# Microservices Decision Guides

February 20, 2017

**IBM Cloud Architecture Center**


## Authors

Roland Barcia

Kyle Brown

Gang Chen

Shahir Daya

Marcelo Martins

Richard Osowski

James Verwaayen

## Table of Contents

# Introduction

To successfully adopt microservices, you must understand and make deliberate decisions that impact how your organization approaches this new style of application development. These decision guides give you an overview of the key decisions you will make at both the business level and at the project level.

## Audience

These guides are intended primarily for application architects, in collaboration with a business analyst.

## Purpose

These documents offer prescriptive guidance to help you make enduring decisions around the successful development of a microservices-style application for your project. We discuss the following decision topics and offer practical guidance for how to achieve them.

- Adopting a microservices style instead of a monolithic approach
- Designing and securing microservices-based applications
- Successfully deploying microservices-based applications
- Ensuring ongoing resilient operation for microservices applications
- Adopting administrative and operational best practices

These documents are intended to answer frequently asked questions around microservices best practices and how IBM and its customers achieve success with their cloud application workloads.

### How to use these documents

You should read these documents along with the domain overview and use cases published in the IBM Cloud Architecture Center.

### Authorship

These documents were authored by a group of application development, cloud, and project management subject matter experts (SMEs) with review inputs from various development and field teams across IBM.

This version is an initial compilation and will be updated with new and timely guidance on the topic of microservices.

## Prerequisites

For a quick introduction to IBM Cloud deployment models and to learn about which cloud deployment models are best for your needs, view the doc: Choosing a cloud implementation and deployment model.

## Decision guides

When creating microservices, you must make the following decisions:

Business decisions guide: Evaluate your business readiness and make the key business-level decisions to adopt a microservices strategy over a monolithic approach

Architecture and design decisions guide: Create a microservices architecture development plan with essential foundational and design principles.

Implementation decisions guide: Learn what implementation considerations you need to think about for your microservices architecture, including runtime platforms, microservices frameworks, programming languages, and more.

Resiliency decisions guide: Create your microservices architecture with fault-tolerance in mind.

Operations decisions guide: Ensure your operations teams can monitor and manage your microservices ecosystem

# Business decisions guide

Before implementing a microservices-based application development strategy, you must first evaluate your business readiness and make the necessary business-level decisions to ensure the long-term success of your project. Compared to the traditional monolithic approach, a microservices strategy involves several different forms of business investment, including financial investments, an investment in the culture of your workplace, and an investment in new development and operations.

## Financial investments

Financially, creating a microservices environment will require costs related to the up front architecture and design effort that is required to build many discrete, loosely coupled, asynchronous services that are resilient, scalable, and easy to monitor and manage. Additionally, you must invest in unit testing, integration testing, and full deployment automation of these microservices.

These investments should result in a lower overall total cost of ownership (TCO) for those applications that need the qualities that microservices satisfy. However, as long-term testing and maintenance costs are reduced, there must be explicit business awareness of these costs and benefits, and a corresponding commitment to make the required financial investments.

### Is there a financial business case for adopting a microservices approach? Are you considering Total Cost of Ownership (TCO) over the full application lifecycle?

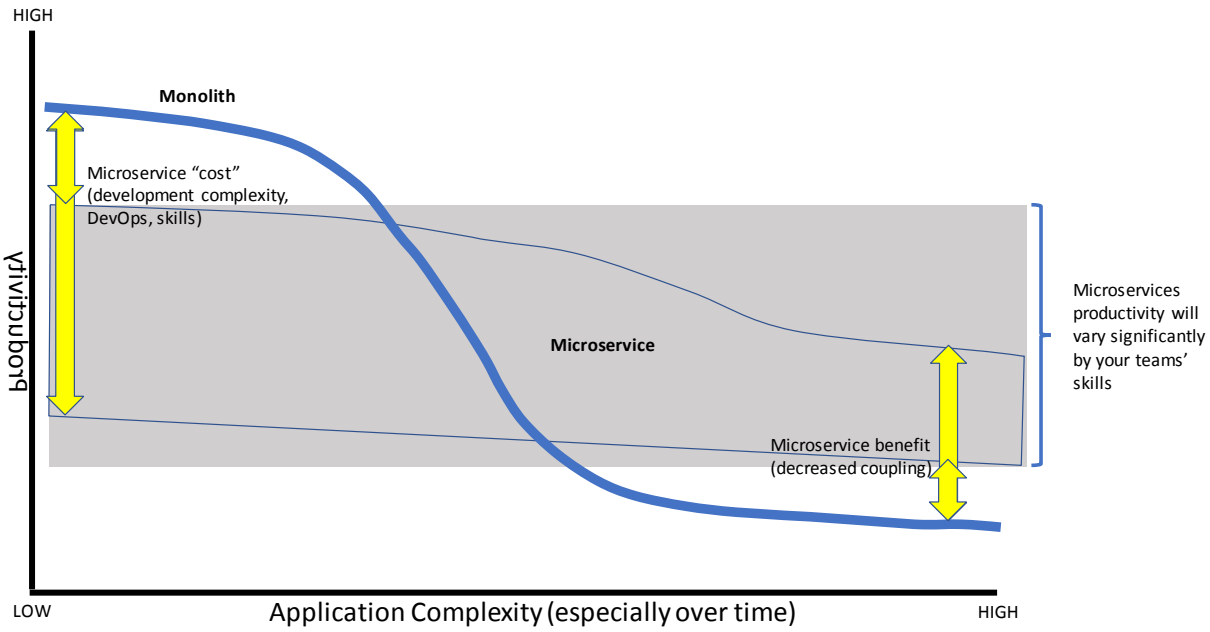
Monoliths and microservices are both valid application architecture styles, with each style presenting pros and cons. To proceed with a microservices style, you need to measure these benefits and costs in the context of your specific project or application.

If, for instance, you have a small application of low complexity that is developed and maintained by a small team and does not change frequently, then it might be hard to justify the cost imposed by microservices, and a monolithic style might suffice. By contrast, however, large and complex applications that are developed and maintained by a large (and likely churning) team, and which need to change frequently, become increasingly difficult to productively maintain when using a monolithic style. This is where microservices deliver their value.

As described at martinfowler.com:

> *‘‘The fulcrum of whether or not to use microservices is the complexity of the system you're contemplating. The microservices approach is all about handling a complex system, but in order to do so, the approach introduces its own set of complexities. When you use microservices you have to work on automated deployment, monitoring, dealing with failure, eventual consistency, and other factors that a distributed system introduces. There are well-known ways to cope with*

*all this, but it's extra effort, and nobody I know in software development seems to have acres of free time."*




Adapted from Fowler - https://martinfowler.com/bliki/MicroservicePremium.html

***Figure 1. Application complexity and productivity over time***

Your business case will be more readily embraced if you have existing monolithic applications that have already proven themselves to be too large and complex to manage. In this case, you're building a business case to solve a known problem.

However, for new or existing applications that don't yet have this problem but will face significant growth or increased mission criticality, your business case needs to demonstrate the anticipated complexities of improving resiliency, improving availability, increasing the frequency of deployment (including the impact of testing), managing the complexity of large development teams, and the complexities of achieving deployment automation, increased monitoring with a monolithic design.

Your business case, therefore, needs to consider these long-term productivity and agility dividends against the up front costs.

## What is the strategic importance, longevity, and expected growth of the application?

You must consider an application's strategic importance – including its expected longevity and expected growth – in your overall business case. The reason for this is that quantifying maintenance costs and rates of application change becomes increasingly imprecise over the long term. You should design applications that are strategic to your business operations or to your business reputation with agility in mind, and set aside a budget set for their continuous improvement and support.

## What are your marketplace pressures (competition and customer demands)?

When evaluating your overall business case for proceeding with microservices for a given application, you should consider the marketplace pressures that you are currently experiencing or anticipate.

Specifically, competition from existing or new customers may demand an increase in the speed with which they can deliver new features that are highly resilient and scalable.

And as your customers' expectations evolve, (influenced by both your competitors' services and by other unrelated services that have delivered exceptional experiences), expect an increased pace of new capability delivery and always-available service. Your business case should include the benefits that microservices offer for responding to the potential of these additional pressures.

Across all industries, demands of the digital business are changing the way that business applications are created and run. Applications need to be both highly resilient and highly scalable, while being extremely agile and thriving in environments of constant change. Microservices are becoming the dominant architecture style for how to develop these applications, while the microservices framework provides the tools and environment to deploy and run them at the required business levels.

## How large and complex is your application?

Large and complex applications tend to be good candidates for microservices development and refactoring. When identifying and ranking applications by complexity consider the following criteria:

- Look for a gap between how many deployments are performed every year versus how many are desired by the business and application teams. Complex applications typically bundle a large number of changes across a small number of change windows. This decreases agility, generally increases risk, and typically dramatically increases testing costs.
- Use function points, story points, maintenance costs, number of defects per release/per year as a determinant of application complexity.

- How large is the testing effort when a change is rolled out? Testing effort is typically directly correlated with complexity.

### Does your engineering team have microservices development skills?

Microservices involves the use of modern frameworks and languages. If your development team does not have these skills, you will need to invest in their training or acquire skills in the marketplace.

## Cultural investments

Adopting microservices entails an investment—or commitment—to a new culture of employee flexibility and empowerment. Traditional employee roles will not suffice in the new culture.

To succeed, you need a culture that gives development teams the ability to make decisions that were previously not in their control. For example, you will need to let the development teams – select the tools and technologies used by developers who are coding microservices, enable teams to customize how they do QA, and empower development teams to set their own delivery dates.

### Do you have the culture, skills, and tools in place to support continuous testing?

Microservices development is built on a continuous delivery and discipline. Continuous testing is a key component this discipline.

One advantage of microservices is that when you change one piece of code, your unit test scope is much smaller than with a traditional, monolithic system. Of course, that doesn't help much if you try to run all of your tests manually. Automation is the key for scaling the deployment process. You need to rely on one test framework that is capable of automatically performing unit, integration, non-functional (e.g. performance), and security tests.

If your development and QA teams do not have the culture or mindset, the skills, or the tooling to support continuous testing, you will need to make appropriate investments in training, methods adoption, methods communication, and tools.

## DevOps investments

Investment in new development and operations (DevOps) disciplines is also required. These disciplines are largely centered on continuous delivery and automation, which allows for quick iterations, low-risk experimentation, continuous feedback, and continuous improvement. For many companies, institutionalizing these disciplines requires a significant investment in employee training as well as acquiring skills from the marketplace.

## Do you have a culture, the skills and the necessary tooling in place to support continuous delivery?

In addition to continuous testing, you must have a continuous delivery and release pipeline to automatically test, build, and deploy the application on every change. With microservices, each microservice is an independent deployment unit. Relying on people to do this job manually is an impossible task. Automation is key here, and the only way to succeed down this evolution path is to automate everything from development through deployment, so that you can build and deploy frequently -- maybe even on a daily basis.

If your development and operations teams do not have the culture or mindset, the skills, or the tooling to support continuous delivery, you will need to make appropriate investments in training, methods adoption, methods communication, and tools.

## Do you have a culture, the skills, and the necessary tooling in place to support microservices monitoring, management, and continuous feedback?

The main difference between a monolithic application and a microservices-oriented one is that in the microservices architecture you have several independent parts that you cannot simply control by looking at a status indicator like an application server console. In addition, you can have several versions of services running. Understanding the behavioral difference between those versions is crucial to determining which will roll out for your customer base.

With microservices, data is never too much. The more data you have, the more you understand your application. When you are monitoring a microservices application, you need to collect more than the usual metrics (CPU, memory) from different points of the execution chain to best understand how the application is working.

After collecting the necessary data, the next important step is to display the data in a way that clearly identifies which bottlenecks exist and what failures are taking place at a specific moment in time. For that, you need a visualization tool that is highly customizable and capable of querying large chunks of data. Frequently, the same tool handles both data collection and visualization, but that is not a requirement.

You also need a collaborative relationship where operators can provide feedback to developers on a number of issues, including:

- New error-handling code
- Which error test cases need to be created based on observed production issues
- Logging and other data collection improvements to improve operational monitoring

If your operations teams do not have the culture or mindset, the skills, or the tooling to support continuous monitoring, then you will need to make appropriate investments in training, methods adoption, methods communication, and tools.

# Architecture and design decisions guide

As with any engineering activity, it is always beneficial to invest time up front to ensure you are using the most appropriate architectural approaches and making the necessary application design-specific decisions to help you achieve your business objectives.

Microservices, by their nature, introduce new approaches and considerations to architecture and design. Security, for instance, is increasing in importance, and must be designed up front, not as an afterthought. Likewise, the selection of the most appropriate application design patterns for the type of microservice function will help you achieve a balance between agility, operability, and the long-term maintainability and evolution of the application.

In this section, we list several foundational architecture and design considerations you will face with microservices development.

### What are the architectural patterns recommended for implementing an application based on a microservices architecture?

The following four patterns provide a basic framework for putting a microservices-based application together:

1. **Single Page Application (SPA)** - With the introduction of more powerful browsers, proliferation of faster networks, and robust client-side languages, many web interfaces shifted from distributed multi-page applications to single-page applications. A single-page application (SPA) is a web-based interface that provides the user with a single entry point to the application and never reloads the page or navigates away from that initial experience. Built using a combination of HTML, CSS, and JavaScript, these applications respond to user input through dynamic service calls to backing REST-based services that update portions of the screen instead of redirecting to an entirely new page. This application architecture often simplifies the front-end experience with the tradeoff of more responsibility on the backing services.

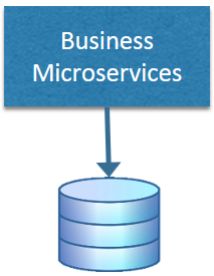
2. **Backend for Frontend (BFF)** – With the wide-spread adoption of single-page applications and REST-based APIs, there was soon a negative impact across user experiences through different channels. Requiring many backing services to populate an SPA would now delegate that responsibility to the browser to manage the many asynchronous REST-based services – often leading to very poor experiences across

devices. The common resolution to this issue was the implementation of a backend aggregator service that would then reduce the overall number of calls from the browser and in turn handle the bulk of the external backing service communication, returning a more easily managed single request to the browser.

This pattern evolved to be known as the Backend for Frontend pattern. The pattern allows front-end teams to deploy their own backend aggregator service (the BFF) that handles the entirety of external service calls needed for their specific user experience - often built for a specific browser, mobile platform, or IOT device. The same team builds both the user experience and the BFF, often in the same language, leading to a both an increase in overall application performance and application delivery.

3. **Business Microservice** – a business microservice performs one comprehensive business function. Deciding to implement each business entity as a microservice is not the end of your design problems, however. You must also think about how you would implement the microservice and how that microservice relates to the other services in your overall business application.

   Business microservices tend to be stateful, and tend to own their own data in a database that they manage.

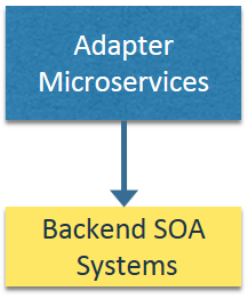



4. **Adapter Microservices** – an adapter microservice wraps and translates existing (usually function-based) services into an entity-based REST interface. This type of microservice treats each new entity interface as a microservice and builds, manages, and scales it independently.

   In many cases, it's a straightforward exercise to convert a function-based interface (for instance one built using SOAP) into a business-concept-based interface. In many ways this can be thought of as moving from a verb-based (functional) approach to a noun-based (entity) approach.

   Often, the functions exposed in a SOAP endpoint correspond one-to-one to CRUD (create, read, update, delete) operations on a single business object type, and therefore can map easily to a REST interface.

These operations would then simply send the corresponding SOAP messages to the existing SOAP endpoint and then translate the XML data types from the corresponding SOAP operations to JSON data types for the new REST interface.




The figure below shows what an application using these patterns looks like:

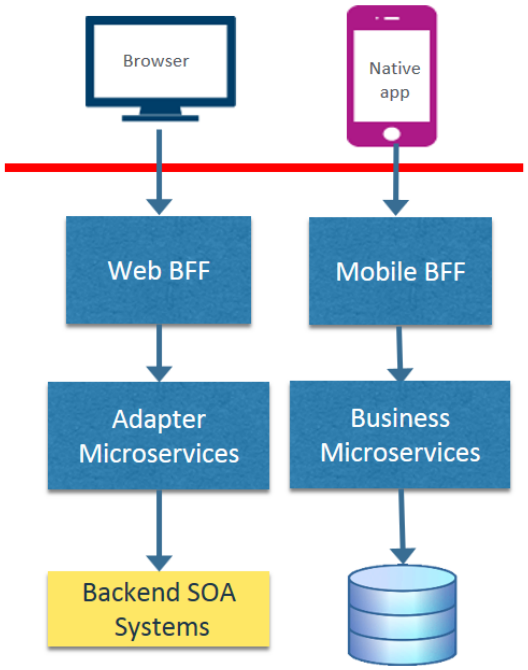



***Figure 2. Application built with microservices patterns***

**Does each business microservice have to own its own data and manage its own database? What if I have two microservices needing the same data?**

The self-contained nature of a microservice dictates that a service shall encompass all external IT resources (e.g., data sources, business rules) necessary to support the business activity. In addition, self-containment necessitates that service dependencies falling outside the scope of the development team should be minimized or, preferably, eliminated. Encapsulation is key. Related

logic and data should remain together, which means drawing strong boundaries between microservices.

When establishing a data model driven by a domain model, not the other way around, the distribution of data management responsibilities by each microservice naturally occurs. The boundaries established by a domain-driven design also imply a certain level of autonomy.

There will be scenarios when a shared database pattern needs to be employed, for example, when there's a need to join data that is owned by multiple services or a business transaction needs to query data that is owned by multiple services.

If that pattern is applied, you need to account for development and runtime contention, for example, if a long running transaction holds a lock on a table.

## How do you identify the right Microservice when you are developing a new greenfield application?

To identify the right microservice, you apply Domain Driven Design principles.

The book *Domain Driven Design* (Evans 2004) captured a "meta-process" for doing software design that Object-oriented Software Development Teams have used for a number of years. The book is unique in that it wasn't about specific design notations, or even specific classes of objects or patterns, but about the general categories of objects that good object-oriented designers identify and work with. The following fundamental object categories are important to identify in a services-design process. These become critical elements of a good microservices design.

**Entities**

An entity is an object that is primarily distinguished by its identity. Entities are the objects in the modeling process that have unique identifiers.

Evans informs us that these types of objects need to have well-defined object life cycles and a good definition of what the identity relationship is – what it means to be the same thing as another thing.

We know from entity-relationship modeling that sometimes entities are well-defined and have a specific well-known identifier, but may not ever live independently. Evans calls this combination of entities an *aggregate*.

In the cases where we have a cluster of entities that need to be maintained consistent in unison, we can refer to those entities as dependent entities, and we need to make sure we know what the root of the aggregate is, since the root defines the dependent entities' lifecycle.

**Value Objects**

Many other types of objects that you find in object modeling do not have a well-defined identity function. They have no conceptual identity. In those cases, you can't (or don't need to) tell one object from any other of its type. Evans refers to these types of objects as value objects.

Evans makes the point that if we start treating all objects in a system as entities, the complication of assessing and managing the identity of each object can become overwhelming and impact performance adversely. He asserts that you only need to care about the attributes of a value object, and that each value object can be treated as immutable.

**Services**
Evans further points out that in most domains, there are some operations that do not conceptually belong to any specific object.

Previous design methods often tried forcing these operations into an entity-based model, often with adverse consequences – this is especially true of operations that didn't operate on one entity, but on a group of related entities.

Instead, Evans suggests we model those objects as standalone interfaces called services. According to Evans, good services should follow these rules:

- They should be stateless
- The service's interface is defined in terms of other elements of your domain model (entities and value objects)
- The service refers to a domain concept that does not naturally correspond to any particular entity or value object.

One last point – these are operations that are part of the business domain – not technical issues of implementation. Tasks like login, authentication, and logging are not appropriate services of this type. On the other hand, a concept like "funds transfer" in the banking industry or "adjudication" in the insurance industry might be.

The outcome of domain driven design gives you:

- A list of entities, some of which are aggregates
- A list of value objects that are associated with one or more entities
- A list of services that correspond to functions that are not part of any particular entity

It turns out that this short list of object types is exactly what you need to be able to perform your first-pass RESTful service design for microservices.
Not all of these objects will become part of your Microservices design – some may be hidden within your service implementation, but this at least gives you a brief introduction to what you are dealing with.

## How do you identify the right microservices when you are refactoring an existing monolithic application towards a microservices-based architecture?

You can apply the same Domain Driven Design principles to identify microservices when refactoring an existing monolithic system. One of the initial goals is to identify bounded contexts. A bounded context is a conceptual boundary where a domain model is applicable. Dependency analysis tools such as JDepend may assist in finding natural system boundaries.

The next step is the decomposition of the monolithic system into components. The goal of this activity is to ensure that each service has a single responsibility. Services should be independently replaceable and upgradeable.

Some good practices to follow when in conducting this process are:

- Try to separate databases before separating services.
- Ensure existing transactions continue to reside within a single service or redesign to use compensation and/or eventual consistency.
- Consider what you want the microservices teams to look like.
- Build new features as microservices around an existing monolithic system.

It is also important to know when not to refactor a monolith into microservices. The main reason to not refactor a monolithic system is when the cost of managing the service outweighs the benefits. Dysfunctional communication patterns in the organization and a cultural climate that does not support team autonomy are other factors that make it hard to adopt microservices.

## What is the recommended approach to design the API exposed by a Microservice and what are the recommended design patterns?

When designing a microservice API, you should consider the following approaches:

**Enforce strong contracts**
A microservice must provide a versioned, well-defined contract to its clients, which are other microservices. Each service must not break these versioned contracts until it's known that no other microservice relies on a particular, versioned contract.

**Avoid chatty interfaces**
Chatty interfaces are interfaces that require you to perform multiple calls to accomplish a task. In a distributed system, this can have detrimental consequences to your service performance and availability.

**Message serialization**
There are many important factors to consider for the serialization format: Who are the users, how much data is being transferred in each request, can the data be compressed? JSON is currently the popular format for microservices APIs, which can be parsed directly into an object graph, but JSON is not a compact format. Performance requirements may lead an API designer to consider other formats.

**Blocking vs. non-blocking APIs**
One of the most important aspects of API design is whether to use blocking or non-blocking calls. Non-blocking APIs scale better, but are more complicated to design and use. Blocking APIs can have shortcomings addressed by some of the resilience patterns described later in this document.

The design patterns that inform the API design of microservices are as follows:

**API Gateway Pattern**

An API Gateway is used to abstract the communication between client applications and internal microservices. The API Gateway allows for the composition of microservices into client-ready services.

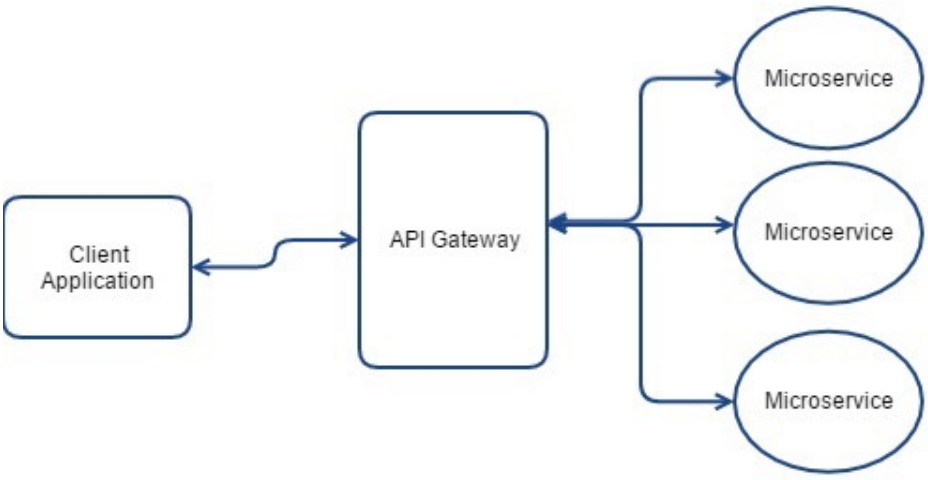



***Figure 3. API Gateway Pattern***

**Microservices Discovery Pattern**

The Microservices Discovery Pattern removes coupling between microservices and client apps. By dynamically registering microservices in an enterprise topology, we allow client applications and other services to dynamically discover microservices and adapt to changes. This pattern also avoids the centralized registry pattern of traditional SOAs.

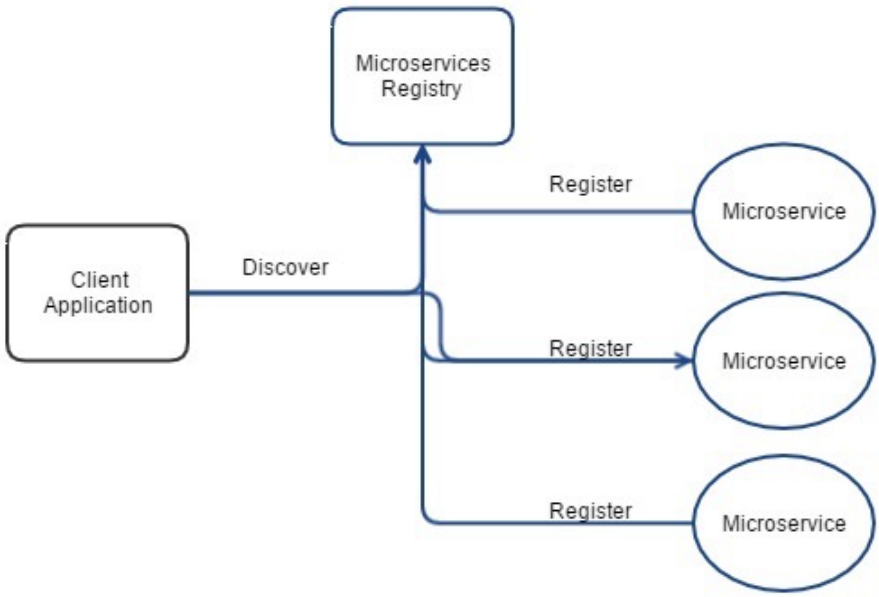



***Figure 4. Microservices Discovery Pattern***

**Microservices Description Pattern**

The Microservices Description Pattern expresses features of microservices in a descriptive format that can be understood by client applications. It also offers a means of managing microservices metadata.

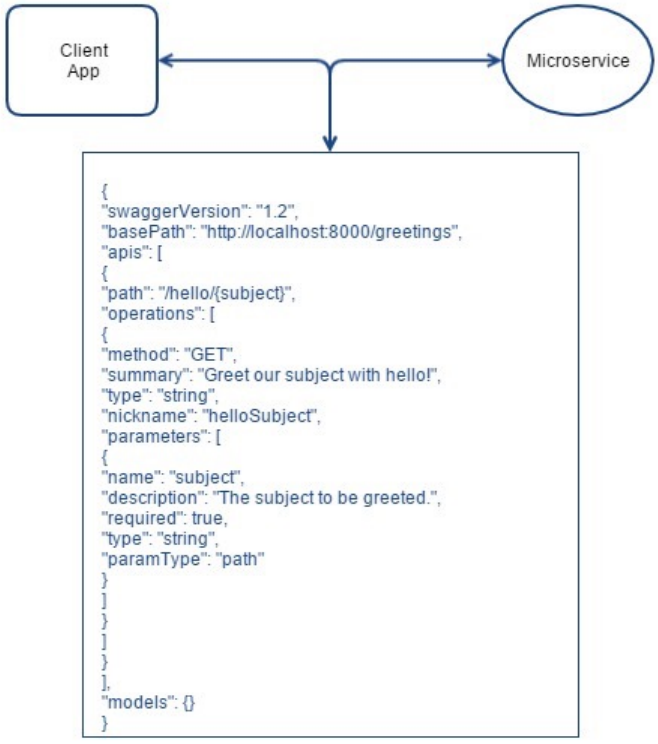



***Figure 5. Microservices Description Pattern***

## What is the recommended approach for versioning microservices?

There are two options for versioning the exposed API of a microservice. If you need to provide additional information on a GET or POST operation, then the change is unlikely to be backwards-compatible. In that case, you need to look at ways of handling this problem. The two most common ways of handling this are:

1. Versioning in the URI
2. Versioning in the header

The REST community is split nearly 50/50 on which is the best approach for this, so we present both.

**URI versioning:** Versioning in the URI is when you change the URI of the resource itself to contain version information. A simple example of this from our bank account example might look like: /accounts/v2.1/{id}

This approach gives you the ability to version an entire resource hierarchy or branch. It's also more semantically meaningful to developers—they can see at a glance which version of a service they are referring to. Modeling the version in this way as a resource enables automated navigation or discovery of resources. For this reason, we recommend it for most purposes.

A disadvantage of this approach is that when you include version information in the URI, you change the resource name and location. This can introduces a complex proliferation of URI aliases that make it difficult to identify which version of your API is the currently supported version. What's more, you can no longer use URIs to compare identity in this approach -- the same identical object may be returned by both the version 2.0 and 2.1 URI's.

Additionally, this may break existing hypermedia links that do not include version information.

You can get fancier with this approach, but this makes it troublesome as the examples show:

versioning at multiple hierarchy nodes - complicated
`/maps/version/2/roadways/version/2`

query parameter – not recommended
`/maps?version=2`

**Header versioning:** Another approach is to include version information in a special header of each request or response. An example of this header might be: X-Version:2.1

An advantage of this approach is that the resource name and location remains unchanged throughout your hierarchy, so you won't have a proliferation of URI aliases. This approach makes it easier for transparent intermediaries to parse the headers for routing in scenarios where you have an ESB in place between service requestors and service providers. Likewise, by keeping the URI the same across versions, the API remains completely semantically meaningful to developers.

A drawback to this type of versioning is that information can't be readily encoded into hypermedia links. What's more, this approach doesn't discriminate among multiple representations. Additionally, it only works with custom clients that know how to encode the special header, thus introducing coupling into your design.

**Impact to the backend**

URI service versioning is the best practice for updating the public API of a service. But putting URI versioning in place doesn't address any breaking changes to the backend data stores that may need to take place. There are two options for how to deal with this, and neither option is great:

1. Option 1: Copy your old data into a new "V2" database and keep the two entirely separate. This means that either you live with data drift or you put a data synchronization solution in place].
2. Option 2: Update your schema in place and add code to v1 (!) to handle the new schema.

The following image shows these two options:

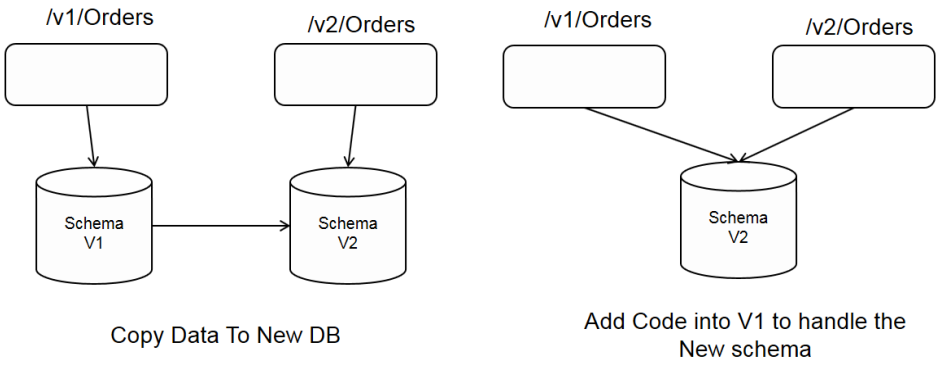



***Figure 6. Two ways to do URI Versioning***

**What are the security considerations when designing microservices?**

Security is a critically important, comprehensive, and continually evolving topic for any IT system, including those based on a microservices architecture.

First, ensure your organization adheres to secure engineering and management practices – implementing control frameworks such as those prescribed by NIST SP 800 or ISO 27001, which is only available via subscription. For instance, the NIST framework provides guidance on 285 security controls. If you need assistance implementing a framework, adopting secure engineering practices, or assessing your security posture and maturity, consider engaging consultancy services, such as those offered by IBM.

In addition to the above controls, you should consider the following internal and external considerations related to microservices security. Some are based on emerging standards and practices and will continue to evolve.

For a comprehensive list of security questions that cross both microservices and standard DevOps Service Management, refer to Graham Lea's article: Microservices security: All the questions you should be asking

**Internal considerations (that is, within your own data center or network):**

- **JSON Web Tokens** (JWT): JWTs are becoming the prevailing standard for representing claims between two parties. These open standard tokens are the foundation for both the OAuth protocol and the OIDC framework. The OAuth protocol is used to provide secure authorization, token expiry, and access token revocation. OIDC provides a secure authentication layer on top of OAUTH. The propagation of identity between microservices, and the resulting authorization decisions, are predominantly based on JWTs as opposed to proprietary approaches or other "heavier weight" token standards such as SAML.
- **TLS** (formerly known as SSL) remains the predominant method of securing HTTP communications (traffic) between all components, including between microservices. TLS uses public key infrastructure (PKI) to create secure sessions using asymmetric cryptography that involves the use of digital certificates that are extremely difficult to compromise. It is advisable to use two-way TLS – meaning both requestor and provider have their own digital certificate, and so both parties can trust whom the other party is.
- **API keys and shared secrets**: APIs and microservices are often further secured through the use of non-PKI encryption that uses symmetric cryptography. Symmetric cryptography requires both parties to use the same key (that is, a "shared secret") to encrypt and decrypt their messages. If you are planning to use API keys to authorize and validate calls that a microservice receives, be sure to follow secure practices for establishing and distributing the keys, and to regularly update (rotate) the keys, otherwise your security can be compromised if a rogue employee or hacker is somehow able to learn of the keys.
- **Data in transit and at rest**: While TLS is an excellent standard for encrypting traffic between components – including data in transit – some data are of high enough importance that they warrant further controls, such as tokenizing. Tokenizing replaces the value of, for instance, a Social Security Number or a credit card number with a token that the receiving party can then look up to get the actual value. Encryption of the data (such as via shared secret approaches mentioned above) within the TLS-encrypted session is also an option for additional security of highly sensitive data in transit. Encrypting data at rest (that is, data within a database or a file system) is also important and will also drive the need for key management servers for storing encryption keys and policies for their rotation. Consider using commercial encryption products such as IBM Security Guardium Data Encryption or IBM Cloud Data Encryption Services. The solution for IBM Bluemix key management is Key Protect. And for on-premises key management, consider an offering such as IBM Secure Key Lifecycle Manager.
- **White listing**: Application or microservice whitelisting is the practice of specifying a list of approved applications and services that are permitted to invoke your microservices. It can also include host and IP whitelisting as well, which further restricts which services

are permitted to call your microservices. Whitelisting is performed in addition to other practices such as two-way TLS. Consider whitelisting for environments that require exceptionally high security.

- **Black listing**: Black listing is the opposite of white listing. It specifies specific applications, microservices or servers that cannot access your microservice. Blacklisting is typically implemented by your network group, independent of specific applications or services as a means to restrict rogue sites or users from accessing your IT environment. Your network group may subscribe to various blacklist feed providers.
- **Toolchains**: Securing your DevOps toolchain is of critical importance in a microservices world. DevOps is critical for microservices; however, you must not inadvertently create security holes with your DevOps tools. Be sure to follow the vendor's recommended security configuration and operational practices, and be sure to include your toolchain in professional security penetration tests.
- **Command Line Interfaces (CLIs)**: Similar to your DevOps toolchain, ensure that any CLI tools you use are secured according to the tool's capabilities or by your workstation or server security controls. You may need to invest in Privileged Identity Management tooling that monitors the use of a user session to detect unauthorized usage.
- **Humans**: Automate wherever possible. Social engineering and human behavior weaknesses are the leading ways to compromise a system.
- **Identity propagation**: Identity propagation is critical in a microservices world. For instance, a user logged into an application may invoke a microservice that ends up calling another microservice. Both microservices will likely need to know the identity of the requesting user. Use industry standards approaches such as JWT described above to securely propagate identity.

**External considerations (that is, communication with other users or systems on the internet):**

- **TLS**: As with the internal consideration, TLS should be used to encrypt traffic. You may not, however, have the ability to enforce two-way TLS for end users, as web browsers typically do not have digital certificates installed for the individual. Servers typically have bona fide certificates installed.
- **Exposed publicly or only behind firewalls**: When exposing microservices as APIs, the architectural placement of your API gateway can be in front of or behind your firewalls. When in front, your gateway needs to be hardened to DMZ standards.

# Implementation decisions guide

You need to consider various implementation options and considerations when building your microservices. In this section, we address key topics related to implementation, including:

- Runtime platform selection

- Choosing between microservice frameworks
- Selecting a programming language
- Debugging your code
- Optimizing messaging approaches for integration
- Techniques for aggregating and orchestrating microservices
- Techniques for service discovery and re-use and more

This is perhaps the most rapidly changing aspect of microservices, so some of this guidance will evolve as technologies and practices mature.

## What programming languages should I write my different types (BFFs, Business, and Adapter) of microservices in?

Dispatchers are most often written in Node.js because it offers:

- Better fidelity with the clients
  - Especially clients that run JavaScript
  - iOS teams might want to use the Swift server-side runtime
- Support for numerous concurrent clients, which is useful because microservices are so I/O intensive

Business services are most often written in Java because Java is better for CPU-intensive tasks and for connectivity to external systems.

The following is a comparison of NodeJS vs. Java to help you determine what language you need for creating your microservices:

| NodeJS | Java |
|---|---|
| Higher performance for I/O | Higher processing performance |
| Easier async programming | Type safety for calculations |
| Fullstack/isomorphic development | Rich processing frameworks |

## When is it appropriate to use an asynchronous protocol for your Microservice?

Using an asynchronous protocol should be the exception rather than the rule for several reasons:

- REST is simpler to consume – there are HTTP clients for every language.
- REST has Swagger as a good documentation language.
- The Web's infrastructure, both internally and externally, makes it easier to secure and manage HTTP connections.

There are situations where a queued solution (a message broker like Rabbit MQ) is best. These situations occur when:

- The communication is naturally asynchronous (fire and forget)
- You are dealing with long-running processes (but use callbacks rather than a pseudo-synchronous polling approach)
  The size or order of the messages requires a more robust approach

When you have to use an asynchronous protocol, it's best to use standards like Kafka or AMQP wherever possible. Stick with the schema design you used for REST to reuse as many DTOs as possible. Implement the queued interfaces as their own independent microservices, so you can scale and manage them separately.

**What is the recommended approach to handle transactions that involve more than one microservice?**

This topic related to Atomicity, Consistency, Isolation, Durability (ACID) transactions involving more than one microservice. A microservices aggregate should strive to use commands and domain events to handle transaction consistency.

A common solution for a microservice follows these steps:

1. The microservice publishes events to a messaging queue
2. A listener consumes the message from the queue
3. The listener then inserts the message into the database without having to use XA/2PC transactions.

An event can be inserted into a dedicated event store that acts like both a database and a messaging publish-subscribe topic.

Another alternative is to use an ACID database and stream changes to that database to a persistent, replicated log like Apache Kafka and process the events with a separate component. The ultimate goal and principle to follow is to have communication between boundaries with immutable point in time events.

## How do you debug a microservices-based application problem when you can't pinpoint which microservice along that call path has the problem?

The simplest and most effective debugging tool for a complex microservices web is consistent use of correlation IDs. A correlation ID is a simple identifier (usually just a number) that is passed in to every service request and passed along on every succeeding request. When any service logs something for any reason, the correlation id is printed in the log entry. This allows you to match or correlate specific requests to one service to other service requests in the same call chain.

To implement correlation IDs correctly, follow these four consistent actions:

1. Create a correlation ID if none exists and attach it as a header to every outgoing service request
2. Capture the incoming correlation ID on every incoming request and log it immediately.
3. Attach the correlation ID to the processing of the request (through a threadlocal variable) and make sure that the same correlation ID is passed on to any downstream requests.
4. Log the correlation ID and timestamp of  any messages that are connected to that processing thread.

Once you have implemented a correlation ID, you can then use a log aggregator to gather together all of the logs across all of the dependent systems in your microservices architecture and can perform a query for the correlation ID against the aggregated log. When placed into timestamp order, the results of this query show the call graph of the solution and allow you to put errors into the appropriate context in terms of which microservice instance and thread handled the calls and what the parameters to the calls were at the time that the problem was encountered.

## What is a service registry and when do I need one?

A service registry is a repository for information about services. At a minimum, the registry is a mapping between a unique identifier and a service instance to decouple the physical address of a service from the identifier.

Often, service registries add other metadata about the service along with the physical address, such as a generic server name (useful when you have multiple instances of a service) and health information such as status or uptime.

Not all microservices projects require a services registry. If you have only a handful of services in your application, then the setup and management of the registry infrastructure is often more trouble than it's worth. But if you have more than six services, then it may become useful when alternative ways of managing service location (such as configuration files) become cumbersome.

## What factors should I consider when selecting a microservices framework?

Building a system using the microservices architectural style presents typical challenges of distributed systems design. The adoption of a microservices framework to address some of the most important cross-cutting concerns of the system is a common and welcome practice.

You should consider the following non-functional aspects when selecting your framework:

a. Community support and industry adoption

b. Stability of the code

The choice of framework is also associated with the runtime chosen for the microservice. Common technical capabilities that you should consider are:

a. Resource handling, for example, network locations of external services such as databases and message brokers
b. Security
c. Monitoring and logging to offer insight into what the application is doing and how it is performing

## What factors to consider when selecting a microservices runtime platform and what are my choices?

Platform as a Service environments reduce microservice operational burdens and increase runtime quality of service.

A microservices-oriented PaaS should provide an efficient way to define instance dependencies, scaling properties, and security policies as PaaS metadata or code scripts.

A PaaS environment that provides DevOps capabilities is also very desirable because it can reduce deployment complexity by automatically spinning up and linking instances.

An ideal PaaS provides the necessary building blocks to register service endpoint locations, associate metadata and policies, connect clients, circuit break around failures, correlate inter-service calls, and load balance traffic. It should also provide service registries, metadata services, discovery services, and service virtualization gateways.

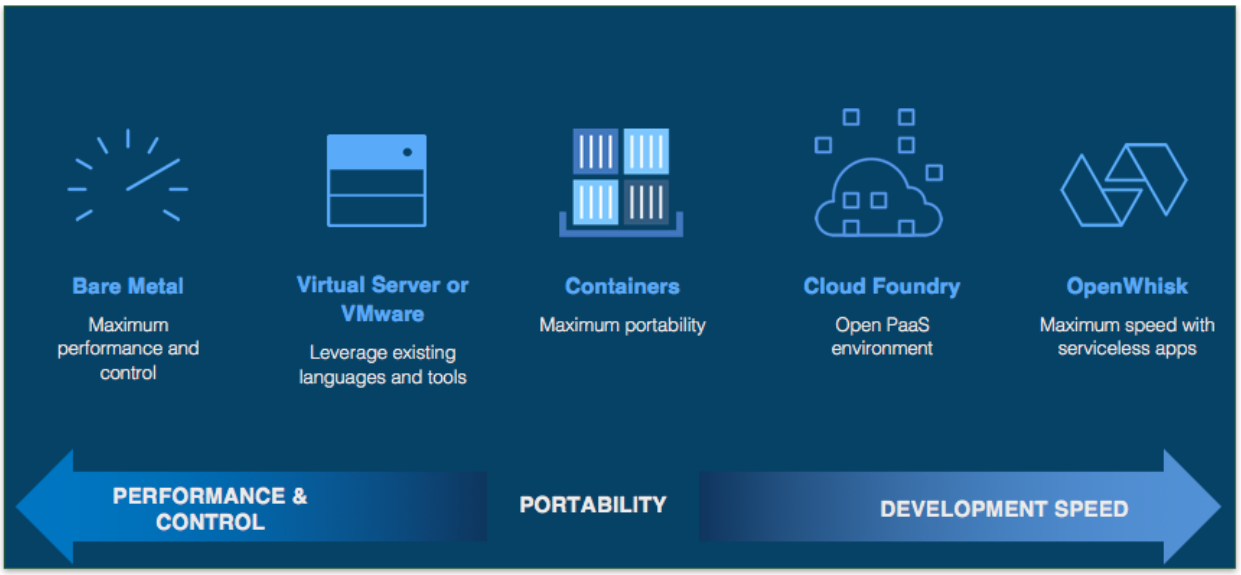



***Figure 7. Microservices compute options***

**Microservice runtime platforms:**

1. Event-driven – Offerings like Apache OpenWhisk and Amazon Web Services Lambda abstract all virtualization and implementation details away from the developer, allowing them to build applications that react to a series of events and, in-turn, easily chain multiple business functions together. As this type of compute platform is entirely managed, implementations are limited to instances where developers can truly be "hands-off".
2. Cloud Foundry - This open source Platform as a Service provides abstraction for how microservices run and communicate with each other and for virtualization such as containers. Often sought after for it's ease-of-use developer experience and providing some level of portability, it requires a fuller stack in the cloud environment to run.
3. Containers – Linux-based (and soon Windows-based) containers provide the most portability across cloud and on-premises environments. Docker is the name synonymous with this technology, based upon their very simple developer experience. Delivering successful container-based implementations requires a good handle on a strong DevOps culture.
4. Virtualization or bare metal – Microservices were being built well before cloud providers existed. Virtual machines and bare metal infrastructure provide the most control to any given developer, while also providing the most responsibility for on-going operation. Applications built using these technologies require DevOps expertise to succeed in an ever-advancing world, with the desire for cloud portability.

## What are the key component choices for the microservices fabric stack?

**Routing and discovery:**

1. Netflix OSS – the Eureka, Ribbon, and Zuul frameworks that are part of the Netflix OSS stack provide this capability. Use this if you plan to build microservices only in Java and are using other parts of the Netflix OSS stack and Spring Cloud.
2. Amalgam8 - provides routing and discovery capabilities. Use Amalgam8 when you plan to build microservices using various programming languages.
3. Apache Zookeeper - A widely used, high-performance coordination service for distributed applications. ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

**Logging and analytics:**

1. Elastic Search, Logstash, Kibana (ELK) – the ELK stack is a popular stack for logging and analytics. It is a good choice for microservices logging and analytics.
2. Splunk – many enterprises have invested in Splunk. Use Splunk if it is company standard and available.

**Logging, monitoring, and alerting:**

1. Graphite - is an enterprise-level monitoring tool known for performing well on systems with limited resources. It stores numeric time-series data and renders graphs of this data on demand. Graphite does not collect monitoring data.
2. Grafana - is most commonly used for visualizing time series data for internet infrastructure and application analytics
3. Collectd - gathers metrics from various sources, e.g. the operating system, applications, logfiles and external devices, and stores this information or makes it available over the network. Those statistics can be used to monitor systems, find performance bottlenecks (i.e. *performance analysis*) and predict future system load

**Messaging:**

1. Kafka - is an open source stream-processing platform developed by the Apache Software Foundation that provides a high-throughput, low-latency platform for handling real-time data feeds. Its storage layer is a massively scalable pub/sub message queue architected as a distributed transaction log.
2. RabbitMQ - is open source message broker software that implements the Advanced Message Queuing Protocol (AMQP), built on the Open Telecom Platform framework for clustering and failover.
3. MQ Light API - A simple yet powerful AMQP-based messaging API. Write apps that run locally, in the cloud, or alongside IBM MQ.
4. IBM Message Hub - a scalable, distributed, high throughput message bus in the cloud, based on Apache Kafka, available as a fully managed Bluemix service.

**Security:**

1. Basic Security – Traditional HTTP Basic Auth that is still widely implemented in applications today, but poorly supported in a distributed microservices architecture.
2. OAuth2 – An open-protocol based security implementation that allows a similar user experience, with common architecture components, across web applications, mobile platforms, and IoT devices. OAuth requires specific architecture components to support implementation, but many open source and enterprise offerings exist to build security from the ground up.
3. JSON Web Tokens (JWT) – An extension to the OAuth model and concept, JWTs are open standards-based headers that allow all actors in a microservices architecture to validate, verify, and generate authorization. Based on JSON initially, this specification can be utilized across many non-JavaScript applications as well.
4. Certificate based – Originally implemented as Secure Sockets Layer (SSL) and more commonly implemented as Transport Layer Security (TLS), certificate-based security allows developers to secure their architecture components using cryptographically secured keys that ensure communication is secret when going over any network interface.

**Config:**

1. Spring Cloud Config - provides server and client-side support for externalized configuration in a distributed system. It provides a central place to manage external properties for applications across all environments. As an application moves through the deployment pipeline from development to test and into production, you can manage the configuration between those environments and be certain that applications have everything they need to run when they migrate.
2. etcd – Another open-source externalized configuration manager, this reliable key-value store provides broader language bindings, more often better performance, and more easily clustered. Often available as a service from cloud service providers.
3. Consul – Provides service discovery capabilities. Consul is another highly scaled key-value store that is language agnostic with REST-based interfaces and broad language support.

## What are the key component choices for the microservices application stack?

Your microservices-based application is written using a language and a runtime stack. Examples include modern, lightweight Java EE stacks like the MicroProfile or Liberty, alternative Java stacks like Spring, or Node.js stacks like StrongLoop.

Let's first discuss the language choice: Java versus JavaScript. There are many other languages that can be used in the microservices arena, but at the time of this paper, those are the two most

prevalent contenders. Of course, you should assess the skills of your team when deciding which language to use as familiarity with a specific language will make the adoption easier. While Java is more popular, JavaScript adoption is gaining momentum..

JavaScript advantages:

- Ease of migration of code from server to browser and vice versa
- Simplified build process
- Easier access to newer, NoSQL databases
- Native JSON support

Java advantages:

- Better and more IDEs available
- Better tools for monitoring clusters of machines, hooks into JVM, and elaborate profiling tools
- Multithreading

There are multiple choices for the runtime engine in Java (Liberty, Microprofile, Spring), but JavaScript has primarily one runtime—Node.js..

**Application frameworks**

There are specific language libraries expected to handle certain elements of a microservices application such as circuit breaking, tracing, and so forth. Let's look at four popular libraries.

The **Spring Framework** is an application framework and inversion of control container for the Java platform. Spring has multiple projects. Some, like Spring Cloud, are ,geared towards addressing microservices concerns; Spring Boot is another project that addresses microservices, but it only supports Java.

**Hystrix** is a latency and fault tolerance library that isolates points of access to remote systems, services, and third-party libraries, stops cascading failure, and enables resilience in complex distributed systems where failure is inevitable. It is an implementation of the circuit breaker pattern described in this document, with original support for Java, but with bindings for other languages like Scala.

**Failsafe** is a lightweight, zero-dependency library that handles failures. It is also a circuit breaker implementation in Java.

**OpenTracing** offers a lingua franca for OSS instrumentation and platform-specific tracing helper libraries. OpenTracing libraries are available in six languages: Go, JavaScript, Java, Python, Objective-C, C++.

OpenTracing addresses user-facing latency optimization, root-cause analysis of backend errors, communication about distinct pieces of a now-distributed system, and more. There are other distributed tracing systems (for example, Zipkin, Dapper, HTrace, X-Trace, among others) that can require application-level instrumentation.

## What are the key component choices for the microservices DevOps stack?

As developers are empowered to make more influential decisions about the strategy of their company, the tools available to them are becoming equally powerful and diverse.

There are many tools for version control of source code (source control management or **SCM)** as well as code collaboration tools, each with advantages and disadvantages. Opting out of version control isn't an option.

GitHub has emerged as one of the most popular tools in the cloud development arena (along with Git for source control) and is gaining steady adoption in enterprises as well. Other open source tools, as well as commercially developed and supported tools (such as IBM Rational Team Concert), are available and used with microservice development. Choose a tool based on its capabilities, your team's skills and support needs, and your enterprise standards.

**Continuous Integration** (CI) allows for automation of tests and builds and notifications about failed builds—including information about who caused the trouble and in which piece of code. One of the most popular tools for CI is Jenkins, an open source automation server, although several other open source tools exist, as well as commercial offerings.

Application test automation and test virtualization technologies are also key to successful CI, as well as for achieving continuous delivery (described below). Popular test automation and virtualization tools vary by language, platform, and device, but include popular open source products like Junit and Crittercism, as well as commercial products like IBM Rational Test Workbench.

**Continuous delivery (CD)** enables you to deploy artifacts automatically or with minimal effort. One of the most popular application deployment automation tools in the market are commercial products - IBM UrbanCode Deploy and IBM UrbanCode Release. However, open source tools like Chef and Puppet are also actively used.

Operation management

- Chef is a configuration management tool that uses a domain-specific language (DSL) for writing system configuration "recipes". Chef is used to streamline the task of configuring and maintaining a company's servers, and can integrate with popular cloud-based platforms, including SoftLayer, to automatically provision and configure new machines.

- Pingdom is a service that tracks the uptime, downtime, and performance of websites. Pingdom monitors websites from multiple locations globally so that it can distinguish genuine downtime from routing and access problems.
- Apica provides real-time user synthetic monitoring, providing metrics on response times, monitor synthetic transactions, and evaluate visitors' experiences.
- DataDog is a SaaS-based data analytics platform, providing monitoring services for cloud-based applications.
- Catchpoint is a real-time performance analytics solution relying on synthetic monitoring.
- Radware is a provider of load balancing and cybersecurity services for data centers.

Log management

- Loggly is a cloud-based log management service provider that uses open source technologies, including Elasticsearch, Apache Lucene 4, and Apache Kafka.
- Logentries - is a cloud-based platform that collects and analyzes logs across software stacks using a pre-processing layer to filter, correlate, and visualize log data. It uses a combination of AWS tools and open source applications to deliver the service.
- Sumologic – is a cloud-native, machine data analytics service for log management and time series metrics.

Configuration Management: Configuration Management remains important in a microservices world. Open source tools like Chef and Puppet are available, as are commercial tools such as the IBM UrbanCode Deploy Configuration Management plug-in.

# Resiliency decisions guide

Improving your application's resiliency is one of the driving forces for microservices adoption. Distributed systems can and will fail, and microservices architectures introduce important resiliency requirements. These include high availability (HA), failover, disaster recovery (DR), circuit breaking, isolation, and so forth.

When dealing with improved resilience, it's important to first address the classic high availability and disaster recovery topics that are relevant to any IT solution. This is especially important when working with microservices architectures.

Read the companion paper to this paper, Microservices Point of View Guide, to understand the importance of using platform, infrastructure, and middleware capabilities to achieve classic HA and DR. This includes, for instance, the use of clustering and load balancing within a data center site using Bluemix Autoscaling services, and establishing Active/Active processing across two or

more sites, using global load balancing services like Akamai and Dyn. Redundancy of the backends that are used by your microservices (for example, database Systems of Record) continue to be important in a microservices architecture.

The decision table below introduces three resiliency patterns that are key to microservices; we provide guidance on their selection as well as guidance on testing the resiliency of your microservices.

## What are the recommended resiliency design patterns for microservices based architectures?

### Circuit Breaker Pattern

A circuit breaker is a switch that automatically toggles itself off when there is an overload or short-circuit. In a microservices ecosystem, a circuit breaker can be activated when a service becomes slow to respond or fails completely to prevent it from being exposed to more requests.

Circuit breakers allow user code to check if external dependencies are available before actually connecting to the external system.

A circuit breaker keeps track of which services fail and, based on thresholds, decides if a service should be used or not. The circuit breaker also hides complexity from the end user code. It keeps statistics hidden and gives simple answers: available or not.

A circuit breaker can be placed anywhere between the consumer and the API provider. It is preferable to place it closer to the consumer to comply with the fail-fast principle:

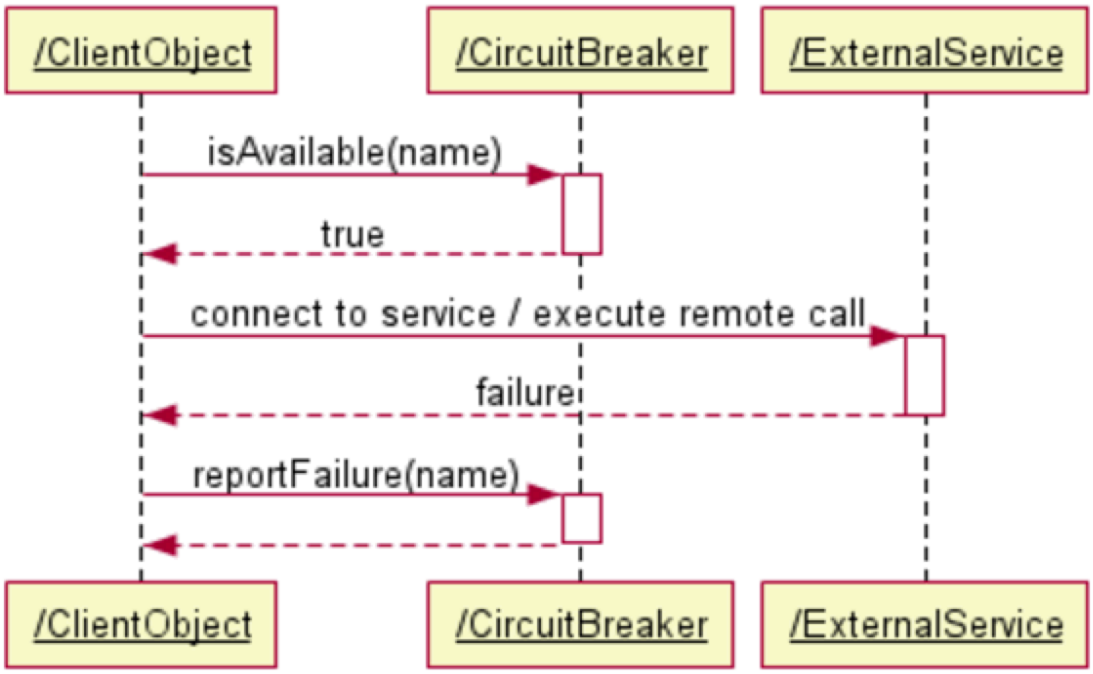



***Figure 8. Circuit Breaker pattern diagram***

**Handshaking Pattern**

By asking a component if it can handle more work before asking it to perform that work, the system has a way to introduce throttling. If the component is too busy, it can tell the clients to back off until it is able to handle more requests.

This pattern can be considered a variant of the Circuit Breaker Pattern described above, establishing a state of "partially ON," aside from the typical states "ON" and "OFF" of a simple breaker.

**Bulkhead Pattern**

The Bulkhead Pattern prevents faults in one part of a system from taking the entire system down. The term comes from ships. A ship is divided into separate, watertight compartments to prevent a single hull breach from flooding the entire ship; it will only flood one bulkhead.

Depending on what type of faults you want to protect the system from, your implementation of this pattern can take many forms.

Netflix Hystrix is one of the most popular implementations of the Bulkhead Pattern. The bulkhead implementation in Hystrix limits the number of concurrent calls to a component. This way, the number of resources (typically threads) that is waiting for a reply from the component is limited.

Hystrix has two different approaches to the bulkhead implementation: thread isolation and semaphore isolation.

## How do I test the resiliency of my system?

One of the key aspects of the microservices architecture is that each microservice has its own lifecycle. Each microservice is owned and operated by an autonomous team. Different teams can independently develop, deploy, and manage their respective microservices as long as they maintain API compatibility. This agility, when combined with continuous integration and deployment tools, enables applications to be deployed tens to hundreds of times a day.

### Chaos engineering

Chaos engineering addresses the uncertainty of distributed systems at scale. Chaos engineering is the facilitation of experiments to uncover systemic weaknesses.

These experiments follow four steps:

1. Define ‘steady state’ as some measurable output of a system that indicates normal behavior.
2. Hypothesize that this steady state will continue in both the control group and the experimental group.
3. Introduce variables that reflect real-world events like servers that crash, hard drives that malfunction, network connections that are severed, and the like.
4. Try to disprove the hypothesis by looking for a difference in steady state between the control group and the experimental group.

The harder it is to disrupt the steady state, the more confidence you can have in the behavior of the system. If a weakness is uncovered, you now have a target for improvement before that behavior manifests in the system at large. (Reference: http://principlesofchaos.org/#sthash.fZ9Jn2Vf.dpuf)

Automation is key to resilience testing as well. Frameworks like Gremlin and tools like Simian Army Chaos Monkey ensure that your applications can tolerate random instance failures.

### Chaos Monkey

The Chaos Monkey is the first entry in the Netflix technical team's Simian Army. Chaos Monkey randomly terminates virtual machine instances and containers that run inside of an environment.

### Gremlin

This framework lets you systematically test the failure-recovery logic in microservices in a way that’s independent of the programming language and the business logic in the microservices.

Gremlin takes advantage of the fact that microservices are loosely coupled and interact with each other solely over a network. Gremlin intercepts the network interaction's microservices (for example, REST API calls) and manipulates them to fake a failure to the caller (for example, return HTTP 503 or reset TCP connection). By observing from the network how other microservices are reacting to this failure, it is now possible to express assertions on the behavior of the end-to-end application during the failure. In production environments, Gremlin's fault injection can be limited to only synthetic users, so that real users remain unaffected.

# Operations decisions guide

The ability to monitor, manage, and provide continuous feedback on the runtime operations of your microservices is critical for the long-term success of your microservices strategy. It is imperative to evaluate your operational readiness to support microservices and make the necessary investments in capacity, be it through investments in people, processes, or technology. Compared to the traditional monolithic approach, a microservices strategy introduces new pressures and velocities around change and response.

When problems do occur in production, it is easier with microservices to identify and isolate the problem. By identifying smaller, non-responsive microservice processes or reviewing log files marked with a microservice team identifier, operations can quickly identify who from development should be brought in to troubleshoot the problem, all in a self-contained team. When problems occur, the team that owns the microservice has a greater incentive to implement measures to prevent similar outages or incidents in the future.

Tuning a microservices-based system, which can consist of dozens of processes, load balancers, and messaging layers, requires a high-quality monitoring and operations infrastructure. Promoting the plethora of microservices through the development pipeline into production requires a high degree of release and deployment automation.

In this section, we offer guidance for both new application ("green field") development, as well as guidance for when you are considering refactoring an existing monolithic app ("brown field" development).

### Do your change and release management practices and tools embrace DevOps?

Changes in a monolithic model require high overhead. Significant coordination of activities is needed because a monolithic application usually has a large business scope and many infrastructure and operational touch points. As a result, each change to the application might require many reviews and approvals from different stakeholders.

By contrast, individual microservices have a much smaller business scope and fewer touch points, reviews, and approvals – which is typically embraced by operations and change teams as well as other stakeholders. However, the quantity of microservices and the frequency in which they change (maybe even daily) will put new stresses on these teams if they follow old, manual and inflexible processes. Microservices need to use automation in every stage of the service delivery pipeline in a way that promotes self-service and minimizes manual coordination work.

Successful organizations use a DevOps toolchain that supports development, deployment, and operations tasks. The collective power of a toolchain is greater than the sum of its individual tool integrations. View an example of a microservices DevOps toolchain in IBM Bluemix or additional toolchain options and description of supporting tools.

In all cases, you will achieve continuous delivery (deployment automation) success by configuring an automated delivery pipeline for each microservice application – building on to the continuous integration practices that your development teams are performing. In other words, you should strive to continuously release into production every good build that has passed its necessary tests and controls.

In addition to tooling, this will also require you to revisit any inflexible roles and lines of responsibilities between Development and Operations and move to more of a shared ownership and roles model.

As described in the blog post, DevOps for the Enterprise: RACI Destroys DevOps, a typical pre-DevOps RACI (Responsible, Accountable, Consulted, Informed) model outlining roles and responsibilities needs to adapt to more of a SORE (Shared Ownership and Roles based on Expertise) model.

The pre-DevOps philosophy articulates whose job it is to perform a certain function, which is less than desirable in a DevOps world. As described on the OneGeek web site,

> "*It sends a signal to the group, that it is not my responsibility for other aspects of delivery – quality, shipping, reliability etc. It creates friction in cooperation in the group – I can't trust somebody else to do my job, to meet my KPI expectations. Can I trust that Dev to push his code to Production or am I going to be woken up at 3am again? Possibly most importantly, as a side-effect of the above, it reduces the surface area of knowledge spread across people. This means I now have less people who understand the entire solution.*"

By contrast, the DevOps philosophy benefits from a SORE model, which empowers all members of the team to contribute to the entire solution. It is beneficial if everyone understands how the solution is built and deployed, can support the system when it fails, and is trusted and encouraged to do so. Ideally the broader team is assigned shared goals whose KPIs are aligned.

## Do your incident/event management practices and tools currently include continuous monitoring of all infrastructure, platform, and applications to ensure that they are functioning optimally?

Microservices require high observability. Your service management team requires the tools to oversee the status of each microservice in the system or across products, and to be notified of infrastructure and applications events. This improves service management productivity and increases your ability to perform effective incident management.

Verify that your existing incident management tools can support your adoption of microservices. If you have gaps, consider tools such as these:

**Resource monitoring** evaluates the resources that support managed solutions or enterprise applications.

- IBM Application Performance Management
- IBM Monitoring and Analytics for Bluemix
- New Relic
- IBM Bluemix RSS/JSON feed (Bluemix alert capturing service)

**Log Monitoring** to watch and analyze structured or unstructured logs with elastic search of support managed solutions or enterprise applications

- ELK (Elasticsearch, Logstash, Kibana)
- Splunk

**Event correlation and consoles** to identify and isolate issues by consolidating all alerts across managed solutions and enterprise application into a small number of actionable alerts

- IBM Netcool Operations Insights
- QRadar (for security incidents).

**Collaboration** to support cloud-based team collaboration that is searchable:

- Slack

**Notifications** that alert on-call users based on notification rules of the triggered incidents

- IBM Alert Notification for Bluemix
- PagerDuty

**Dashboard** to visualize and template time-series metrics and analysis in a dynamic and interactive format

- Grafana
- IBM Dashboard Application Services Hub (DASH)
- Bluemix Status Console.

**Runbook** to help users define, build, orchestrate, automate, and manage script-based operational tasks as runbooks
- IBM Runbook Automation

There can be significant variation in incident management capabilities across platforms. For instance, logging can vary significantly for Docker Datacentre (DDC) when deployed on-premise, versus in a Microsoft Azure cloud, versus in AWS cloud, versus in an IBM Bluemix cloud.

## Do you have the ability to set alerting thresholds for what is considered optimal?

At minimum, collect the following metrics, against which to set alerting thresholds on your microservices:

- Number of simultaneous requests being served
- Execution time of each request (under the provider perspective)
- Response time of each request (under the consumer perspective)
- Network latency introduced
- Number of simultaneous connections being served
- Token expirations
- Authentication errors
- Execution errors
- Circuit breaker activations
- Number of workers per service
- Load distribution

Ideally, you should also collect and receive alerts about your application users' behavioral data, including navigation patterns, clicks, and visualizations. This helps you identify areas of customer struggle (that is, signifying potential issues with the application) and also provides valuable insight to your marketing and product teams to optimize the application, including adding new value-added features with cross-sell/up-sell potential.

Many tools allow the customer to provide feedback directly to you . This form of continuous feedback is a valuable part of the DevOps process. Tools to consider include:

- IBM Tealeaf Customer Behavior Analysis Suite
- IBM Digital Analytics
- IBM Mobile First Quality Assurance
- Open Web Analytics (OWA)
- Webalizer
- W3Per

## Are your remaining IT service operations-focused disciplines ready to embrace microservices?

Other service operations-focused disciplines include:

- IT Operations and Control, which addresses backup and recovery and other routine maintenance tasks
- Problem Management
- Request Fulfillment

It is important that your Operations teams are well informed and collaborate with the development teams to gain up-front awareness of the remaining operational requirements. Additionally, operations teams also need to be able to provide levels of service that will ensure the ongoing successful management of your microservices-based applications.

In general, existing toolsets and processes can be used as long as they are flexible enough to support the volume, frequency, and granularity of your portfolio of microservices. Note, too, that managed cloud service providers (such as the IBM Bluemix) perform a variety of IT Service Operations functions on your behalf.