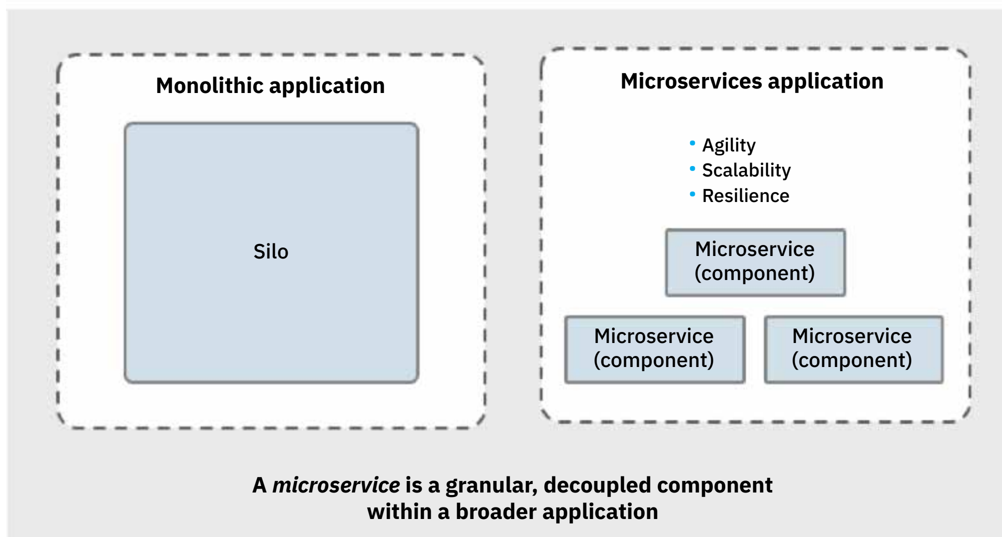


Figure 1: Monolithic application versus microservices

There are five simple rules that drive the implementation of applications built using the microservices architecture:

1. **Break large monoliths down into many small services** - A single network-accessible service is the smallest deployable unit for a microservices application. Each service runs its own process. This rule is called one service per container. Container refers to a Docker container or any other lightweight deployment mechanism such as a Cloud Foundry runtime

2. **Optimise services for a single function** – In a traditional monolithic SOA approach, a single application runtime performs multiple business functions. In a microservices approach, there is only one business function per service. This makes each service smaller and simpler to write and maintain. This is referred to as the Single Responsibility Principle (SRP)

3. **Communicate through REST APIs and message brokers** – One of the drawbacks of the SOA approach is that there are numerous standards and options for implementing SOA services. The microservices approach strictly limits the types of network connectivity that a service can implement to achieve maximum simplicity. Likewise, microservices tend to avoid the tight coupling introduced by implicit communication through a database. All communication from service to service must be through the service API or at least must use an explicit communication pattern such as the Claim Check Pattern [Hohpe and Woolf].

4. **Apply per-service CI/CD** – In a large application comprised of many services, different services evolve at different rates. Each service has a unique, continuous integration/delivery pipeline allows that proceeds at a natural pace. This isn't possible with the monolithic approach, where every aspect of the system is force-released at the speed of the slowest-moving part of the system

5. **Apply per-service high availability (HA)/clustering decisions** – When building large systems, clustering is not a one-size-fits-all approach. The monolithic approach of scaling all services in the monolith at the same level leads to overuse of some servers and underuse of others, or even worse, starvation of some services by others that monopolise all the available shared resources, such as thread pools. The reality is that in a large system, not all services need to scale; many services can be deployed in a minimum number of servers to conserve resources. Others require scaling up to very large numbers.

The combination of these five rules and their benefits are the primary reasons microservices architecture has become so popular.

The microservices architecture has become a de facto standard for developing large-scale commercial applications. In Microservices a definition of this new architectural term, Martin Fowler defines microservices:

“In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralised management of these services, which may be written in different programming languages and use different data storage technologies.”

One of the key differences between microservices and paradigms like SOA and APIs is the focus on deployed and running components. Microservices focuses on the granularity of deployed components rather than interfaces, as shown in the figure below.

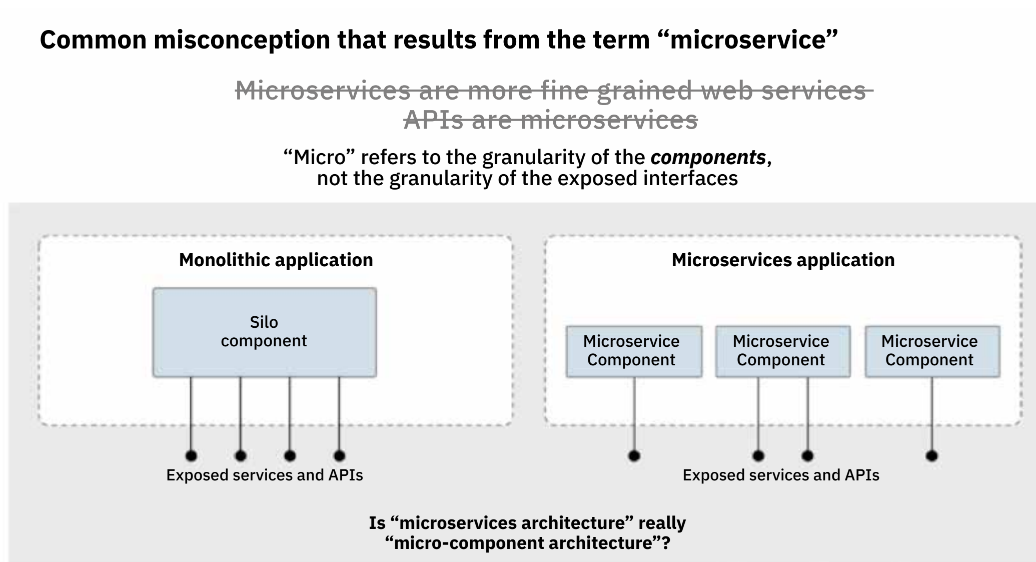


Figure 2: Difference between an API and a microservice

Many organisations, driven by operations, build monolithic applications where use cases and functions are deployed into a single, large application. While there is some ease around operations, changing these applications to be based on microservices requires a large effort. Three factors drive microservices development:

1. **How development teams are organised:** Microservices development is best done with an engineering approach that focuses on decomposing an application into single-function modules with well-defined interfaces, which are independently deployed and operated by small teams who own the entire lifecycle of the service. Microservices accelerate delivery by minimising communication and coordination between people while reducing the scope and risk of change

2. **How apps are built:** Microservice-based applications make some assumptions about the way they are built and the environment they run in. The environment is often referred to as *cloud native applications or 12 factor applications*. A microservices-based architecture leverages the strengths and accommodates the challenges of a standardised cloud environment, including concepts such as elastic scaling, immutable deployment, disposable instances and less predictable infrastructure

3. **How apps are delivered and run:** The use of containers as a standard way to run applications drives the packaging and running of microservices-based applications. Containers are not a new technology. Linux containers are an operating system level capability that makes it possible to run multiple isolated Linux systems (or containers) on one control Linux host. Linux containers serve as a lightweight alternative to full virtual machines. Even though containers are not a new concept, frameworks like Docker have popularised their usage by designing a way to create an image for running containers. This gives you a standard way to package an application and all its dependencies so it can be moved between environments and run without change. Other frameworks like Cloud Foundry use containers to run applications, but abstract the virtualisation away.

The figure below shows how monolithic application architecture evolves into a microservices-based one.

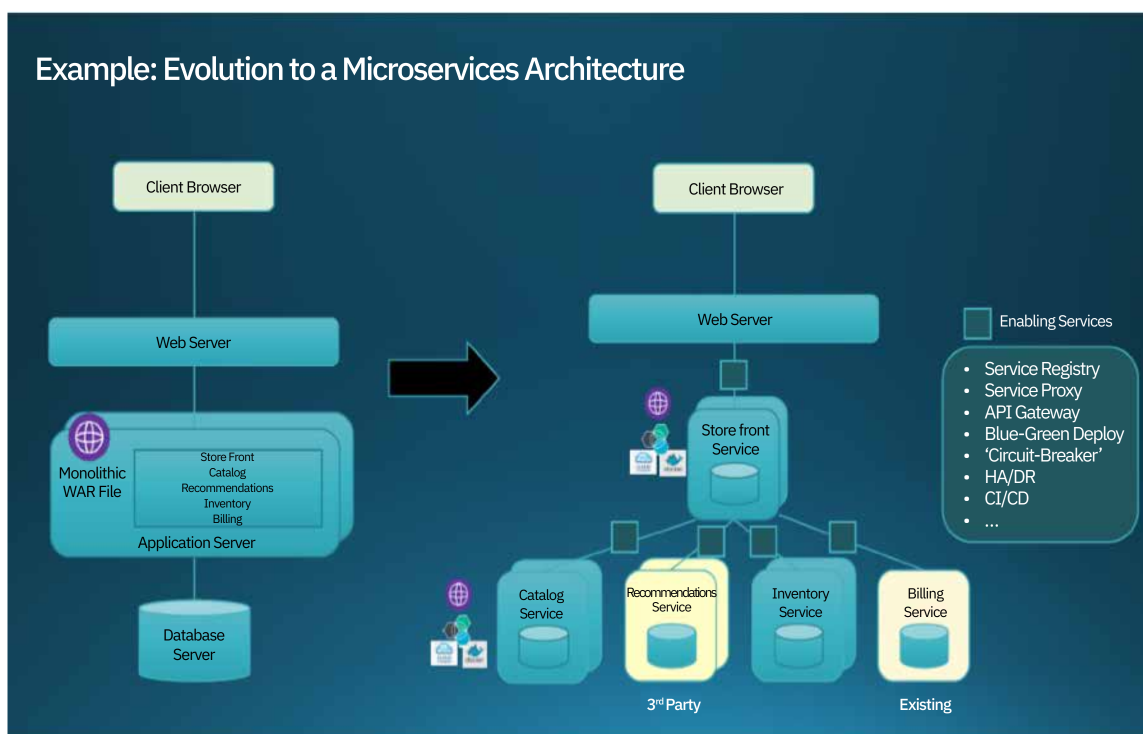


Figure 3: Difference between monolithic applications and microservices

Figure 3 shows a single enterprise archive that hosts all the components of a retail website. The application archive contains all the business functions, such as catalog and inventory, as well as website logic, business logic and persistence logic for each component. In addition, the application shares a single database, which often contains tightly coupled data models and is shared with other applications.

Notice on the right side of the image that the business capabilities are now deployed into separate applications with their own data. This new style of architecture introduces concern related to microservices communication, data ownership, data syncing and resiliency.

The Characteristics of a Microservice Architecture chapter by James Lewis and Martin Flower discusses key aspects of microservices-based applications.

Using the above reference example, a summary of the aspects includes:

Componentisation through services: This aspect is covered earlier in this paper.

Organised around business capabilities: We discussed the team development notion earlier and organising around business capabilities requires changes to how we think about development teams.

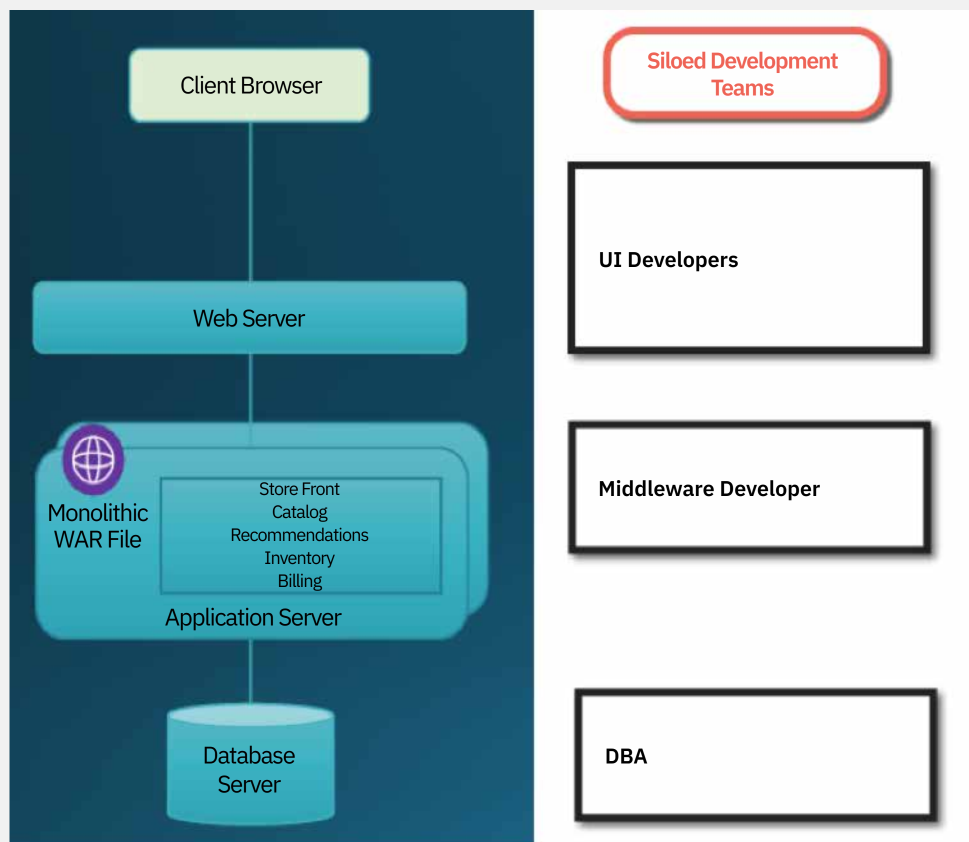


Figure 4: Siloed development teams

Overview

Consider how teams are often organised by roles as shown below:

Compare that to a microservices-based team approach, which organises around business capabilities, as figure 5 shows.

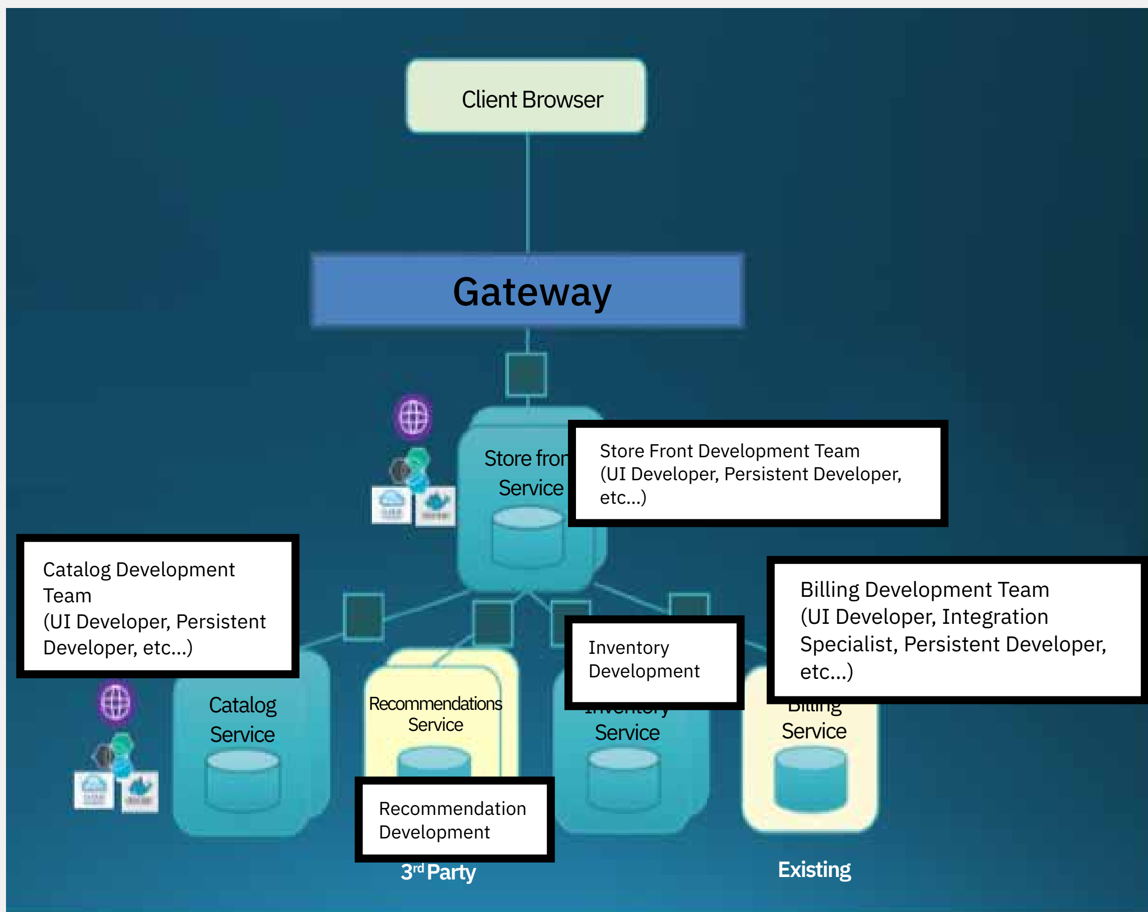


Figure 5: Business capabilities-based development teams

Solve business problems with microservices

Most companies build new, native cloud applications using a microservices-based approach. However, many companies are also seeing the need to refactor existing monolithic applications to move faster.

Migrate monolithic applications

Figure 6 shows an example of migrating from a monolithic architecture to a microservices-based architecture. In this case, a retailer wants to move towards microservices in order to move faster, learn more about the customer and introduce modern features like payments.

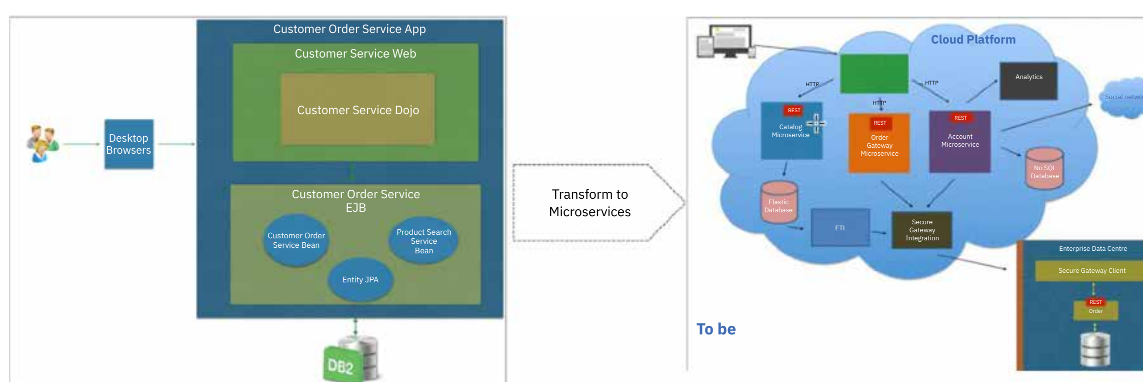


Figure 6. Migrating to a microservices-based architecture

The retailer in the above scenario followed these stages to move from a monolithic to a microservices architecture:

1. **Determine how to handle the catalog.** The first business problem they had to solve was how to manage the catalog of items. The customers couldn't find product data and the retailer couldn't expose things to other sites. The team decided to build a single microservice for the business catalog using the following steps:

    - Imported data into an elastic search to provide new ways to search data and identify new data patterns
    - Linked their existing website to the new search
    - With this pilot, they introduced a new CI/CD model as a foundation.

The team convinced management to move to a microservices model and expand the business.

2. **Learn more about the customer** - To learn more about the customer, the team created an account microservice:

    - First, they figured out how to change the business from a consumer-focused to an inventory-focused organisation. They designed a new customer model and used a NOSQL database like Mongo DB or Cloudant, which provided an unstructured data model. They knew that over time, the customer data would be enriched based on analytics, marketing and cognitive data
    - The existing customer data was used to track orders

3. **Create a new user experience** - The team created a new front end for mobile and web platforms and a new native mobile app. With a modern user interface and catalog, they created a new experience for the end user. The new catalog was integrated with the existing ordering logic, which comprised the core business and was too complex to break up just yet

4. **Order microservice** - The team focused on creating new order APIs for mobile, plus integrating with existing transactions. The business decided to create an adapter microservice that called into the existing system of record (SOR). They took the opportunity to integrate with new modern payments in this adapter layer

5. **Expand the business** - The retailer expanded the business model by adding a new auction feature, furthering the new microservice model.

Microservices architecture

Now learn about the details of a microservices-based architecture.

Microservices capabilities

Application components run as microservices in the cloud, communicating with each other through the microservice fabric. Different types of applications use different patterns. We show you a web/mobile example later. Figure 7 shows the capabilities needed for a microservices-based architecture.

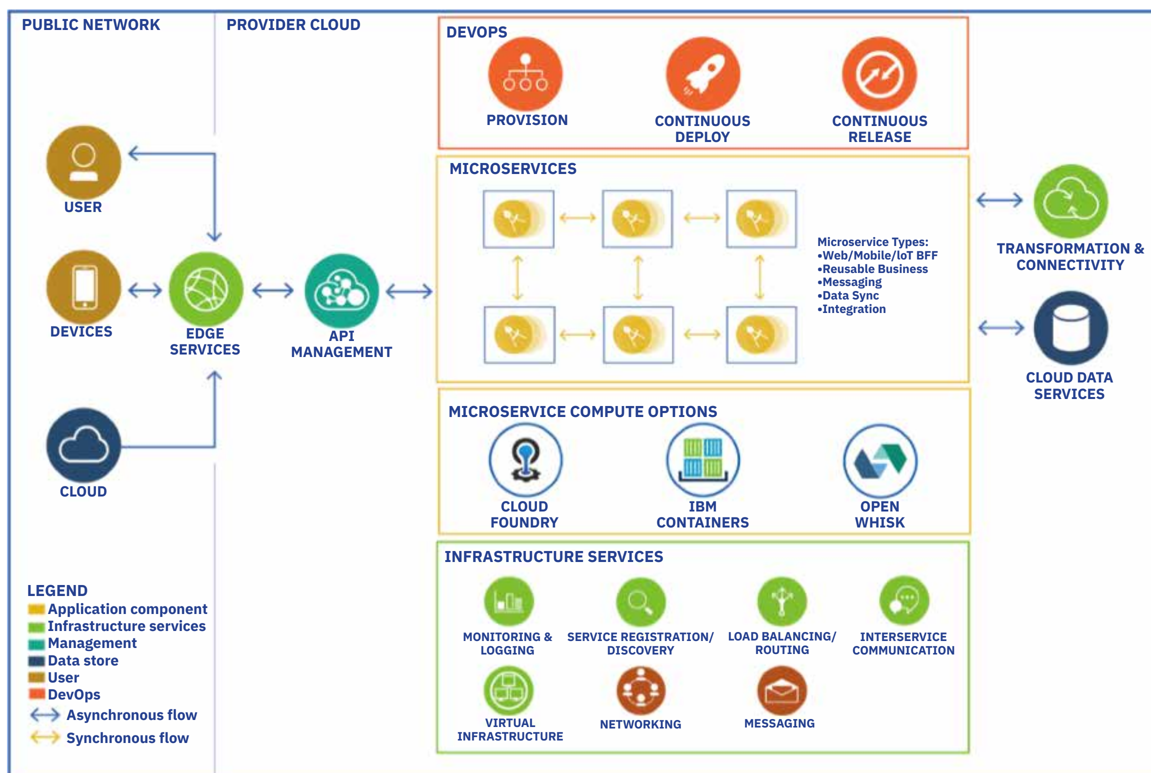


Figure 7. Capabilities of a microservices-based architecture

Reading from left to right on the diagram, the next step is to review microservices capabilities. Many microservices are exposed through APIs and some of these microservices need to be consumed through an *API Gateway*.

An API Gateway can be as simple as a proxy of endpoints. Some API Gateways are more sophisticated, with security at first contact, monitoring and API versioning, or full API management systems with a developer portal for third-party consumption.

DevOps

You must develop a strong automation strategy using DevOps to create a successful microservices architecture, which encompasses provisioning, continuous deployment and continuous release. Your strategy should include the following:

- **Provisioning** - A successful microservices architecture requires automated provisioning for application environments. The Platform as a Service (PaaS) layer usually provides this service through a managed services, such as Container as a Service (CaaS) in IBM® Cloud. If you run a container infrastructure over an Infrastructure as a Service (IaaS) layer yourself using Docker orchestration engines like Kubernetes or Docker Datacentre (DDC), you must automate how those environments are provisioned and updated using automated provisioning over a virtual machine environment

- **Continuous deployment** - An automated build and deployment process for Docker images or microservices applications is required in the development environment. If you have a multi-cloud strategy, your build and deploy automation needs to abstract the differences in how you do things like auto-scale or cloud policies

- **Continuous release** - A DevOps environment supports a strong culture of test-driven development and automated testing. This includes unit testing, automated functional and performance testing environment validation testing.

Microservices compute options

There are various compute options for running microservices:

- **Docker containers** - These containers provide the most portability across cloud and on-premises environments. Using Docker containers requires strong DevOps

- **Cloud Foundry** - This open source PaaS provides abstraction for how microservices run and communicate with each other and for virtualisation such as containers. While Cloud Foundry offers some level of portability, it requires a fuller stack in the cloud environment to run

- **Functions** - Event-based compute option with the highest level of abstraction and ease of use. Developers deploy simple event handlers to respond to events that cloud components emit into a central message hub. All virtualisation is abstracted

- **Virtual machines or bare metal** - You can create microservices-based applications and run them in virtual machines (VM). This requires a lot more provisioning and DevOps to succeed. VMs and bare metal offer the most flexibility at the expense of ease.

Figure 8 summarises these options:

Figure 8. Microservices compute options

Infrastructure services

The cloud environment that surrounds the microservices provides the necessary fabric and services for networking, messaging, microservice communication, logging and monitoring, virtualisation, service discovery and proxying, resiliency features and more.

Application architecture

Below is an example of a microservices application architecture.

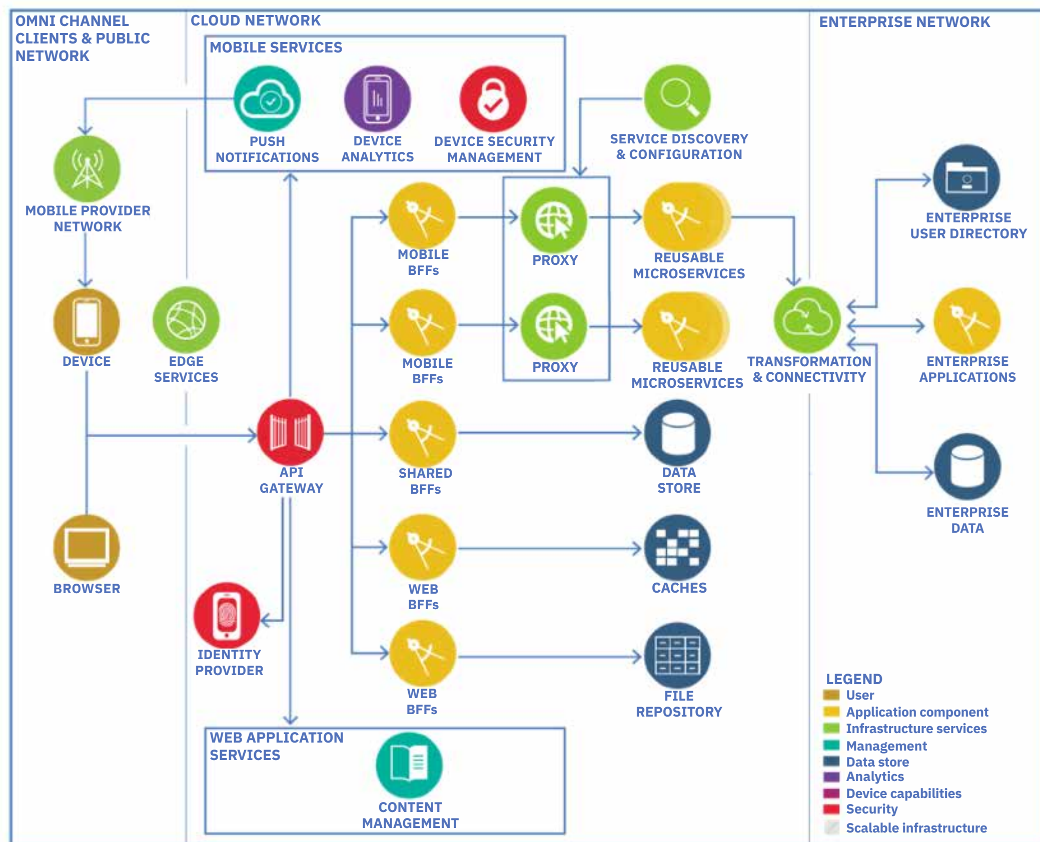


Figure 9. Microservices application architecture

To summarise components of this architecture:

- The OmniChannel applications in this example contain both a native iOS application and an Angular-based web application. The diagram depicts them as a device and a browser
- Mobile applications use the IBM Mobile Analytics for Cloud service to collect device analytics for operations and business

- Both client applications make API calls through an API Gateway. The API Gateway, IBM API Connect, provides an OAuth Provider to implement API security

- The APIs are implemented as Node.js microservices patterns that are referred to as Backend for Frontends (BFFs). Other possible names include Experience API (xAPI) or Iteration API. In this layer, front-end developers usually write back-end logic for their front-end. The Inventory BFF is implemented using the Express framework. The Social Review BFF is implemented using the API Connect LoopBack framework. These microservices run in IBM Cloud as Cloud Foundry applications

Read a customer case study   Get technical details

- The Node.JS BFFs invoke another layer of reusable Java microservices. In a real-world project, a different team will usually write this. These reusable microservices are written in Java™ using Spring Boot. They run inside IBM containers using Docker

- Node BFFs and Java microservices communicate to each other using a microservices fabric. Examples include Istio and Netflix OSS

- The Java microservices may interact with cloud databases and integration layers.

Microservices stack

It is important to define your microservices stack. Below is an example of a stack with some choices already made. You need to define the different layers, including the application, fabric and DevOps. Please refer to the IBM [Microservices Decision Guide] to see how IBM determined these microservices-enabling technologies.

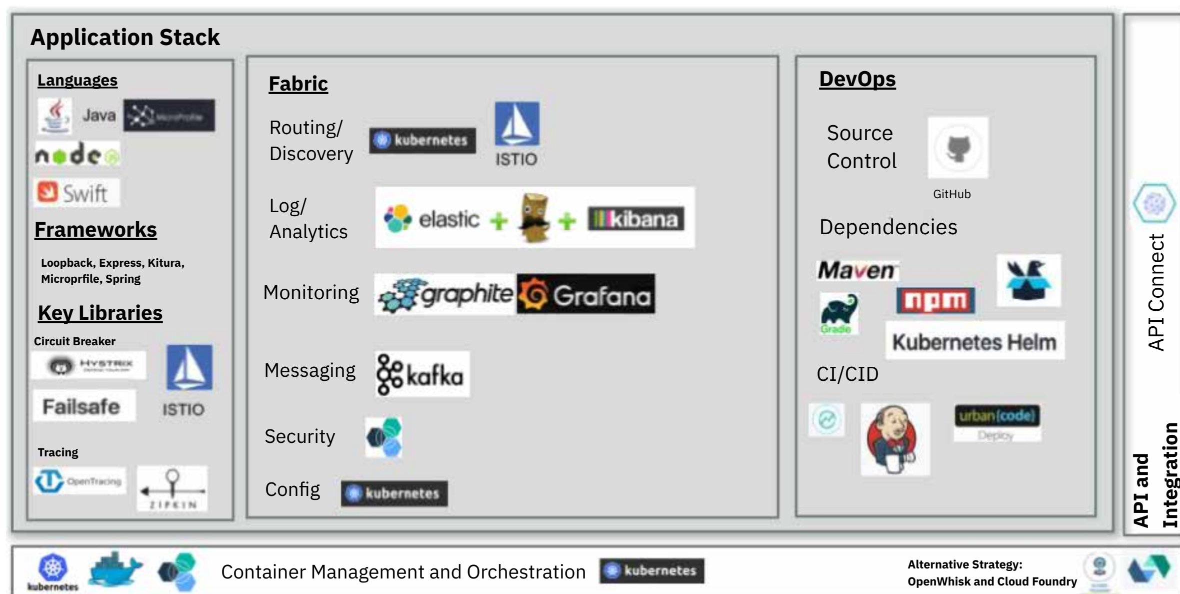


Figure 10. Microservices stack

Application

A language and a runtime stack write your application. Examples include modern, lightweight Java Platform, Enterprise Edition (Java EE) stacks like the MicroProfile, WebSphere Liberty, alternative Java stacks like Spring, or Node.js stacks like StrongLoop.

Specific language libraries are expected to handle microservices application elements such as circuit breaking and tracing.

Microservices fabric

A microservices fabric is a stack of capabilities that provides a set of necessary capabilities:

- Routing and discovery is a core capability for cloud-aware, inter-microservice communication. A microservice instance might be auto-scaled and have dynamic clusters for example and you must discover the microservice by name and then route to a running instance. Frameworks like Zuul/Eureka provide this as part of the Netflix OSS stack. Istio is a preferred option that is polyglot-based and provides finer-grained control of service call routing and control to the underlying platform. The figure below shows an example of the Istio architecture

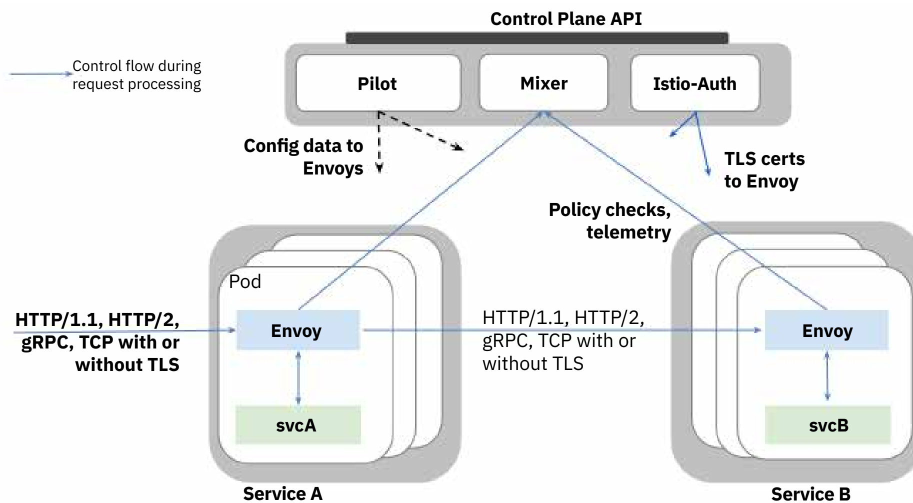


Figure 11. Istio architecture

- For logs and analytics, a stack that captures streaming logs and forwards them to various places is required. There are several frameworks available for that

- Monitoring dashboards for various data, including runtime, application and circuit breaker information is required. Using the log stack, these tools provide a unified view

- Messaging and security layers are important as well.

DevOps

You must have a strong set of DevOps tools to complete requirements for provisioning, orchestration, build and deploy and operations.

Container management

A container orchestration and runtime environment such as the IBM Cloud Container Service, Kubernetes or Docker Data Centre is necessary to support a microservices stack.

Resiliency in microservices-based architectures

With such fine-grained components, you must be aware of all resiliency points in a microservices architecture. This includes high availability, failover, DR, circuit breaking and isolation.

Figure 12 illustrates resiliency in microservices-based architectures and shows the use of a global load balancer and multi-site redundancy.

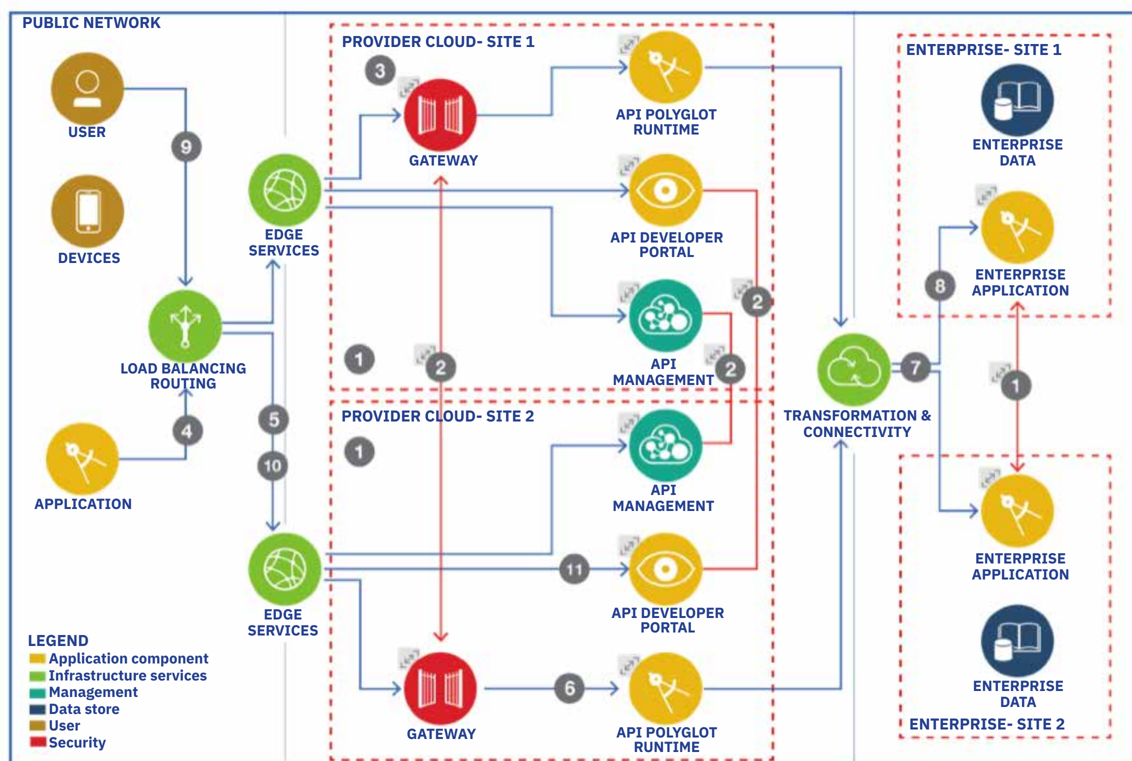


Figure 12. Resiliency in a microservices architecture

High availability and disaster recovery patterns for microservices

Resiliency considerations:

- Redundant data back ends
- Replicated data back ends
- Deploy multiple times per region
- Deploy to multiple regions (three data centres)
- Global load balancing.

When dealing with resilience, it important to make some distinctions between high availability (HA) and disaster recovery (DR).

HA ensures that services are available to the end users when maintenance activities like deploying updates, rebooting the hosting virtual machines, applying security patches to the hosting OS are performed on the system.

HA usually doesn’t refer to major unplanned issues like complete site loss due to major power outages, earthquakes, severe hardware failures or full-site connectivity loss. In such cases, if the services have strict service level objectives (SLO), you should make the whole application stack (infrastructure, services and application components) redundant by relying on at least two different IBM Cloud regions. This is typically defined as a Disaster Recovery (DR) architecture.

There are many options to implement DR solutions. For the sake of simplicity, the different options can be grouped into three major categories:

Active/Passive, Active/Stand by and **Active/Active.**

Active/Passive	Active/Standby	Active/Active
This option keeps the full application stack active in one location, while another application stack is deployed in a different location, but kept idle or shut down. In case of prolonged unavailability of the primary site, the application stack is activated in the backup site. Usually that requires restoring backups taken in the primary site. This approach is not recommended if losing data is a problem or the availability of the service is critical because recovery time objective (RTO) is less than a few hours.	With this option, the full application stack is active in both primary and backup locations; however, only the primary site serves users' transactions. The backup site stores a replica of the status of the main location through data replication like database replication or disk replication. In case of prolonged unavailability of the primary site, all client transactions are routed to the backup site. This approach provides good recovery point objective (RPO) and RTO (minutes), but it is significantly more expensive than Active/Passive because of the double deployment. There is waste of resources because the standby assets can't be used to improve scalability and throughput.	In this case, both locations are active and client transactions are distributed to both regions according to predefined policies like round-robin, load balancing and location. In case one site fails, the other site serves all the clients. It's possible to achieve RPO and RTO close to zero with this configuration. The drawback —both regions must be sized to handle the full load, even if they are used at half of their capabilities when both locations are available. In that case, the Auto-Scaling for IBM Cloud service allocates resources according to needs, similar to BlueCompute sample application.

Scalability and performance considerations

Adding resilience usually implies having redundant deployments, that can also be used to improve performance and scalability. That is true for the Active/Active case, described in the above section. In case of global applications, it is possible to redirect users' transactions to the closest location to improve response time and latency by using global routing solutions from Akamai or Dyn.

Implementing microservices projects

The next logical step is to understand how to implement microservice-based projects in your organisation.

You can either build a microservices system from scratch or from an existing monolithic system. Most IBM clients begin their microservices journey seeking to update their existing monoliths so this section focuses on how to assess and implement microservices when working from an existing monolithic architecture.

While building microservice projects without a prior application are relevant or important, use a monolith-first approach when building microservices. In short, this means you build your application in whatever way you can to validate your idea first.

Then, you apply the principles in this white paper to scale and evolve your initial monolith into a microservices project. There is no value in creating architecturally pure microservices that do not offer value back to the business.

An excellent first-hand account of implementing a microservices application using the monolith-first approach is documented by the Game On! team in The Chronicles.

Existing application evolution

By now you know there are many concepts you need to understand when undergoing a microservices-based transformation. In this section, we discuss three areas that you need to understand to implement a successful microservices project— :- your business, your culture and skill set and your technology.

Understand and define your business needs

Why are you are thinking about moving to microservices? For many businesses, more efficient software development and operation practices are required to deliver value to the business faster and deliver a better user experience.

Before you can understand the impact of a microservices project on your existing applications and infrastructure, you must understand which parts of your business are moving too slowly to produce satisfactory results. In many cases, the organisation’s systems of engagement (SOE) are causing the slow down. These systems are available through many channels, including web, mobile and APIs. Lack of speed is the primary reason to move to a microservices-based architecture.

Before you can adopt a microservices-oriented approach, you must
know what is not getting to market fast enough. First you need to identify which pieces of the application need improvements and modifications to make them faster. From there you can pinpoint which parts of the existing monolith should be targeted for microservice evolution.

Design and architecture artifacts, such as user experience flows or architecture diagrams are valuable to use at this step. The team can quickly identify and prioritise sections of the monolith, as shown in figure 13.

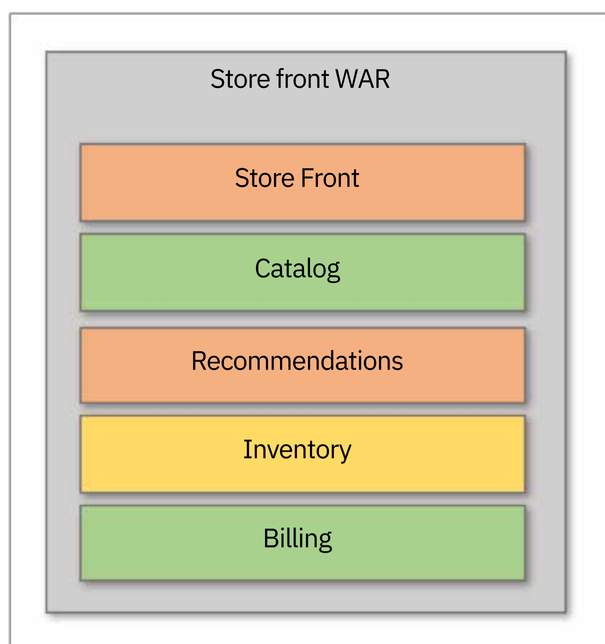


Figure 13. Existing monolith pain points, using Red-Yellow-Green scale for priority

This identification process does not need to be exhaustive and should be iterative in nature. The key goals are to:

- Identify the separate business functions your monolith is providing
- Understand the relative speed and complexity required to change those business functions
- Understand the desire of the business to see faster feedback cycles for specific business functions.

Understand your culture and skill set

While not specific to microservices-based architectures, a thorough understanding of an organisation's teams, culture and skill sets is critical during a digital transformation.

Typically, in engineering monoliths, most organisations are built into siloes, with participation as needed along the software development lifecycle. This often creates well-defined boundaries, with restrictive roles and responsibilities along those boundaries. Figure 14 shows a typical monolithic organisational structure.

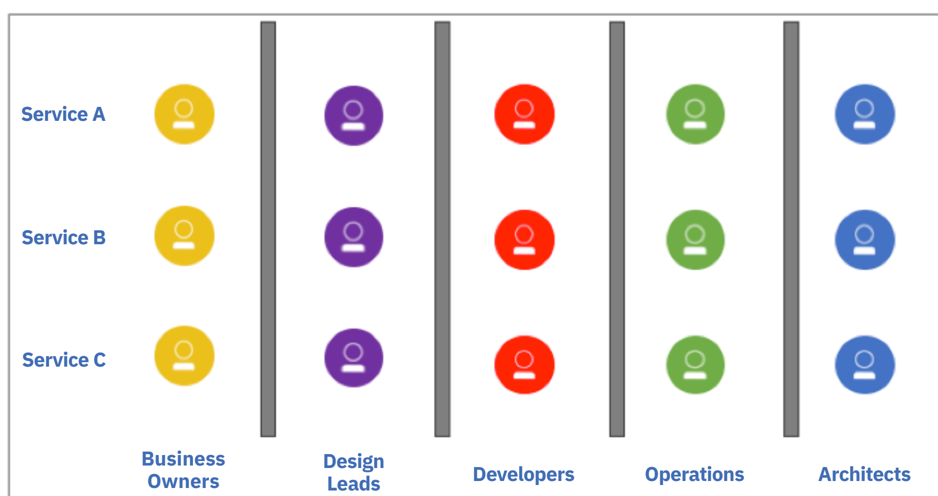


Figure 14. Traditional IT organisational structures

By comparison, Microservices architectures can only succeed when teams have the power to own the complete software development and operations lifecycle. To own the entire DevOps lifecyle, the teams need members with different roles and responsibilities.

These cross-functional teams are built to empower microservice-based architectures. Instead of siloed team members, all roles and responsibilities are contained in the same team. Everyone from design to development to operations works closely together and is often collocated. Physical team structures are outside the scope of this paper, but virtual team boundaries are most successful in transforming business when formed in a cross-functional manner.

This allows the business to clearly identify design experiences, understand possible delivery timelines and minimise operational expenses, since design, development and operations stakeholders are all represented on the team. Figure 15 shows an optimal DevOps team configuration.

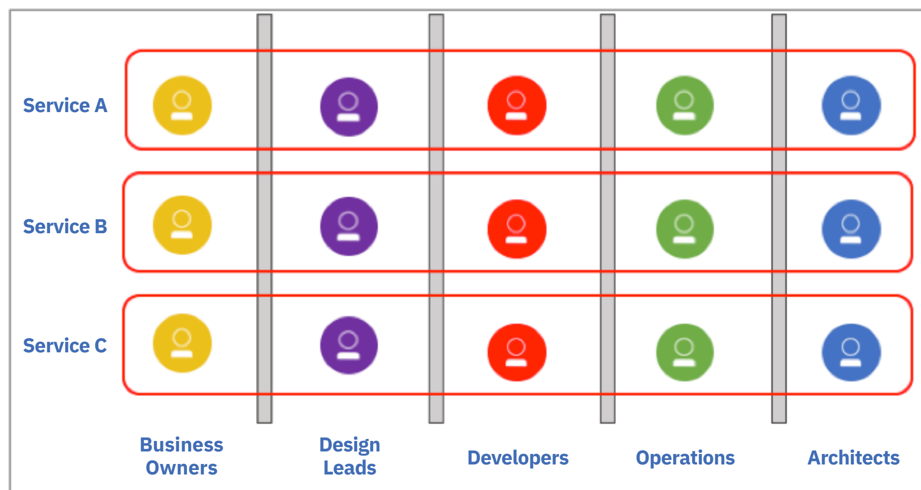


Figure 15. Optimised DevOps teams to deliver microservices success

A cross-functional team also supports the rapid growth of individual skills across the team. When a team owns everything the microservice is responsible for, from design to operations to runtime data, single team members are not relegated to single tasks. Often, front-end engineers develop database administration skills, while operations-oriented team members learn more about differences in user-interface frameworks. Expanding skill sets this way helps the entire IT organisation succeed with microservices, it is much easier to build new teams comprised of well-rounded team members versus looking for specialists to fill very specific roles.

Unless you address the business problem and your team's culture and skill sets, you won't be able to effectively implement the microservices technology and you will keep the same processes and structures in place.

Understand the technology

Proper analysis of existing technology stacks vary widely from organisation to organisation, but the simplified approach we describe helps ensure both the initial and sustained success of your microservices projects. Starting small and defining iterative, progressive successes is a much more achievable and fruitful approach than a transform-everything-at-once approach.

Figure 16. Iterative approach to microservices evolutions

The first phase of understanding your technology is to identify the coarse-grained services that are in the existing monolith. You can often align this with domain-driven design (DDD) principles that allow you to apply well-defined design principles to existing applications that were never built with them in mind. Identifying these coursegrained services helps you understand the complexity of the data structures, the level of coupling between current components and the teams who are responsible for new coarse-grained services. A successful review gives you a clear understanding of data boundaries, both inside of a given service and across services.

Once you identify the coarse-grained services, you must create a plan for how to evolve them into fine-grained microservices. These microservices, based on your previous work, should all work with similar data, manage their own data and understand what data they need to read and write to other services. From here, you can identify and implement resiliency, scalability and agility of the individual fine-grained microservices.

As previously mentioned, APIs and microservices are not a one-to-one comparison, but they are simply two parts of the larger whole. Once you have a better understanding of your fine-grained microservices, you will also have a better understanding of your interfaces, including which interfaces are on the critical path, which interfaces are optional and which interfaces are no longer needed. If you cannot map an existing interface or API to one of your coarsegrained or fine-grained microservices, it is highly likely that it is not needed.

Size the microservices effort

With all the analysis and planning work complete, it's time to define timelines, delivery velocities and expected results. These will vary widely but will not be an entirely new process from existing to digital transformation projects.

The heavy lifting of understanding the business, understanding the team structure and understanding the technology will now help ensure the organisation is prepared to understand the entirety of the microservices evolution of any given monolith, whether it is in a proof-of-concept scope, pilot scope, or large-scale evolution scope.

Microservices patterns

Development patterns for microservices

In development, patterns are useful for applying known solutions to common types of requirements and this applies to microservices as well. One of Martin Fowler's microservices design principles is that microservices are 'Organised around business capabilities.'[2] That stems directly from the discovery that just because you can distribute something doesn't mean you should.

These patterns are commonly used for microservices:

- The **Façade pattern** defines a specific external API for a system or subsystem. The subtext of this pattern is that this API is business driven
- **Entity and Aggregate patterns** are useful for identifying specific business concepts that map directly into microservices, for development teams who are not used to designing in terms of business interfaces
- **Services pattern** offers a way to map operations that do not correspond to a single entity or aggregate into an entity-based approach that's needed for microservices
- **Adapter Microservices pattern** is useful in the corporate world, where in many cases development teams do not have decentralised control over pattern data. In an Adapter Microservice, you adapt between two different APIs. One API is a business-oriented API built using RESTful or lightweight messaging techniques, with the same domain-driven techniques as a traditional microservice. The second API is a legacy API or traditional WS-* based SOAP service
- **Strangler Application pattern** addresses the fact that businesses and applications never actually live in a green-field environment. The programs that can benefit from microservices the most are big monolithic applications that can be refactored. The pattern provides an approach for managing refactoring. The Strangler Application pattern is covered in detail later in this paper.

Operations patterns for microservices

While microservices makes it faster to change and deploy a single service, it also makes managing and maintaining a set of services a greater effort as compared to a corresponding monolithic application. These operations patterns for microservices that were originally developed for conventional application management apply to the operations side of DevOps:

- **Service Registry pattern**

makes it possible to change the implementation of the downstream microservices and gives you the choice of service location to vary in different stages of your DevOps pipeline. This is achieved by avoiding hard-coding specific microservice endpoints into your code. Without Service Registry, your application will quickly flounder as changes to code start propagating upward through a call chain of microservices.

- **Correlation ID and Log Aggregator patterns**

achieve better isolation, while at the same time making it possible to more easily debug microservices. The Correlation ID pattern allows trace propagation through a number of microservices written in a number of different languages. The Log Aggregator pattern complements Correlation ID by allowing the logs from a number of different microservices to be aggregated into a single, searchable repository. Together, these patterns allow for efficient and understandable debugging of microservices regardless of the number of services or depth of each call stack.

- **Circuit Breaker pattern**

helps avoid wasting time on handling downstream failures if you know that they are already occurring. To do this, you plant a *circuit breaker* section of code in upstream services calls that detect when a downstream service is malfunctioning and avoid trying to call it. The benefit of this approach is that each call fails fast. You can provide a better overall experience to your users and avoid mismanaging resources like threads and connection pools when you know that the downstream calls are destined to fail.

Focusing in on the Strangler Pattern

Fowler's Strangler Pattern is based on an analogy to a vine that strangles a tree it's wrapped around. The idea is that you use the structure of a web application built out of individual Uniform Resource Identifiers (URIs) that map functionally to different aspects of a business domain, to split up an application into different functional domains and replace them with a new microservices-based implementation, one domain at a time. These two aspects form separate applications living side-by-side in the same URI space. Over time, the newly refactored application strangles or replaces the original application until finally you can shut off the monolithic application.

The Strangler Pattern steps are transform, coexist and eliminate:

- **Transform** - Create a parallel new site, for example, in IBM Cloud or on your existing environment, but based on more modern approaches
- **Coexist** - Leave the existing site where it is for a time. Incrementally redirect from the existing site to the new site for newly implemented functionality
- **Eliminate** - Remove the old functionality from the existing site, or simply stop maintaining it, as traffic is redirected away from that portion of the site.

The great thing about applying this pattern is that it creates incremental value in a much faster timeframe than if you tried to do everything in one big migration. It also gives you an incremental approach for adopting microservices, if you find that the approach doesn't work in your environment, you have a simple way to change direction.

When does the Strangler application pattern work and when does it not work?

- **Web or API-based monolith -** Starting from an existing Web or API-based monolith is the first requirement for successful application of the pattern. The purpose of the strangler pattern is to give you a way to easily move back and forth between new and old functionality. If your application is a web application, then its URL structure gives you a framework for choosing how and which parts of the system are implemented. However, any application based on a fixed set of APIs, such as set of SOAP APIs you are transforming to REST or even a set of queues implemented in a messaging system, will allow you to apply the pattern. On the other hand, thick client applications or numerous native mobile applications are not well suited for this approach since they don't necessarily have a structure that allows you to pull the application apart easily

- **Standardised URL structure (true use of URLs) -** Even though web applications all work according to standards imposed by the structure of the web, for example, HTTP and HTML, you can use a wide variety of application architectures to implement web applications. There is a lot of leeway within this approach that can complicate your attempt to pull the application apart. For instance, when there is an intermediate layer underneath the server requests, for example, a portal approach, you may have a problem using URLs to divide the application up. The decision for switching and routing isn't made at the browser level, but deeper in the application, which is more complicated

- **Meta UIs -** When the UI is business-process based, or constructed on the fly, it becomes difficult to separate code for the UI and business logic into different microservices. The approach still works, but the chunk size (see below) is larger and must be implemented all at once.

How not to apply the Strangler Application Pattern

The Strangler Application Pattern is not a cure-all. You don't want to apply it in every case or especially these :

- Don't apply it one page at a time. The smallest 'sliver' (see the release management section below) is a single microservice. That microservice needs to be complete and self-sufficient and more importantly, it needs to completely own the data that it manages. You want to avoid having two different data access methods for your data at once to avoid consistency problems

- Don't apply the pattern all at the same time. If you do, you're not really applying the Strangler Pattern and find yourself back in the Big Bang approach.

How best to apply the Strangler Application Pattern

So, if a single page is too small and an entire application is too large, what is the right level of granularity for applying the Strangler Application? To be successful, you have to interweave two different aspects of your application refactoring:

- Refactoring your back-end to the microservices design (the **inside part**)

- Refactoring your front-end to accommodate the microservices and to make any new functional changes that are driving the refactoring (the **outside part**).

Let's begin with the inside part:

1. Start by identifying the bounded contexts in your application design. A bounded context is another pattern from Eric Evans' Domain-Driven Design. According to the book, a bounded context, "defines the context within which a model applies."[3] A bounded context is a shared conceptual framework constraining the meaning of a number of entities within a larger set of business models. In an airline application, flight booking is a bounded context; whereas the airline loyalty program is a different bounded context. While they may share terms such as 'flight., the way in which those terms are used and defined are quite different

2. Choose the smallest, least costly bounded context to refactor. Rank your other bounded contexts in order of complexity from least complex to most complex. Start with the least complex bounded contexts to prove the value of your refactoring and resolve any problems in adopting the process, before you take on more complex and potentially costly refactoring tasks

3. Conceptually plan out the microservices within the context by applying the Entity, Aggregate and Service patterns from *Domain-Driven Design* as described above. At this point, you're just trying to get an understanding of which microservices likely exist so that you can use that approximation in the next set of steps.

Next, move on to the outside part:

1. Analyse the relationships between the screens in your existing User Interface (UI). In particular, look for large-scale flows that link several screens together tightly. If you are building an airline website, one flow may be booking a ticket, which is comprised of several related screens that provide information to complete the booking process. A different flow might be centreed on signing up for the airline's loyalty program. Understanding the set of flows helps you move on to the next refactoring step

2. As you either examine your existing UI or new UI, look for the aspects that correspond to the microservices identified in the inside part. Identify which flows correspond to which microservices. The output of this step is a list of flows one one side of a page and a list of the microservices that might implement each flow on the other side

3. Size your chunk based on the assumption that the UI changes must be self-consistent. Assume that one or more flows are the minimum size of change that can be released to a customer, but the chunk itself may be bigger than one flow. For instance, you may consider all of customer loyalty to be a single chunk, even though it may be made up of two or three separate flows

4. Choose whether to release an entire chunk at a time or a chunk as a series of slivers.

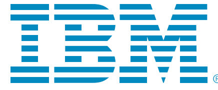
References

- IBM Cloud Garage Method, Microservices Architecture Centre
- End-to-End Reference Implementation (GitHub)
- Refactoring to Microservices
- Microservices Patterns IO
- Exploring Microservices: Game-On (GitBook)

[1, 2]"Martin Fowler, *Microservices a definition of this new architectural term,* *https://martinfowler.com/articles/microservices.html*

[3]Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software, 2003*

Fin.



IBM United Kingdom Limited
PO Box 41, North Harbour
Portsmouth, Hampshire PO6 3AU
United Kingdom

IBM Ireland Limited
Oldbrook House
24-32 Pembroke Road
Dublin 4

IBM Ireland registered in Ireland under company number 16226.

IBM, the IBM Cloud, WebSphere, IBM logo, and ibm.com are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

This document is current as of the initial date of publication and may be changed by IBM at any time. Not all offerings are available in every country in which IBM operates.

The client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions. It is the user’s responsibility to evaluate and verify the operation of any other products or programs with IBM products and programs. THE INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING WITHOUT ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND ANY WARRANTY OR CONDITION OF NON-INFRINGEMENT. IBM products are warranted according to the terms and conditions of the agreements under which they are provided.

© Copyright IBM Corporation 2018

Please Recycle

KUO12423-GBEN-00