TWELVE FACTOR APP

Twelve-factor methodology

The twelve-factor app (12factor.net) is a methodology defined by the developers at Heroku for building platform-as-a-service apps that have these qualities:

- Use declarative formats for setup automation, which minimizes time and cost for new developers joining the project
- Have a clean contract with the underlying operating system, offering maximum portability among execution environments
- Are suitable for deployment on modern cloud platforms, removing the need for servers and systems administration
- Minimize divergence among development and production, enabling continuous deployment for maximum agility
- Can scale up without significant changes to tooling, architecture, or development practices

You'll see cloud native used interchangeably with twelve-factor app. Twelve-factor is the methodology, and cloud native refers to a computing environment and its tools. More details are available from the Cloud Native Computing Foundation.

The twelve-factor app is a set of best practices for creating apps that includes implementing, deploying, monitoring, and managing. It is a methodology that defines 12 factors that services should follow so that you can build portable, resilient apps for cloud environments. Web apps that deliver software-as-a-service (SaaS) are often cited as common examples of twelve-factor apps.

Twelve-factor apps have become a key yardstick by which components are measured to establish whether they are truly ready for cloud native deployment. It seems reasonable therefore that if you were to pursue lightweight integration, then you will want to use these principles there too to create a twelve-factor integration.

Twelve-factor apps can be implemented in any programming language and use any backing services such as database, messaging, and caching.

Finally, a twelve-factor app has characteristics that are ideal for developing individual microservices. The 12 factors ensure that an app can take advantage of the cloud infrastructure to deliver agility and scalability benefits. Although 12factor.net does not refer specifically to microservice architecture, it is typically used to define the appropriate characteristics for apps that are built as microservices. Apps built using the microservice approach typically need to integrate with data sources outside of the microservice architecture, for example, to integrate with systems of record built decades ago or services provided by a partner.

The twelve factors

To effectively design and build apps for the cloud, you should understand the twelve factors:

I. **Codebase**: One codebase tracked in revision control, many deployments
II. **Dependencies**: Explicitly declare and isolate dependencies
III. **Config**: Store configuration in the environment
IV. **Backing services**: Treat backing services as attached resources
V. **Build, release, run**: Strictly separate build and run stages
VI. **Processes**: Execute the app as one or more stateless processes
VII. **Port binding**: Export services via port binding
VIII. **Concurrency**: Scale out via the process model
IX. **Disposability**: Maximize robustness with fast startup and graceful shutdown
X. **Dev/prod parity**: Keep development, staging, and production as similar as possible
XI. **Logs**: Treat logs as event streams
XII. **Admin processes**: Run admin/management tasks as one-off processes

1. Codebase

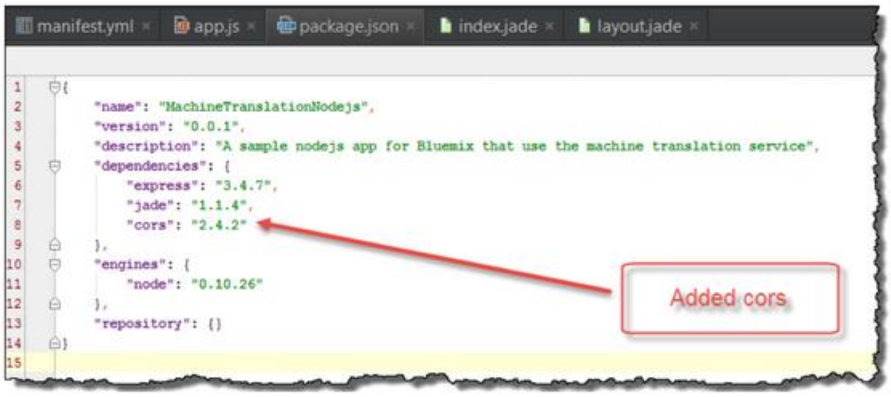
Use one codebase for all the code in your app, which is stored in a revision control system, supporting many deployments. For example, units of deployment could be Docker containers, Cloud Foundry apps, or a TAR file made up of a Helm chart.
Each microservice should have its own project with its own main branch and membership but should all be stored in the same repository.
Tip: To automate deployment, you can use popular tools such as Gradle or Jenkins.

2. Dependencies

Explicitly declare and isolate your dependencies. This is common problem for many developers. Build packs take care of many of these dependencies for you. Never rely on systemwide dependencies. Here are typical language-specific configurations:

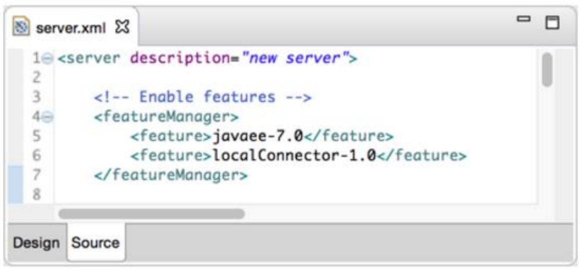
- Node.js: Node Package Manager (npm)
- Liberty: Feature manager
- Ruby: Bundler
- Java EE: Application resources

For example, a Node.js app is described by an npm package file (typically named package.json). Among other configuration details, it lists what other packages this package depends on. The app configuration is part of the app, and you manage the configuration in source control, for example, GitHub.



```
{
    "name": "MachineTranslationNodejs",
    "version": "0.0.1",
    "description": "A sample nodejs app for Bluemix that use the machine translation service",
    "dependencies": {
        "express": "3.4.7",
        "jade": "1.1.4",
        "cors": "2.4.2"
    },
    "engines": {
        "node": "0.10.26"
    },
    "repository": {}
}
```

Another example is the server.xml file for WebSphere Liberty. The server is configured by its server.xml file. In that file, the Feature Manager is configured to enable (or include) the Liberty features that are required by the apps that will run in the server. The server configuration is part of deploying the app, and once again, you manage it in source control.



```
<server description="new server">

    <!-- Enable features -->
    <featureManager>
        <feature>javaee-7.0</feature>
        <feature>localConnector-1.0</feature>
    </featureManager>
```

3. Configuration

Store the configuration in the environment, for example, the configuration location of a database in test versus production. The configuration can change, and changes are tracked effectively. Enable the same code to be deployed to different environments:

- Store configuration in the environment
- Separate configuration from source

Configuration information might include:

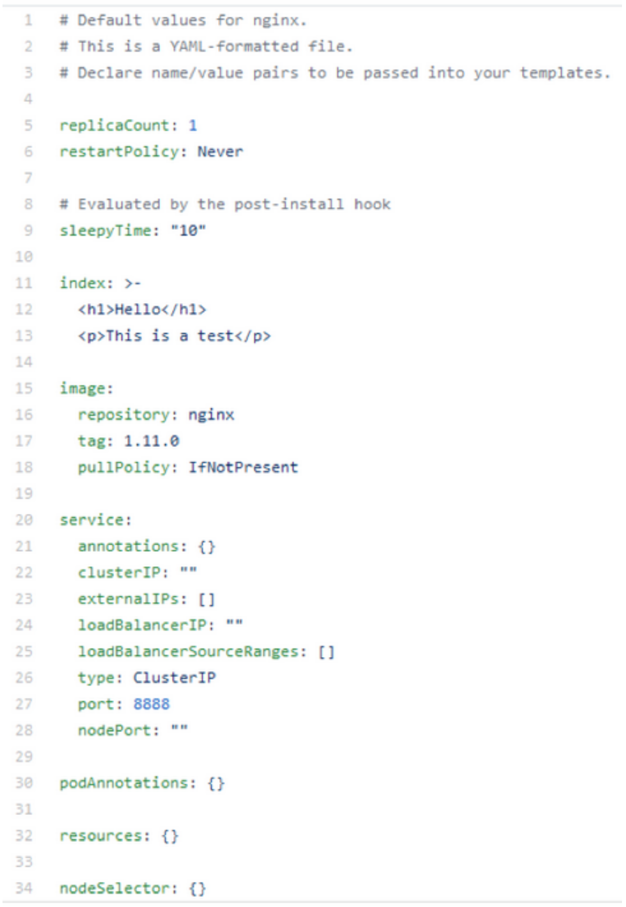
- Resource handles to databases and other backing services
- Credentials to external sources (for example, Twitter)
- Per-deploy values (for example, canonical hostname for deployment)
- Anything that is likely to vary among deployments (dev, test, stage, prod)

Store the configuration in the environment. Do not store it in these locations:

- The code
- Properties files (considered part of the code)
- The build (one build, many deployments)
- The app server (for example, JNDI data sources)

For example, for a containerized application to be deployed to Kubernetes clusters, the configuration can be stored in a YAML file. Among other things, the YAML file specifies the replica sets, networking, and health checks. Again, the container configuration is part of deploying the app, and you manage it in source control for your deployment files.

Here is an example for an nginx server. The Helm values.yaml file exposes a few of the configuration options in the charts, though some are not exposed there. This example file does not include user names or passwords, which exemplifies the statement "…the codebase could be made open source at any moment, without compromising any credentials."



```yaml
# Default values for nginx.
# This is a YAML-formatted file.
# Declare name/value pairs to be passed into your templates.

replicaCount: 1
restartPolicy: Never

# Evaluated by the post-install hook
sleepyTime: "10"

index: >-
  <h1>Hello</h1>
  <p>This is a test</p>

image:
  repository: nginx
  tag: 1.11.0
  pullPolicy: IfNotPresent

service:
  annotations: {}
  clusterIP: ""
  externalIPs: []
  loadBalancerIP: ""
  loadBalancerSourceRanges: []
  type: ClusterIP
  port: 8888
  nodePort: ""

podAnnotations: {}

resources: {}

nodeSelector: {}
```

The nginx Helm chart shows how to compose several resources into one chart, and it illustrates more complex template usage. It creates a replica set, a config map, and a service. The replica set starts an nginx pod. The config map stores the files that the nginx server can serve.

4. Backing services

Backing services are treated as attached resources. They might be running on the same compute, a different compute, or on third-party hosts. You need to treat these as attached resources that can be bound.
Tips:

- Treat the following backing services as attached resources:
    - Databases
    - Messaging systems
    - LDAP servers
    - Others
- Treat local and remote resources identically.

5. Build, release, run

Separate the ways in which you build, release, and run apps. A codebase is transformed into a (nondevelopment) deployment through three stages:

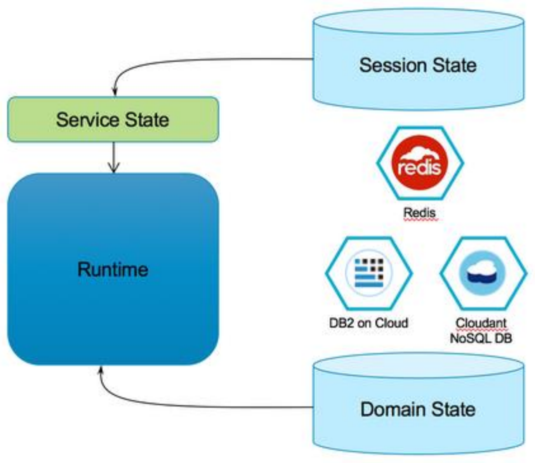
- **Build stage**: A transform that converts a code repository into an executable bundle known as a build. Using a version of the code at a commit specified by the deployment process, the build stage fetches dependencies and compiles binaries and assets.
- **Release stage**: Takes the build produced by the build stage and combines it with the deployment's current config. The resulting release contains both the build and the config and is ready for immediate execution in the execution environment.
- **Run stage** (also known as runtime): Runs the app in the execution environment by launching some set of the app's processes against a selected release. The runtime code should not be modifiable because there's no way to put those changes back into the build stage.

6. Processe

Run the app as one or more stateless processes. This is one of the most important things to keep in mind when you are trying to build a truly scalable, distributed, cloud-ready app. Because there is no state within a particular process, if you lose a process, it does not matter because the traffic is automatically and seamlessly routed to other processes in the environment that can handle that work.
Tips:

- Do not rely on session affinity, also called sticky sessions.
- Store state in a stateful backing service that is external to the process.

Runtimes should be stateless, but the services can or do have state.



This graphic illustrates how the state is stored in services and passed into the runtime. The runtime uses the state long enough to perform a unit of work and then throws it away. The runtime does not maintain state between units of work.
Here are a few key details on app state:
Service state:

- Data passed into the stateless service
- Scope: Transaction or unit of work
- Example: Service operation parameters

Session state:

- Data for history of a user session
- Scope: Session
- Example: HTTP session

Domain state:

- Data available to multiple or all services
- Scope: Global
- Example: Enterprise database of record

Usage:

- Service state is populated by service client, often from session state
- Service state often contains keys, used to retrieve domain state

7. Port binding

Export your services with port binding. This action is typically taken care of for developers by the operational and deployment model, but developers need to ensure that they do not hardcode port values. The app should be able to run and connect to services without manually specifying configuration values.
As an example, a web app binds to an HTTP port and listens for requests that come in on that port.

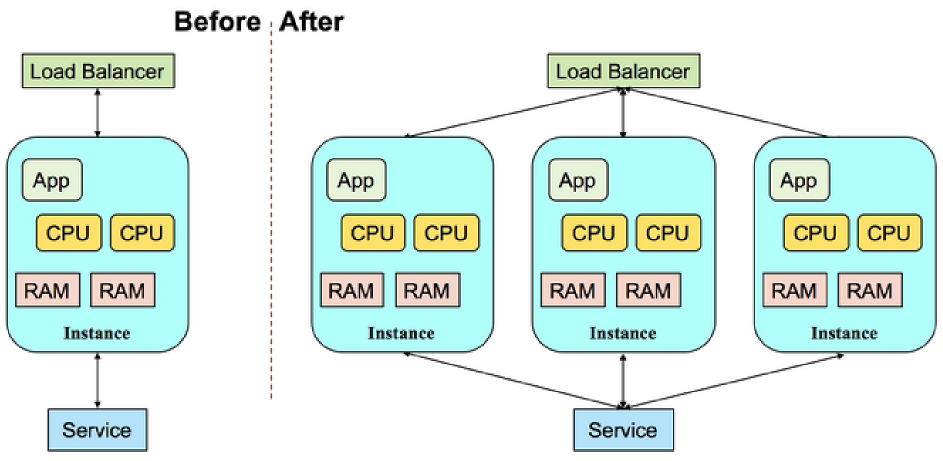
8. Concurrency

Use the process model to scale out. Another way to think of this tenet is the concept of horizontal scaling.
Tips:

- Scale out, not up, using the process model.
- To add capacity, run more instances.
- There are limits to how far an individual process can scale.
- Stateless apps make scaling simple.

With vertical scaling, you run more instances of the app. One runtime runs more instances, so the runtime requires more CPU and memory, which makes it bigger, and all of the CPU and memory have to belong to a single host.

With horizontal scaling, you run more instances of the app and run more runtimes, which keeps each runtime the same size. More runtimes still require more CPU and memory, but they can be distributed across multiple hosts, so all of the CPU and memory doesn't need to belong to a single host. The following graphic shows an example of horizontal scaling.

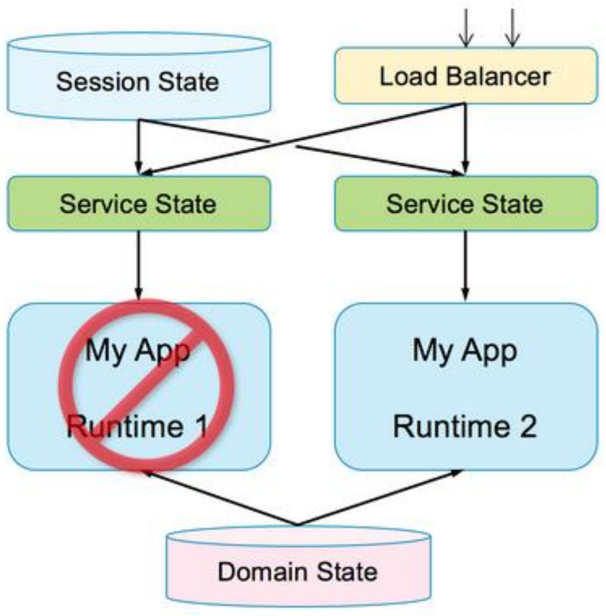


9. **Disposability**

Ensure that processes start fast and gracefully shut down, which means that processors should start almost instantaneously. When you shut them down, there should be no housekeeping or extra work that you must do.
Tips:

- App instances are disposable.
- Apps should handle shutdown signals or hardware failures with crash-only design.
- Note that containers are built on the disposability tenet.

Consider failover. Do not get attached with your runtimes; they are not meant to last forever. Runtimes are immutable, so each is as good as any other. Add or remove them as needed for elasticity. If one dies or becomes unhealthy, kill it, and start another just like it.
This graphic shows how an app can be run in two instances. When one instance is shut down or dies, its load fails over to the remaining instance, such that all the load goes to the remaining instance.



10. Development and production parity

Make your development, staging, and production environments as similar as possible. This tenet ties into agile software delivery, continuous integration, and continuous deployment concepts. You do not want development or production to get so out of sync that one does not reflect the other, which means you cannot fix either. You want them to be reasonably similar.
Tips:

- Use the same backing services in each environment, such as deployment toolchains.
- Minimize incompatible elements across environments:
    - Backing services
    - Tools
    - Platforms

11. Logs

Treat your logs as event streams, meaning that the app does not write or manage log files. Without correct telemetry in your environment, you have no idea what is going on. When complex issues arise, they might be difficult to fix and understand.
Each process writes to stdout:

- Apps should not write to specialized log files.
- The environment decides how to gather, aggregate, and persist stdout output.

An example of a log tool is the ELK stack (Elasticsearch, Logstash, and Kibana) that streams, stores, searches, and monitors logs.

12. Admin processes

Set up processes for your administrative or management tasks, even one-off processes. Do not create admin processes that might be repeated many times that are not well encapsulated as code themselves. It is important to create a process, even if it is a one-off process that does the particular task.
Examples of admin processes:

- Migrate a database
- Create an SQL database:
    - DDL script that creates the schema
    - SQL script that populates initial data
    - Script that runs these scripts as part of creating the database
- Migrate data to a new schema
- Debug

When you perform a one-time only admin task, do not assume it will be one-time only. Assume that you will need to run it once per environment (like Dev, Test, and Prod), or per deployment, or each time a problem mysteriously reoccurs. Because you will need to run it repeatedly, make it easily repeatable by making it into a script or program. That executable can be run with a single command, maintained to keep it working, stored in source code management, versioned, and shared with other developers and administrators. Even if it is only ever used once, at least you have a record of what you did.

Summary of the twelve-factor app
As you build 12-factor apps, you'll look for a hosting environment that inherently supports the 12-factor tenets. Portable, flexible containers are a natural location for you to consolidate tooling and operational needs for your apps. Later in this course, you'll hear how IBM Cloud Kubernetes Service supports 12-factor development. For example, the service provides integrated operational tools for logging and monitoring, Kubernetes for scaling, portability across dev/test/prod, isolation of workloads, and more.
To summarize briefly, a twelve-factor app should be designed and maintained in these ways:

1. Stored in a single codebase, tracked in a version control system: one codebase, many deployments
2. Has explicitly declared and isolated external dependencies

3. Has deployment-specific configuration stored in environment variables, and not in the code
4. Treats backing services (for example, data stores, message queues, and so on) as attached or replaceable resources
5. Built in distinct stages (build, release, run) with strict separation among them (no knock-on effects or cycles)
6. Runs as one or more stateless processes that share nothing, and assumes process memory is transient
7. Is completely self-contained and provides a service endpoint on well-defined (environment determined) host and port
8. Managed and scaled through process instances (horizontal scaling)
9. Disposable with minimal startup, graceful shutdown, and toleration for abrupt process termination
10. Designed for continuous development and deployment with minimal difference between the app in development and the app in production
11. Treats logs as event streams: the outer or hosting environment deals with processing and routing log files
12. Keeps one-off admin scripts with the app to ensure the admin scripts run with the same environment as the app itself

MICROSERVICES

Unmaintainable, monolithic apps

Back in 2005, the apps for each of Expedia's travel sites were too difficult to modify.
Each site was an interconnected tangle of code. If programmers wanted to add the ability to handle credit card billing in Germany, for example, that might end up causing errors in the flight-search feature.
As a result, changes were rare: A site would be updated only a couple times a year.
Source: You: Expedia has bet everything on understanding the psyche of the modern traveler in Bloomberg Business Week, February 29, 2016.
Don't let this happen to you anymore!
We've all worked on apps like this. The code has become such a mess that you can't change anything without breaking something else. And even after you get some code fixes in, building and deploying a new version of the app is a nightmare. As a result, even when users ask for a simple change, it takes a long time to get the updated app redeployed.
This is a call to action: We need to do better!

What are microservices?

Microservices are an app architectural style that divides an app into components where each component is a full but miniature app that's focused on producing a single business task according to The Single Responsibility principle.
A microservices design implements tasks from beginning to end: From the GUI to the database, or at least from the service API to the database so that different GUIs and client apps can reuse the same business task functionality. The business task is meaningful to the business users, meaning no technical or infrastructure microservices.
Each microservice has a well-defined interface and dependencies (for example, to other microservices and to external resources) so that the microservice can run fairly independently, and the team can develop it fairly independently.

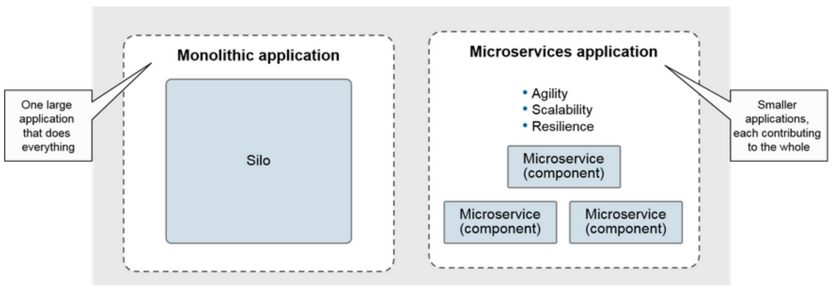
Microservices: Making developers more efficient

Microservices are often described in terms of using technology better. But microservices also make developers (not just computers) more efficient.

It does so by enabling them to accomplish meaningful work while working in small teams. Small teams make developers (and people in general) more productive because they spend less time in meetings (and otherwise communicating with and coordinating with others) and more time developing code. Microservices accelerate delivery by minimizing communication and coordination among people, and reducing the scope and risk of change.

Microservices architecture

The aim of a microservice architecture is to completely decouple app components from one another such that they can be maintained, scaled, and more.



It's an evolution of app architecture, service-oriented architecture (SOA), and publishing APIs:

- SOA: Focus on reuse, technical integration issues, technical APIs
- Microservices: Focus on functional decomposition, business capabilities, business APIs

Microservice architecture as summarized in Martin Fowler's paper would have been better named micro-component architecture because it is really about breaking apps up into smaller pieces (micro-components). For more information, see Microservices by Martin Fowler.

Example app that uses microservices

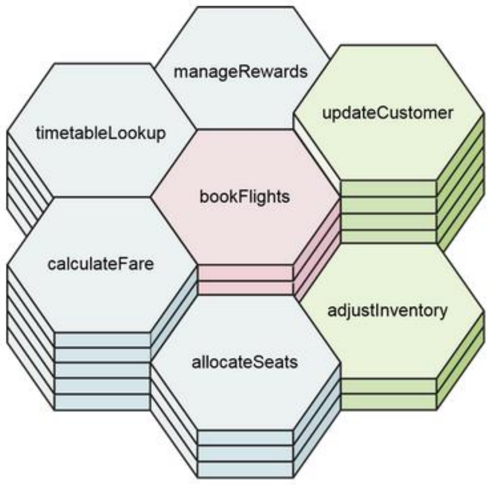
Here's an example of an app with a microservices architecture that demonstrates business tasks as components.

This example app is for booking airline tickets. The app needs these components:

- Logging
- Metrics
- Health check
- Service endpoint
- Service registry
- Service management

Each of these components is a fairly independent business task that can be developed separately and in priority order. Each means something to the business users and each has quality of service (QoS) targets.

For example, in the following image, you can imagine that an airline booking app has components such as fare calculations, seat allocation, flight rewards programs, and so on.

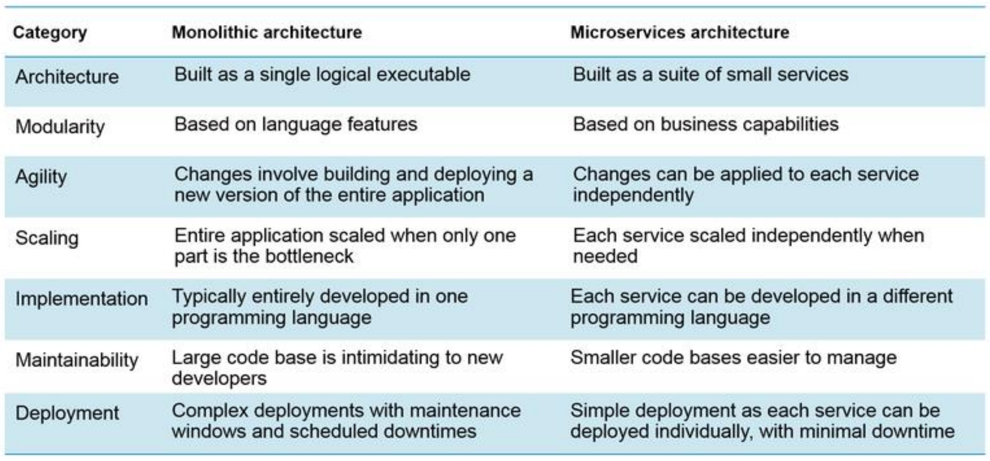


Key tenets of a microservices architecture

These tenets test whether the architecture you're implementing is microservices. The more you're implementing these qualities into your app, the more you're on the right track.

- Large monolith architectures are broken down into many small services:
    - Each service runs in its own process.
    - The applicable cloud rule is one service per container.
- Services are optimized for a single function:
    - There is only one business function per service.
    - The Single Responsibility Principle: A microservice should have one, and only one, reason to change.
- Communication is through REST API and message brokers:
    - Avoid tight coupling introduced by communication through a database.
- Continuous integration and continuous deployment (CI/CD) is defined per service:
    - Services evolve at different rates.
    - You let the system evolve but set architectural principles to guide that evolution.
- High availability (HA) and clustering decisions are defined per service:
    - One size or scaling policy is not appropriate for all.
    - Not all services need to scale; others require autoscaling up to large numbers.

Comparing monolithic and microservices architectures
A side-by-side comparison shows the old way of architecting apps and the new way.



Category	Monolithic architecture	Microservices architecture
Architecture	Built as a single logical executable	Built as a suite of small services
Modularity	Based on language features	Based on business capabilities
Agility	Changes involve building and deploying a new version of the entire application	Changes can be applied to each service independently
Scaling	Entire application scaled when only one part is the bottleneck	Each service scaled independently when needed
Implementation	Typically entirely developed in one programming language	Each service can be developed in a different programming language
Maintainability	Large code base is intimidating to new developers	Smaller code bases easier to manage
Deployment	Complex deployments with maintenance windows and scheduled downtimes	Simple deployment as each service can be deployed individually, with minimal downtime

Emergence of microservices from modern tools and processes
So if this approach is such a good idea now, why weren't we doing this 10 or 20 years ago?
Many things have converged to make microservices make sense as an approach right now. Here are a few reasons; there are certainly more.

Ease and feasibility of distributing components

- Internet, intranet, or network maturity
- RESTful API conventions or perceived simplicity, and lightweight messaging

Ease and simplicity of cloud hosting

- Lightweight runtimes
    - Examples: Node.js and WebSphere App Server Liberty
- Simplified infrastructure
    - OS virtualization with hypervisors, containerization, infrastructure as a service (cloud infrastructure)
    - Workload orchestration (examples: Kubernetes or Mesos)
- Platform as a service:
    - Autoscaling, SLA management, messaging, caching, build management

Agile development methods, for example:

- Scrum, XP, TDD, IBM Cloud Garage Method, CI/CD
- Standardized code management in GitHub

Operational requirements for microservices
As much as we'd like to say microservices are great and only make things better, there's no free lunch; they have operational requirements too.

The biggest requirement is operational complexity because there are more moving parts to monitor and manage. Many enterprises have difficulty knowing simply how many apps they have running in production and what they are, much less knowing when one's stopped running. Now imagine that instead of trying to track dozens of apps, you're trying to track hundreds if not thousands of microservices. This is the full employment act for DevOps—you'd better get good at it!
Microservices value independence over reuse among services. (Reuse within a service is still prized.) Reuse creates dependencies, which makes it harder for teams to work independently; make sure the coordination friction is worth the reuse efficiency.
The more pieces running independently, the more the distributed system becomes an issue (network latency, disconnects, fault tolerance, serialization), the harder it is to find everything, and the harder it is to do end-to-end testing with confidence.
Yet, hold on! Istio provides the intelligent service mesh. More on that later.
Summary: Advantages of microservices
These are the advantages of microservices:

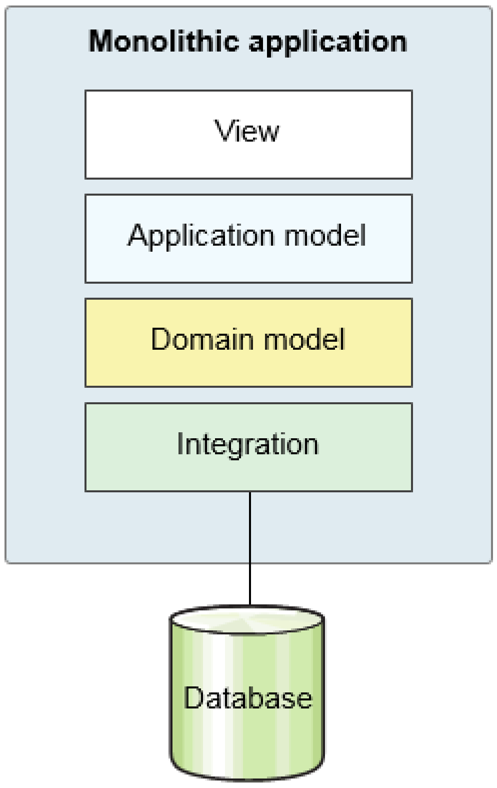
- Developed independently and has limited, explicit dependencies on other services
- Developed by a single small team in which all team members can understand the entire code base
- Developed on its own timetable so that new versions are delivered independently of other services
- Scales and fails independently, which isolates any problems
- Each can be developed in a different language
- Manages their own data to select the best technology and schema

MICROSERVICES ARCHITECTURE EVOLUTION: FROM SOA TO MICROSERVICES

Layered application architecture

So, this microservices stuff isn't how we architect apps. That's not how we do it around here! How did we get here? What can I learn from what I've done in the past with SOA and apply that to what I'll do in the future?

In the beginning (1980-2000 or so), there was the four-layer architecture although it was really three layers. The app model was a major innovation in the 1990s. We didn't even call it a monolith, but of course it was a monolith.



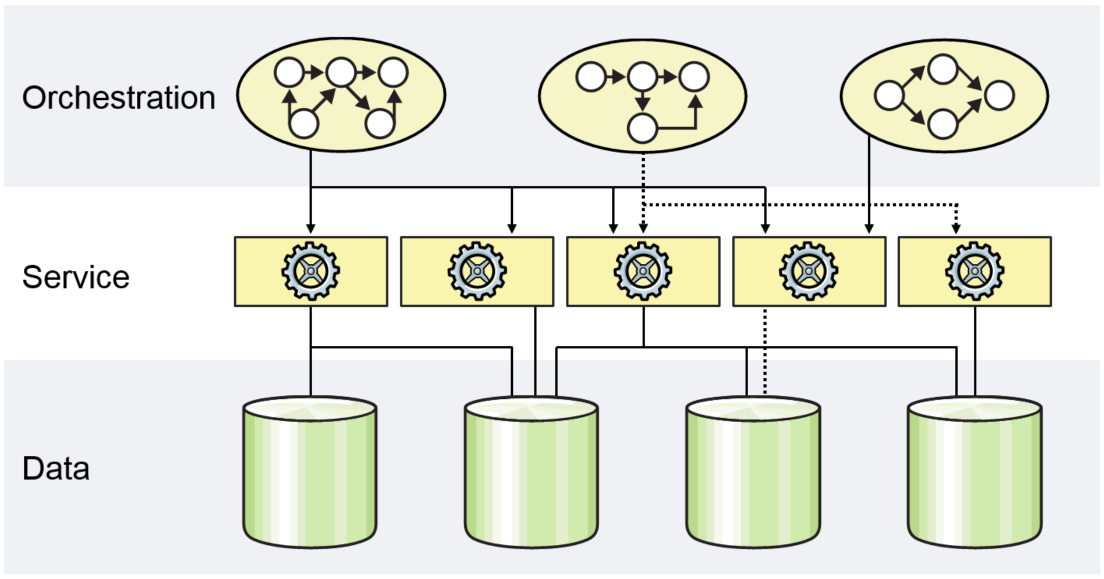
This model was sometimes deployed as tiers with a GUI tier (View and App Model) running in a separate process from a domain tier (Domain Model and Integration) running (of course) separately from the database server. There's a great debate as to whether these tiers should be deployed across different servers, or code should be more co-located. The separate tier approach led to Java EJBs with remote interfaces. When the clients were found not to be EJB clients but Web browsers, the move was to deploy the tiers in the same server and use local EJB interfaces.

Teams specialized by layers. GUI folks used AWT and Swing, and JSPs developed the views. Experts on the business requirements focused on the domain classes. The integration layer was developed by programmers who could get the object-relational frameworks and tools to work. Developers spent a lot of time in meetings deciding how the layers of code should work together. A change in one layer usually broke the ones above it. No one person could develop or maintain the whole app or even a vertical slice through the whole app.

SOA stack

With SOA, the layers were redesigned. The domain layer became a layer of services that encapsulated the back-end systems of record (SoRs) and databases of record. Individual services were units of work (transactions) that GUIs could use. But to get any real work done, the SOA stack had to string together sequences of services as business processes that were themselves macroservices.

Again, teams specialized. Some designed services; others implemented them to work with SoRs, and others specialized in BPM. No one could maintain a vertical slice.



Microservices and SOA

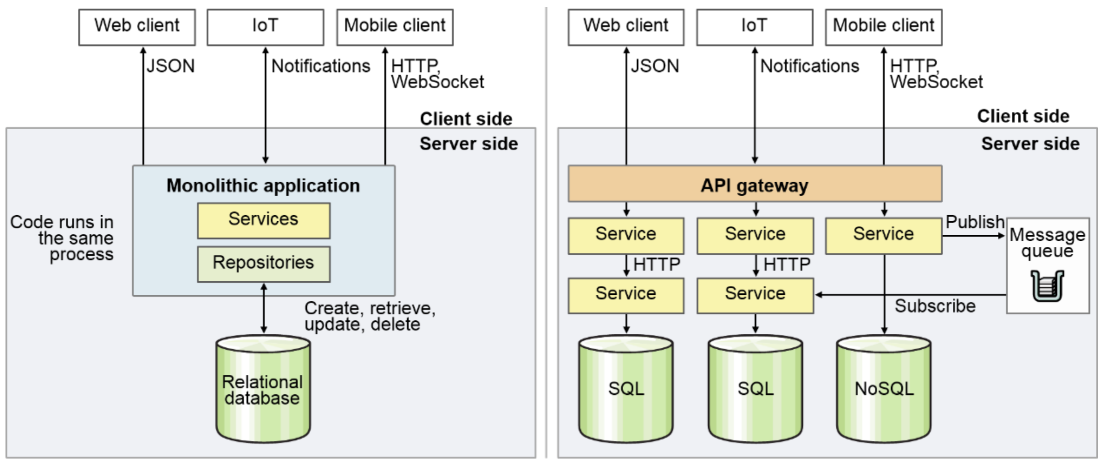
Isn't microservices just SOA by a different name? Microservices proponents like to say they're "SOA done right," but what does that mean? They're different but related: an evolution from SOA to microservices. Both SOA and microservices deal with a system of services that communicate over a network, but there are differences.

	Where's the focus	Results
SOA	On reuse	• This approach tends to align with a centrally funded model. • SOA services tend to be "servants of many masters". • This means that a change to a SOA service might impact multiple consumers.
Microservices	Breaking down monolithic apps into smaller, more manageable components	• The objective is more flexible, decoupled, faster development. • Challenges here relate to DevOps, management views, and controls.

For more information about whether microservices are an evolution of SOA or are completely different concepts, see Microservices vs SOA: How to start an argument.

Monolithic architecture versus microservices architecture

The following diagram shows the architecture of a monolithic app on the left compared to how such an app can become a set of microservices as shown on the right.



Microservices and IBM Cloud Kubernetes Service

Going from a monolithic architecture to a microservice architecture is not as hard as you might think. Many large organizations face existing apps today that need to make a transition to cloud-friendly tools. For example, you can extend functionality to a monolithic app by adding cloud services. You can also start using operational tools, such as Grafana for monitoring, Istio for a service mesh, and IBM Cloud App ID for user authentication.

So how does this evolution work? With IBM Cloud Kubernetes Service, you repackage an existing app, even monolithic ones into containers, for example:

1. Extract an existing Java app and prepare it for WebSphere Liberty on the cloud.
2. Set up Liberty and the database as microservices in Kubernetes.
3. Containerize the app as-is and deploy it to a cluster.
4. Add on a cloud service, such as AI.
5. Use cloud operation tools to monitor the app.

Consider these other aspects of a containers-based approach:

- **Native CI/CD tools for fast roll-outs**: Kubernetes orchestration, native Kubernetes rollbacks, Istio canary testing, and Helm's repeatable deployments
- **Flexible scaling infrastructure**: when a single data center and a single app instance isn't enough
- **Simplified management**: when infrastructure is stood up and managed for you, such as a Kubernetes master is managed, IaaS is managed, automated health monitoring and recovery for nodes
- **Built-in security**: IBM Vulnerability Advisor + X-Force Exchange for insights; tightly controlled network traffic; isolation for master, nodes, and single tenant, encryption; and TLS certificates in IBM Certificate Manager

Microservices architecture mapped to the SOA stack

Let's take a look at microservices application architecture. We'll compare it to the previous architectures described in this course.

Previous architectures (four layer and SOA stack) are not gone completely.

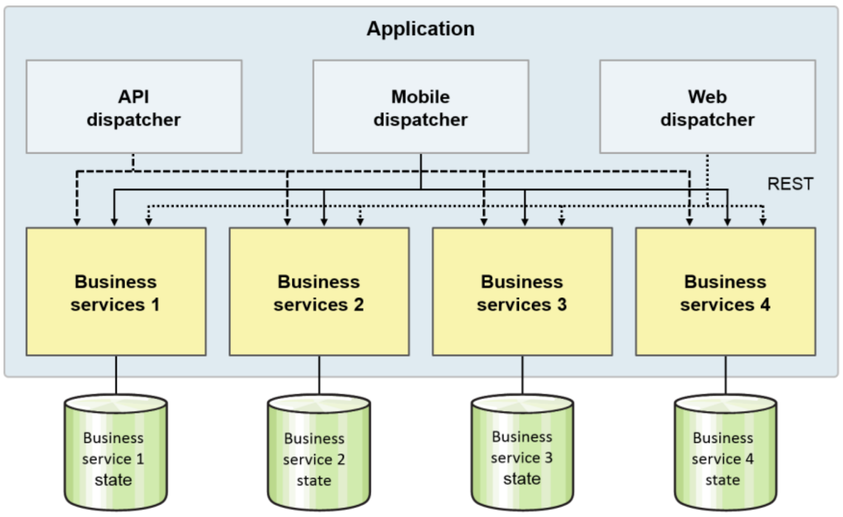
The business services tend to do what the domain model and integration layers used to do and so are still implemented that way. This is also what the SOA services used to do (or were supposed to). It's just that these layers don't span the whole app; they're encapsulated within each service.

The dispatchers do what the app model used to do: one-stop shopping for the client so that all of the business services together look like a single app that does exactly what the client needs. But now you have an app model for each client type, not a single app model layer that spans the entire app.

What happened to the view layer? It moved into the clients where it belonged all along. Either the client is a mobile app, which is a view with perhaps a bit of its own app model; or it's a partner app that's more than just a view but that has its own views; or it's a web app. Web apps used to contain a lot of view code to render the HTML and needed state from the view in an HTTP session, but no more. With modern techniques using HTML 5 and CSS 3, the web browser uses static files that are downloaded from the website to do all that rendering and store the session state.

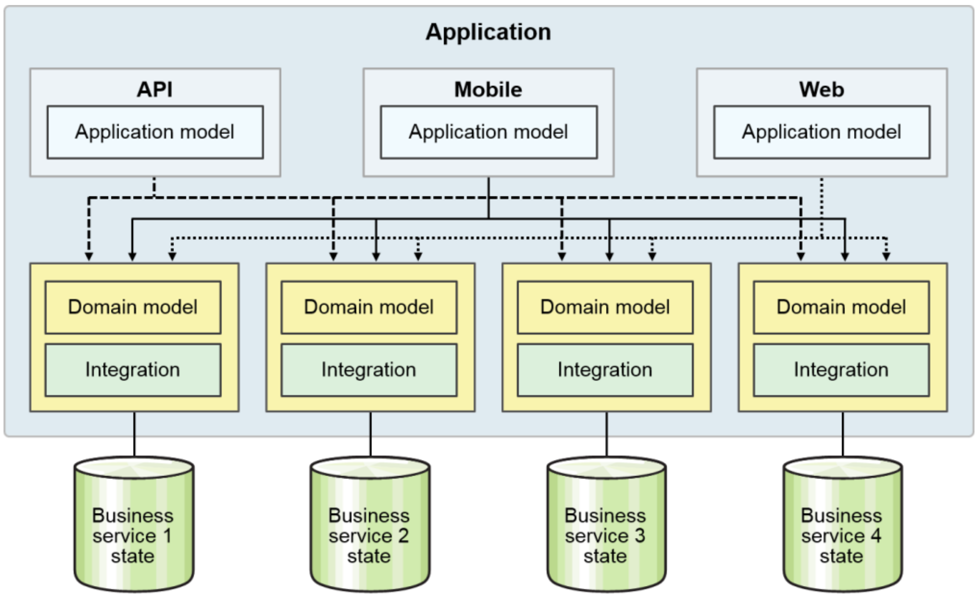
Example airline architecture

This diagram demonstrates how you can functionally decompose a monolithic application to use individual microservices.



Microservices layers

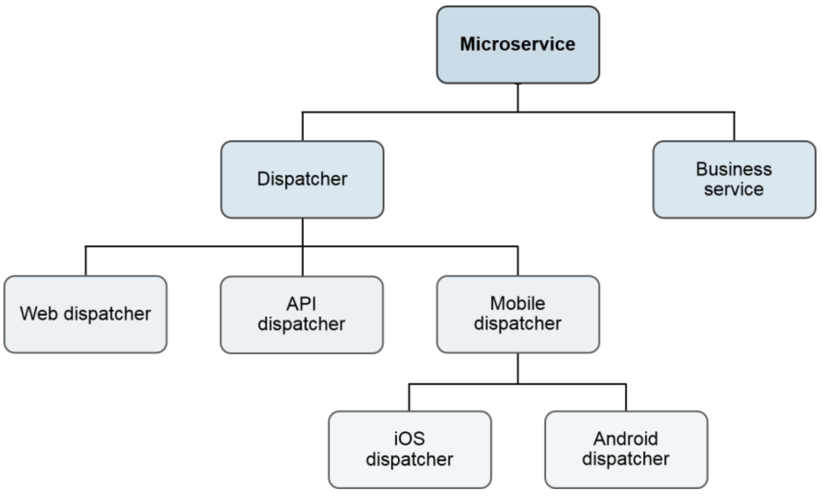
Each service is a smaller half-monolith.



Microservices types hierarchy

All business services are pretty much the same architecturally, but there are subtype specializations of dispatchers. This doesn't show a class or inheritance hierarchy; it shows specialization.

How many of these you implement for an app depends on what types of clients the app supports and how specialized those clients require their dispatchers to be. One mobile dispatcher might support all devices, or you might need one for Blackberry. One iOS dispatcher might be insufficient, or you might need separate ones for iPhones versus iPads. One web dispatcher might be insufficient, or you might need ones for HTML 5 browsers versus older ones.



Language decisions

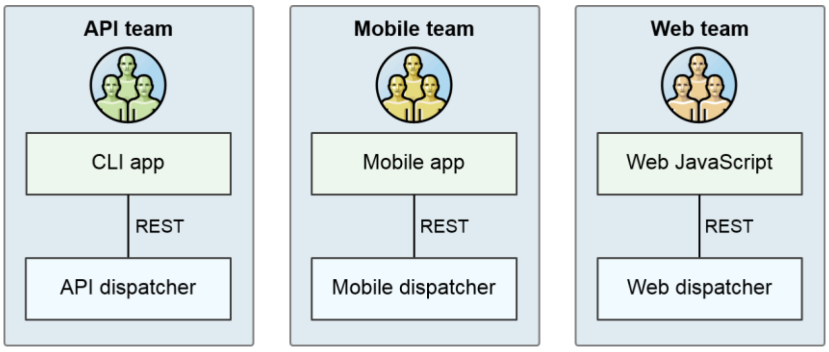
Each microservice can be developed in any language that you want as long as it can be deployed on the cloud and supports REST APIs, but here are some trends.

Dispatchers are often implemented in Node. Node is good at handling large numbers of clients and lots of concurrent I/O. For clients that are running JavaScript, Node on the server can be an easier fit.

Business services are often written in Java. Java handles CPU-intensive tasks well and is good at connecting to external systems, especially with managed pools of shared connections. Many enterprises have lots of Java programmers on staff who have experience writing SOA services along these lines, so put them to work developing similar business logic as microservices.

Backend for frontend (BFF)

Typically, rather than have one team write all or multiple dispatchers while someone else implements the clients, instead put the same team in charge of a dispatcher/client pair. The two need to be designed for each other, so have the communication take place within a team, not between teams. The skills tend to be different for different types of clients, so let the teams specialize.



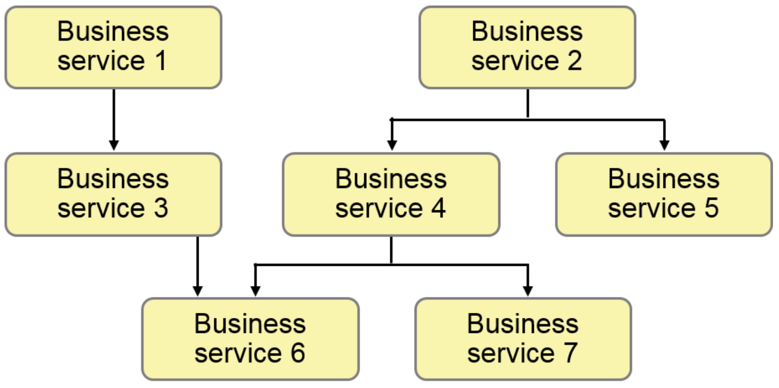
You can read more in [Pattern: Backends For Frontends]() by Sam Newman.

Business service microservices dependencies: Typical

Business services can depend on other business services. Just make sure each service is a complete business task. Resist the urge to separate out the database access layer into its own service.

If services are each a complete business task, having business services collaborate is fine and can be helpful.

Notice the dependencies shown here. This is an acyclic directed graph. This means that if service A depends on service B, then B does not also depend on A. These types of dependencies are easier to manage.

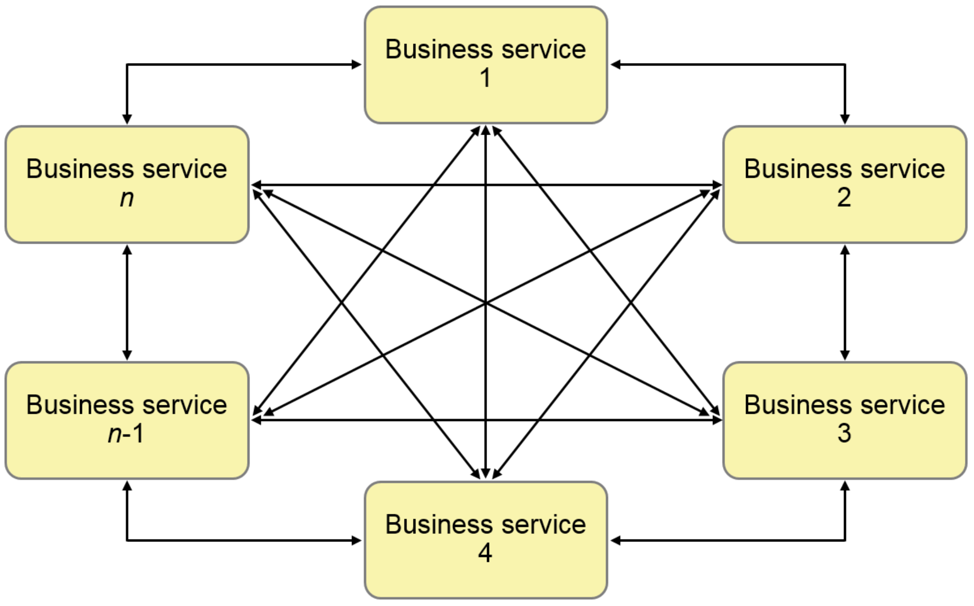


Business service microservices dependencies: Death Star

This is a “death star” of dependencies among microservices. Everything depends on everything else. Search for microservices death star on the Internet, and you’ll find pictures and descriptions of apps with hundreds of services all drawn in a circle with lines connecting them all to each other. Yikes!

How do you deploy such an architecture? Which one do you deploy first? If you have to change the API on one (a new version), which other ones do you need to change?

Avoid letting this Death Star topology happen to your architecture.



Communication among services

Now that you’ve got a bunch of microservices, how do you connect them to each other?

Communication among services should be language-neutral. Different microservices can be implemented in different languages (now or in the future), so don’t lock yourself into a language-specific integration technology, for example, Java sockets or even CORBA/IIOP.

The standard these days is REST and really JSON/REST, so use that. For asynchronous integration, follow an open, cloud-friendly standard like IBM Message Hub with Apache Kafka. Other examples of asynchronous protocols are AMQP, MQ Light, RabbitMQ.

Resist the urge to use other approaches like XML or serialization. That leads to a combinatorial explosion of protocols that each API provider and consumer must support.

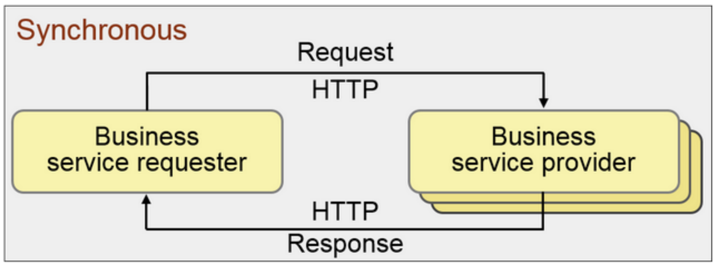
With IBM Cloud Kubernetes Service, microservice service communication goes over virtual local area networks (VLANs). By default, IBM Cloud Kubernetes Service provides safe and free-flowing communication that you can change for inbound and outbound communication from the worker nodes.

A Kubernetes service is a group of pods and provides network connections to these pods for other microservices in the cluster without exposing the actual private IP address of each pod. Similarly, you can choose to keep microservices private and use the built-in security features.

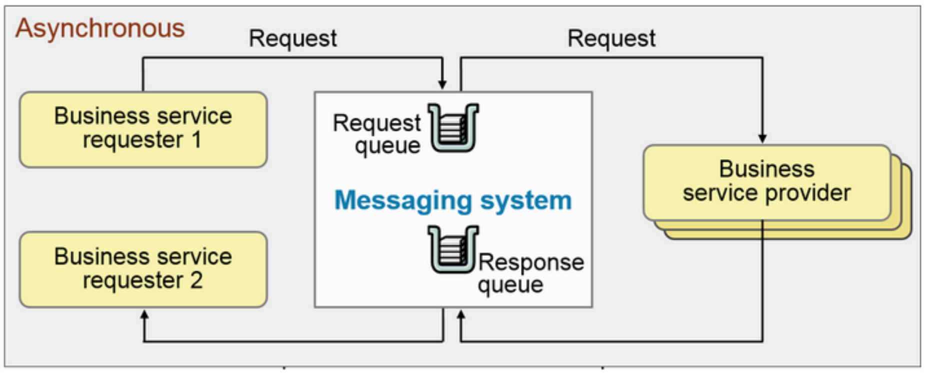
Synchronous versus asynchronous communication

Synchronous versus asynchronous communication

Synchronous REST is usually easier to get something working, so start with that. With synchronous, the whole loop (requester, provider, messages) must keep working through the whole lifetime of the invocation.



Then, consider strategically converting some integration points to asynchronous, either because of the nature of the request (long running, runs in the background) or to make the integration more reliable and robust. With asynchronous, the invocation is broken into 3-4 parts. If any one fails, the system can probably retry it for you. And because the requester is stateless, the instance that receives the response doesn't even need to be the instance that sent the request.

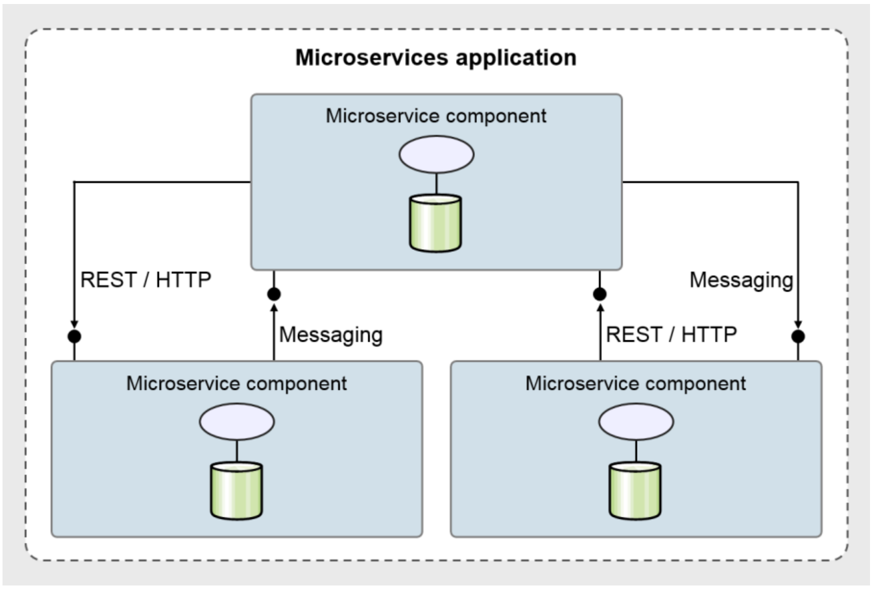

Asynchronous
Request
Request
Business service requester 1
Business service requester 2
Request queue
Messaging system
Response queue
Business service provider

Asynchronous communication can make microservices more robust:

- The requester doesn't have to block while the provider runs.
- A different requester instance can handle the response.
- The messaging system holds the action and result.

Microservices intercommunication

You can mix and match synchronous and asynchronous communication. Usually a request/response invocation is either all synchronous or all asynchronous, but this shows that a single invocation can be sync in one direction and asynchronous in the other.



You can use lightweight protocols:

- REST such as JSON or HTTP
- Messaging such as Kafka

The aim is complete decoupling, which is achieved by these methods:

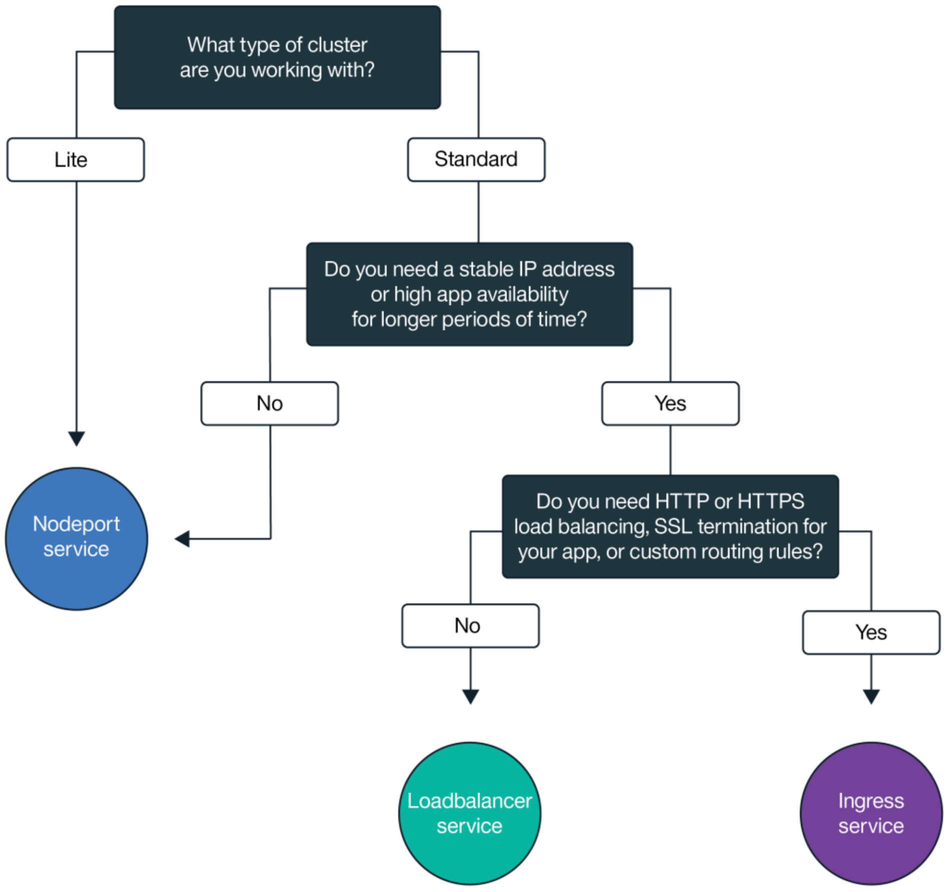
- Messaging wherever possible
- Service registry or discovery
- Load balancing
- Circuit breaker patterns

Microservices communication in IBM Cloud Kubernetes Service

Now that you've explored how microservices communicate with each other, you'll look at how they communicate as an app within a Kubernetes cluster.

When you create a Kubernetes cluster in IBM Cloud Kubernetes Service, every cluster must be connected to a public VLAN. The public VLAN determines the public IP address that is assigned to a worker node during cluster creation.

The public network interface for the worker nodes is protected by Calico network policies. These policies block most inbound traffic by default. However, inbound traffic that is necessary for Kubernetes to function is allowed, as are connections to NodePort, Loadbalancer, and Ingress services. NodePort is an easy way to start with simple public access to apps. As you progress to more sophisticated networking, the LoadBalancer gives you some portable public and private IP addresses. And when you grow to a larger microservices implementation, Ingress provides routing for multiple apps across HTTPS, TCP, or UDP load balancer. The following decision tree can guide your public access decisions:

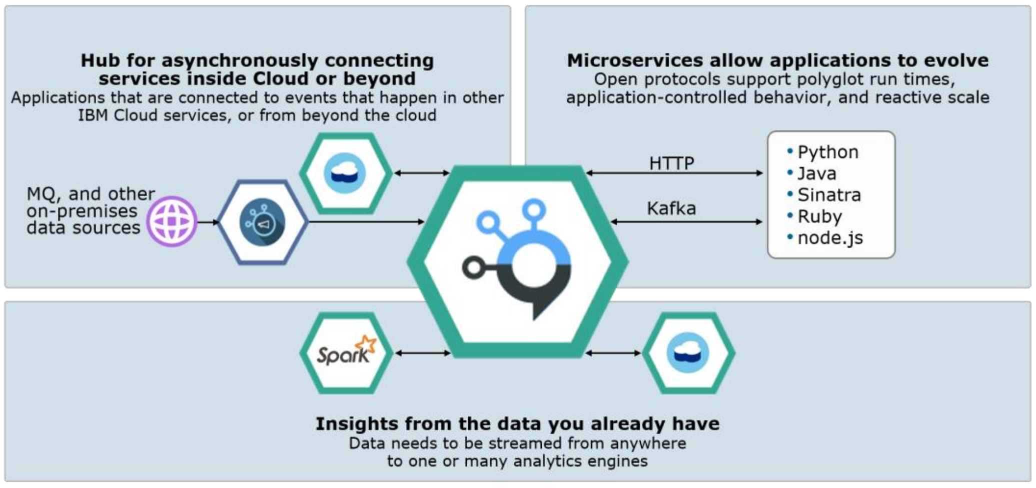

What type of cluster are you working with?
Lite
Standard
Do you need a stable IP address or high app availability for longer periods of time?
No
Yes
Do you need HTTP or HTTPS load balancing, SSL termination for your app, or custom routing rules?
No
Yes
Nodeport service
Loadbalancer service
Ingress service

For more information about these policies, including how to modify them, see [Network policies].

IBM Message Hub service

IBM Cloud has a messaging system service called IBM Message Hub. It’s based on the [Apache Kafka project], which is rapidly becoming a standard for high-volume messaging on cloud platforms.

It’s used to integrate lots of services in cloud, but you can also use it to integrate your apps and microservices running in IBM Cloud. It supports two APIs: a simple HTTP REST API and a more complex but more powerful Kafka API.


Hub for asynchronously connecting services inside Cloud or beyond
Applications that are connected to events that happen in other IBM Cloud services, or from beyond the cloud
MQ, and other on-premises data sources
Microservices allow applications to evolve
Open protocols support polyglot run times, application-controlled behavior, and reactive scale
HTTP
Kafka
Python
Java
Sinatra
Ruby
node.js
Spark
Insights from the data you already have
Data needs to be streamed from anywhere to one or many analytics engines

Comparison of operations for monolithic and microservice architectures

Service meshes can assist you with microservice implementations:

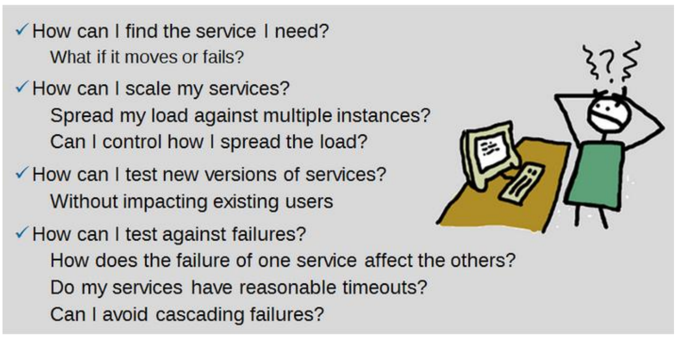
- How can I find the service I need?
- How can I scale my services?
- How can I test new versions of services without impacting new users?
- How can I test against failures?
- How can I secure service-to-service communication?
- How can I route traffic in a specific way?
- How can I test circuit breaking or fault injection?
- How can I monitor my microservices and collect metrics?
- How can I do tracing?
- Most importantly, how can I do all these things without changing individual microservices application code?

In traditional, monolithic apps, the locations of resources are well known, relatively static, and found in configuration files (or hard coded). Even if a monolithic app has been divided into different tiers, such as for the user interface, app logic, database access, and others, the location of resources used by the app tend to be well defined and static. Often times, all tiers of the app are hosted in the same geographical location even if they are replicated for availability. Connections among the different parts of the app tend to be well defined, and they don’t tend to change very much.

However, this is not the case for a microservices implementation. Whether that app is built from scratch or is gradually being built by breaking off functions of a monolithic app, the location of a given service could be anywhere. The microservice could be in a corporate data center, a public cloud provider, or some combination. It could be hosted on bare metal servers, virtual machines, containers, or all these.

A given service will usually have more than one instance or copy. Instances of a service can come and go as that service scales up and down under different load conditions. Instances can fail and be replaced by other instances hosted from the same physical location or other locations. You can also have certain instances hosting one version of a service and others hosting a different version. As you are using these services in your app, the last known location of the service might not be where it is now.

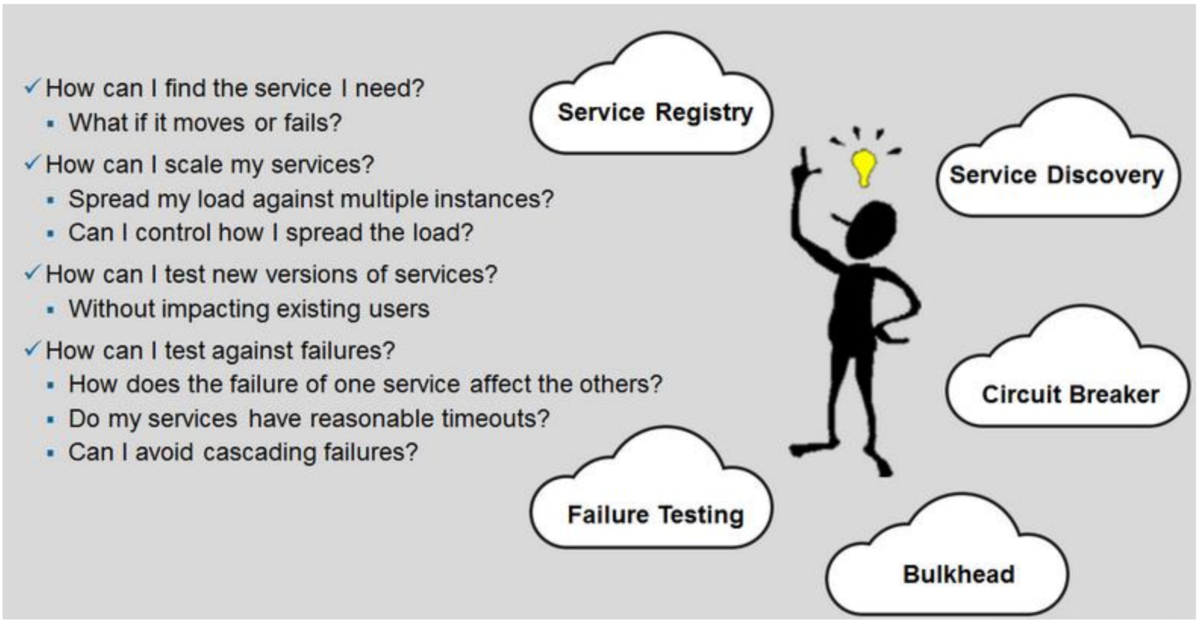
The following questions reflect issues you encounter with microservices:



- ✓ How can I find the service I need?
  - What if it moves or fails?
- ✓ How can I scale my services?
  - Spread my load against multiple instances?
  - Can I control how I spread the load?
- ✓ How can I test new versions of services?
  - Without impacting existing users
- ✓ How can I test against failures?
  - How does the failure of one service affect the others?
  - Do my services have reasonable timeouts?
  - Can I avoid cascading failures?

A service mesh can help

A service mesh can help you solve the issues that arise when you’re implementing a microservices app. Here is the list of issues shown earlier. Each issue is matched with a generic function, typically provided by a service mesh that can address the issue. The next several sections in this course will provide more detailed description of each of these generic functions, but here is a brief introduction.

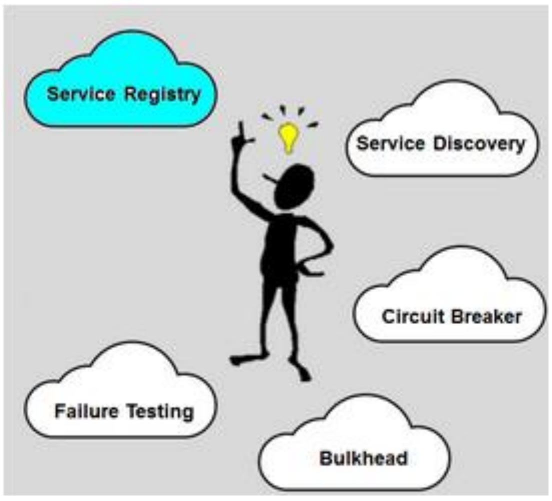


- ✓ How can I find the service I need?
  - What if it moves or fails?
- ✓ How can I scale my services?
  - Spread my load against multiple instances?
  - Can I control how I spread the load?
- ✓ How can I test new versions of services?
  - Without impacting existing users
- ✓ How can I test against failures?
  - How does the failure of one service affect the others?
  - Do my services have reasonable timeouts?
  - Can I avoid cascading failures?

A **service registry** is used to keep track of the location and health of services, so a requesting service can be directed to a provider service in a reliable way. Working hand in hand with the service registry is the function of service discovery. **Service discovery** uses the service registry's list of available services and instances of those services, and it directs requests to the appropriate instance based on pure load balancing or any other rule that has been configured.

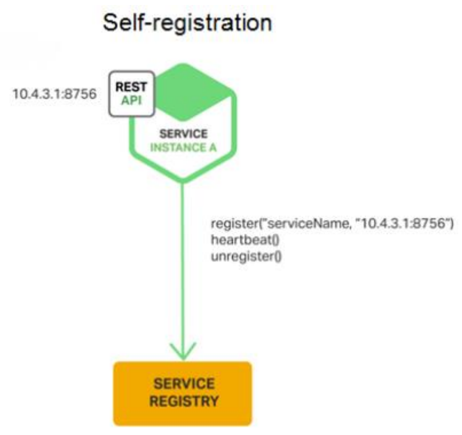
Some service meshes provide automated **testing** facilities. This gives you the ability to simulate the failure or temporary unavailability of one or more services in your app, and to make sure the resiliency features that you designed will work to keep the app functional. Two functions that can help to provide this resiliency are **circuit breakers** and **bulkheads**. Open-source mesh functions can be used to solve many of the challenges of implementing microservices.

Service registry



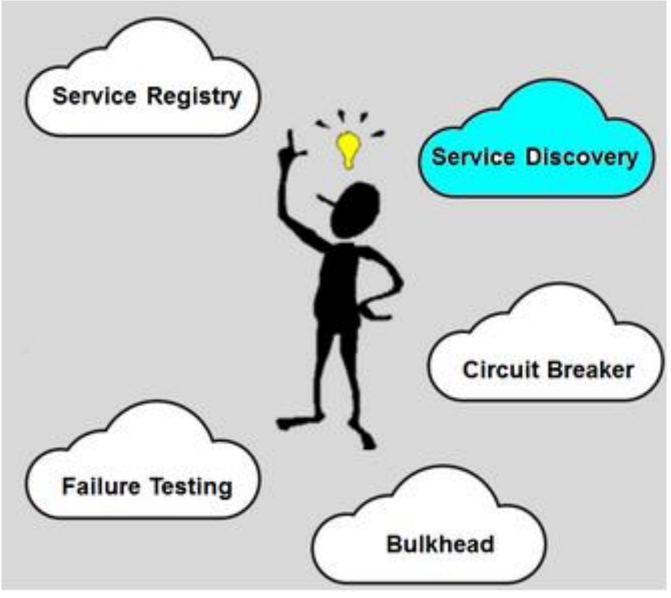
Every service mesh will have a service registry. The service registry is a simple key-value pair data store, maintaining a current list of all working service instances and their locations. As each service instance starts, it will register itself with the service registry, either through its own client code or through a third-party registrar function. For instance, your service instances might be implemented in containers that use the Kubernetes container cluster manager. Kubernetes provides a third-party registrar service for those containers in its clusters.

After a service instance has registered its existence with the service registry, the service registry will use a heartbeat mechanism, either in-bound or out-bound, to keep its list of instances current.



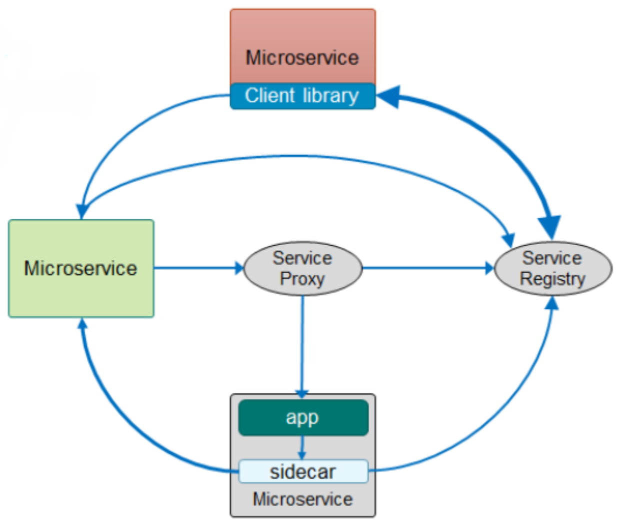
Because the service registry is so critical to the implementation, it needs to be made highly available in some way.

Service discovery and service proxy



The heart of any service mesh is the service registry and service discovery functions. These are the functions that enable each service to find and connect with the other services it needs to do its work. The source of all knowledge is the service registry. The service registry maintains a list of all service instances and their locations. As each service instance starts, it checks in with the service registry, providing its name and location.

After this initial check-in, each service registry implementation will have some kind of heartbeat mechanism to keep its list current, removing any instances it can no longer contact. Depending on the overall design of the app, each service instance can be either a service client, requesting data or functions from other services, a service provider, providing data or functions to service clients, or both. Whenever a service needs another service, it uses the service discovery function to find an instance of that service.



Client-side discovery

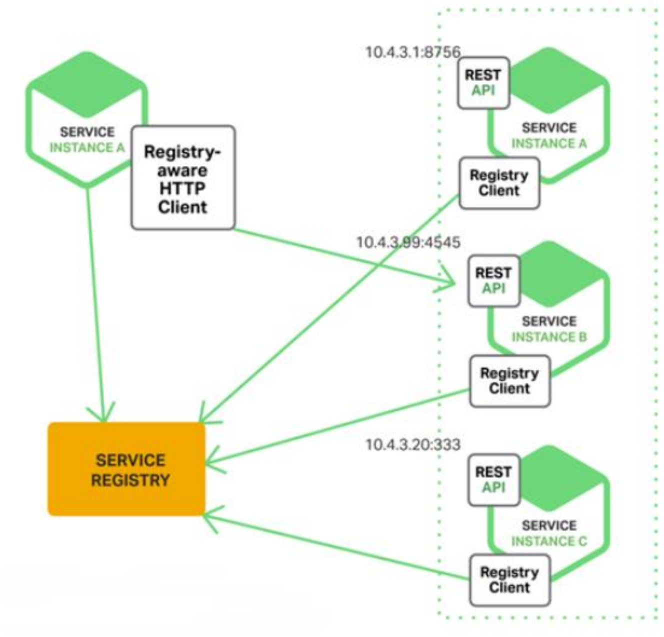
The first type of service discovery is client-side discovery. Using this method, the requesting service, or service client, works directly with the service registry to determine all the available instances of the service it needs and is also responsible for deciding which of those service instances it will use for its request.

As mentioned previously, you can use several possible rules to help you decide which instance to use.

You might use round-robin load balancing where you spread each request evenly across all the instances, which sends each request to the next instance in the list. You might also use a different kind of rule, having certain clients access certain provider instances, to test new versions or for some other reason.

In any case, when you use client-side discovery, the client service is responsible for making the decision and following the decision to contact the appropriate service instance. The client will use some client code to make that happen.

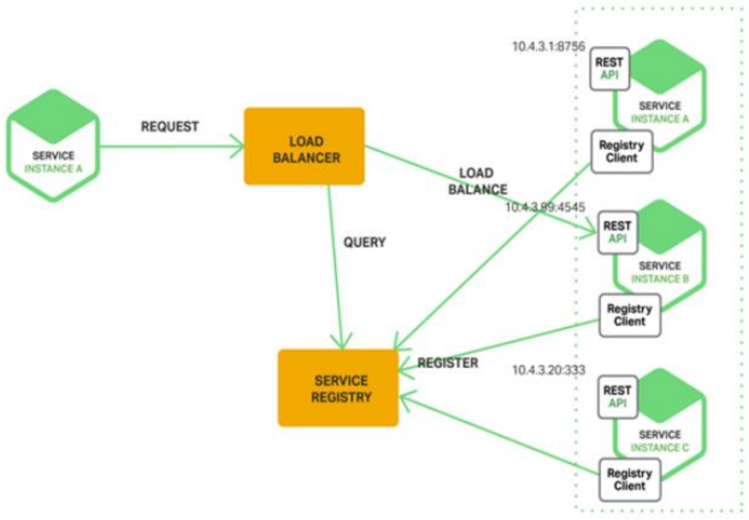
In the following illustration, the requesting service instance A on the left has discovered from the service registry that there are three instances of the service it needs. Using the rule it has chosen, the requesting service has determined to contact service instance B on the right for its request.



Server-side discovery

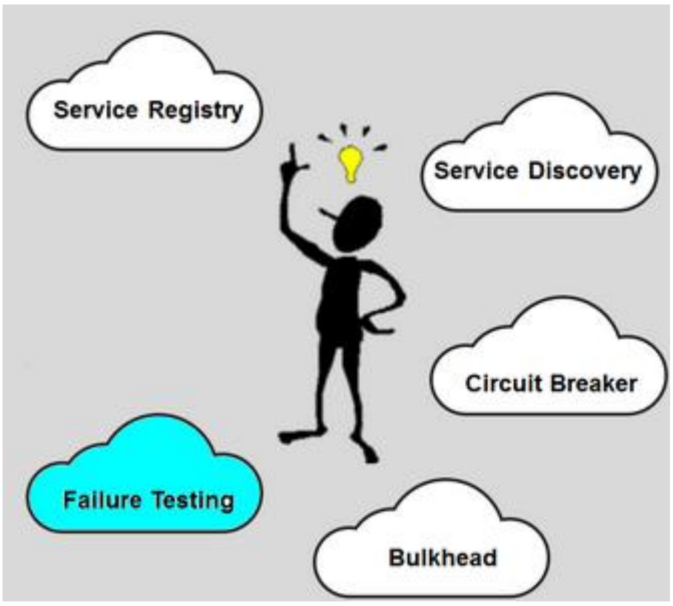
The second type of service discovery is called server-side discovery. This method introduces another component, labelled the load balancer in the following illustration. It is commonly called the service proxy in microservices discussions.

The task of querying the service registry and directing a request to the appropriate service instance is delegated to the load balancer. In some implementations, such as Istio, the load balancer can be programmed with specific rules to govern its choice of service instance, beyond just using round-robin load balancing. The nginx load balancer is built into several service meshes you might read about in the marketplace.



A major benefit of this load-balancing approach is that you need only the function of querying the service registry and directing requests to service instances implemented one time in the load balancer. A possible drawback is that the load balancer is one more component to manage. In most cases, this is not a problem because the load balancer is provided by the service mesh.

Automated testing



One of the premises of cloud native and microservice architectures is that they should be designed to run on unreliable systems. You should expect components to fail and build resiliency into your apps such that they can withstand those failures. One of the basic design points of this strategy is to always have more than one instance of any given service so that your app can survive the failure of a given instance and continue to function while that instance is restarted.

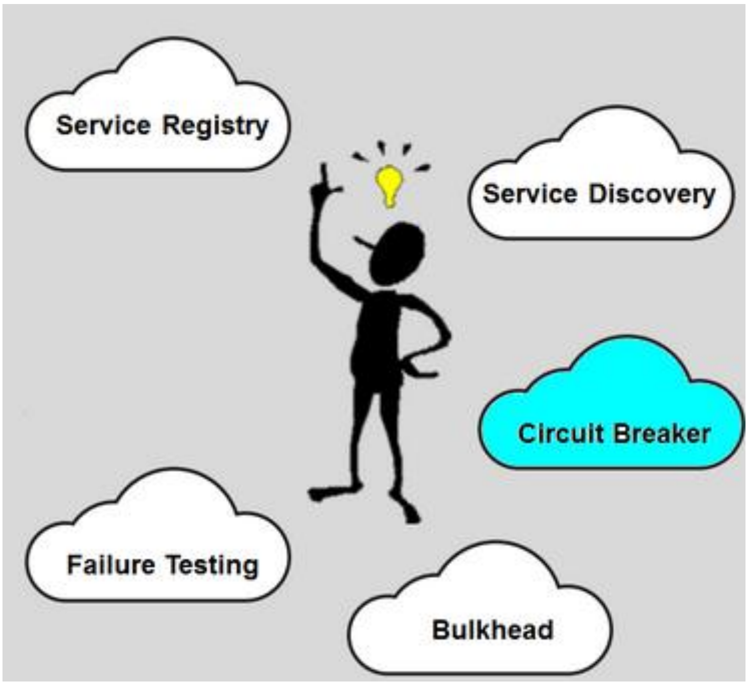
Automated testing tools are useful to ensure the resiliency of your microservices app. Examples of these tools include Netflix Chaos Monkey, the Netflix Simian Army, and the features built into Istio.

With IBM Cloud Kubernetes Service, you can manage the rollout of your changes in an automated and controlled fashion. If your rollout tests aren't going well, you can roll back your deployment to the previous revision.

This flow shows how you use Kubernetes to easily manage app problems during testing:

1. Before you begin, create a deployment.
2. Roll out a change to a microservice. For example, you might want to change the image that you used in your initial deployment. When you deploy microservices with Kubernetes, the change is immediately applied and logged in the roll-out history.
3. Test the status of your deployment with your operational and testing tools.
4. View the rollout history for the deployment and identify the revision number of your last deployment.
5. When ready, roll back to the previous version or specify a revision.

Circuit breaker

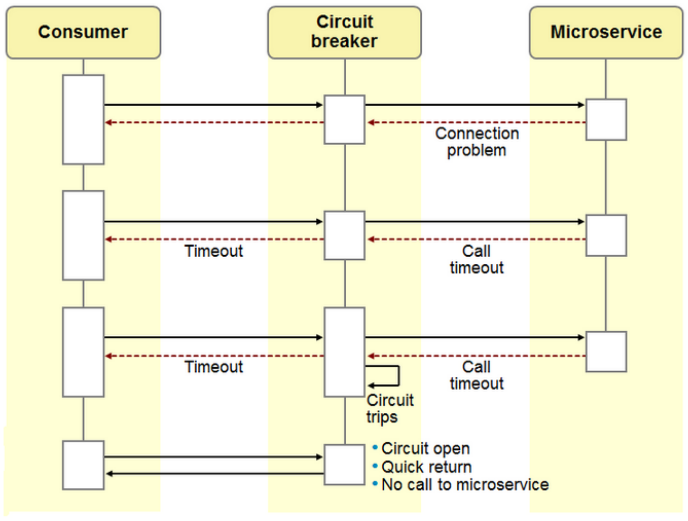


The circuit breaker pattern is a programming tool included with Istio and Netflix Hystrix. Circuit breakers work to prevent response delays from a given service dependency caused by failure or latency, so that delays don't cause broader latencies and failures throughout the app. The breaker allows you to set timeout thresholds and failure thresholds that allow you to fail fast from such situations with the expectation of building an alternative plan into your app.

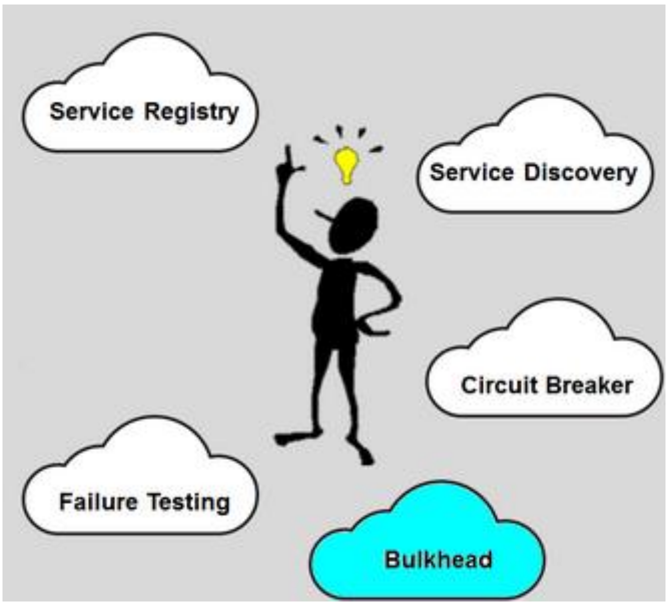
The circuit breaker works as shown in the diagram below. You can define threshold numbers for timeout and timed out requests. As you make service calls through Hystrix and the thresholds are exceeded, the circuit trips and subsequent requests fail immediately. This allows you to follow the alternative plan that you should have built in to your app.

You also configure a value for these situations:

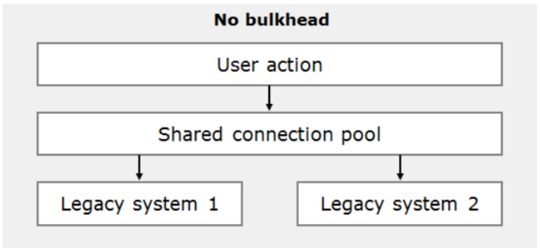
- How long a recovery time you allow before opening the circuit again
- Trying a call to the service again
- Testing your thresholds

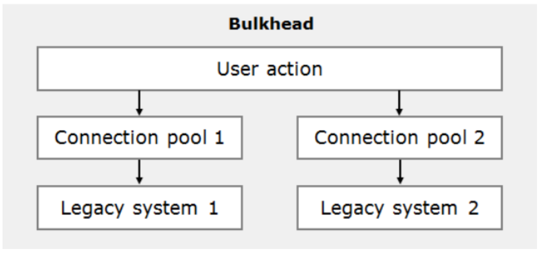


Bulkhead



The Bulkhead pattern is another tool included with Netflix Hystrix that prevents response delays from a given service call from causing wider problems throughout the app. This behavior is accomplished by using separate thread pools for connection attempts to other systems so that delays called by latency or failure of a given service are isolated to that service and do not use all the threads for the app as more requests are made. By using separate connection pools for each dependent service, failure on any service is isolated to that service and does not cause broader issues in the app.





Istio open platform service mesh

Istio is an open platform service mesh implementation for connecting, managing, and securing microservices. Istio provides an easy way to create a network of deployed services with load balancing, service-to-service authentication, monitoring, and more without requiring any changes in service code. You add Istio support to services by deploying a special sidecar proxy throughout your environment that intercepts all network communication among microservices, which is configured and managed using Istio's control plane functionality.

History of Istio

The release of Istio was a result of collaboration between IBM, Google, and Lyft to provide traffic flow management, access policy enforcement, and telemetry data aggregation between microservices. All those are achieved without requiring any changes to the application code. Thus, developers can focus on business logic and quickly integrate new features.

Before combining forces, IBM, Google, and Lyft had been addressing separate but complementary pieces of the problem.

- IBM's Amalgam8 project is a unified service mesh that provides a traffic routing fabric with a programmable control plane to help internal and enterprise customers with A/B testing, canary releases, and to systematically test the resilience of services against failures.
- Google's Service Control provides a service mesh with a control plane that focuses on enforcing policies such as ACLs, rate limits, and authentication in addition to gathering telemetry data from various services and proxies.
- Lyft developed the Envoy proxy to aid its microservices journey, which brought the company from a monolithic app to a production system spanning 10,000 or more VMs handling 100 or more microservices.

Why use Istio?

Istio addresses many of the challenges faced by developers and operators as monolithic applications transition towards a distributed microservice architecture. As a service mesh grows in size and complexity, it can become harder to understand and manage. Its requirements can include discovery, load balancing, failure recovery, metrics, and monitoring, and often more complex operational requirements such as A/B testing, canary releases, rate limiting, access control, and end-to-end authentication.

Istio provides a complete solution to satisfy the diverse requirements of microservice applications by providing behavioral insights and operational control over the service mesh as a whole. It provides several key capabilities uniformly across a network of services:

- Traffic management: Control the flow of traffic and API calls between services, make calls more reliable, and make the network more robust in the face of adverse conditions.
- Observability: Gain understanding of the dependencies between services and the nature and flow of traffic between them, providing the ability to quickly identify issues.
- Policy enforcement: Apply organizational policies to the interaction between services and ensure access policies are enforced and resources are fairly distributed among consumers. Policy changes are made by configuring the mesh, not by changing application code.
- Service identity and security: Provide services in the mesh with a verifiable identity and provide the ability to protect service traffic as it flows over networks of varying degrees of trustability.

In addition to these behaviors, Istio is designed for extensibility to meet diverse deployment needs:

- Platform support: Istio can run in a variety of environments including ones that span cloud, on-premises, Kubernetes, Mesos, and so on.
- Integration and customization. The policy enforcement component can be extended and customized to integrate with existing solutions for ACLs, logging, monitoring, quotas, auditing, and more.

These capabilities greatly decrease the coupling between application code, the underlying platform, and the policy. This decreased coupling not only makes services easier to implement, but also makes it simpler for operators to move application deployments between environments or to new policy schemes. Applications become inherently more portable as a result.

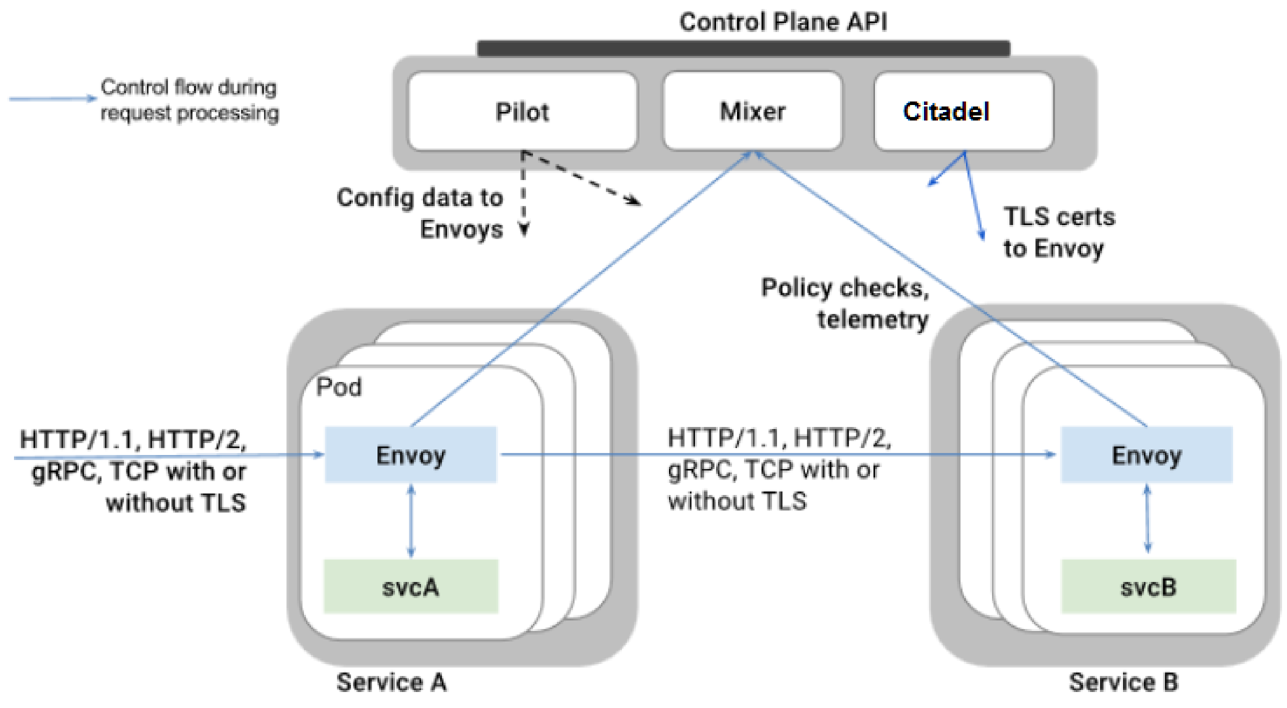
How Istio works

An Istio service mesh is logically split into a data plane and a control plane.

- The data plane is composed of a set of intelligent proxies (Envoy) deployed as sidecars that mediate and control all network communication among microservices.

- The control plane is responsible for managing and configuring proxies to route traffic and enforcing policies at runtime.

The following diagram shows the different components in each plane:



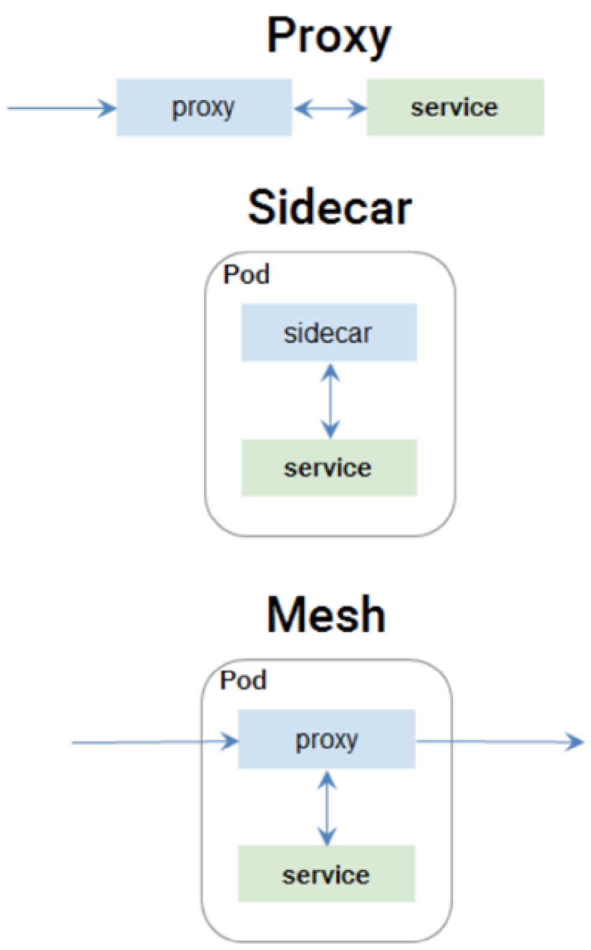
You'll hear the term sidecar when discussing Istio. You'll see how Istio relies on sidecars as an important component of its architecture. Let's first understand what sidecar is.

Sidecar proxy

A service mesh implementation, such as Istio, uses proxies that are usually deployed as sidecars in pods. A proxy controls access to another object. Istio uses proxies between services and clients. It enables the service mesh to manage interactions.

As a point of review, Kubernetes deploys containers into pods and all the containers in a pod are hosted on the same node. A pod can manage the network access to the containers in the pod. A pod typically hosts a single container, but can host multiple pods as a unit.

A sidecar adds behavior to a container without changing it. In that sense, the sidecar and the service behave as a single enhanced unit. The pods host the sidecar and service as a single unit.



Envoy

Istio uses an extended version of the Envoy proxy, a high-performance proxy developed in C++, to mediate all inbound and outbound traffic for all services in the service mesh. Istio uses Envoy's many built-in features such as dynamic service discovery, load balancing, TLS termination, HTTP/2 and gRPC proxying, circuit breakers, health checks, staged rollouts with %-based traffic split, fault injection, and rich metrics.

Envoy is deployed as a sidecar to the relevant service in the same Kubernetes pod. This allows Istio to extract a wealth of signals about traffic behavior as attributes, which in turn it can use in Mixer to enforce policy decisions and be sent to monitoring systems to provide information about the behavior of the entire mesh. The sidecar proxy model also allows you to add Istio capabilities to an existing deployment with no need to rearchitect or rewrite code. You can read more about why this approach was chosen in Design Goals.

Mixer

Mixer is a platform-independent component responsible for enforcing access control and usage policies across the service mesh and collecting telemetry data from the Envoy proxy and other services. The proxy extracts request level attributes, which are sent to Mixer for evaluation. For more information on this attribute extraction and policy evaluation, see Mixer Configuration. Mixer includes a flexible plugin model that enables it to interface with a variety of host environments and infrastructure back ends, which abstracts the Envoy proxy and Istio-managed services from these details.

Pilot

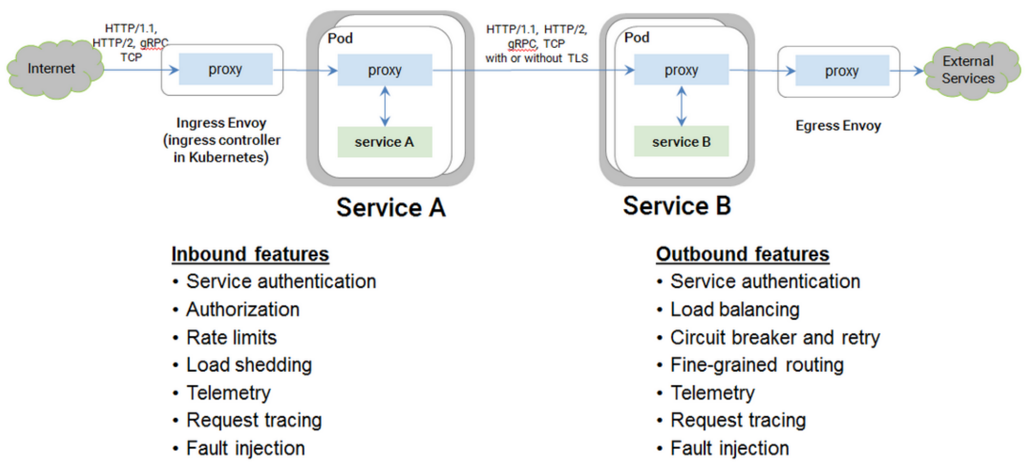
Pilot provides service discovery for the Envoy sidecars, traffic management capabilities for intelligent routing (for example, A/B tests and canary deployments), and resiliency (timeouts, retries, circuit breakers, and more). It converts a high-level routing rules that control traffic behavior into Envoy-specific configurations and propagates them to the sidecars at runtime. Pilot abstracts platform-specific service discovery mechanisms and synthesizes them into a standard format consumable by any sidecar that conforms to the Envoy data plane APIs. This loose coupling allows Istio to run on multiple environments (for example, Kubernetes, Consul/Nomad) while maintaining the same operator interface for traffic management.

Citadel

Citadel (previously called *Istio Auth*) provides strong service-to-service and end-user authentication using mutual TLS, with built-in identity and credential management. It can be used to upgrade unencrypted traffic in the service mesh and provides operators the ability to enforce policy that is based on service identity rather than network controls. Future releases of Istio will add fine-grained access control and auditing to control and monitor who accesses your service, API, or resource by using a variety of access control mechanisms, including attribute and role-based access control and authorization hooks.

Istio mesh request flow

The flow shown here is generic to any service mesh but can be considered for Istio.


Internet
HTTP/1.1, HTTP/2, gRPC TCP
proxy
Ingress Envoy (ingress controller in Kubernetes)
Pod
proxy
service A
Service A
HTTP/1.1, HTTP/2, gRPC, TCP with or without TLS
Pod
proxy
service B
Service B
proxy
Egress Envoy
External Services

Inbound features
- Service authentication
- Authorization
- Rate limits
- Load shedding
- Telemetry
- Request tracing
- Fault injection

Outbound features
- Service authentication
- Load balancing
- Circuit breaker and retry
- Fine-grained routing
- Telemetry
- Request tracing
- Fault injection

As stated earlier, a service mesh can be built by using sidecars. The sidecars handle both ingress (entering) and egress (exiting) traffic to a service. In a similar way, the Istio mesh manages and controls the ability of services and users to access the services under its control.

The use of sidecars allows for several desirable properties to be included in the service mesh:

- Common functions performed inside the app code are included in the sidecar. These include telemetry, distributed tracing, and TLS termination/initiation.
- Useful features can also be included in the sidecar, such as circuit breakers, rate limits, intelligent routing for A/B testing, and canary releases.

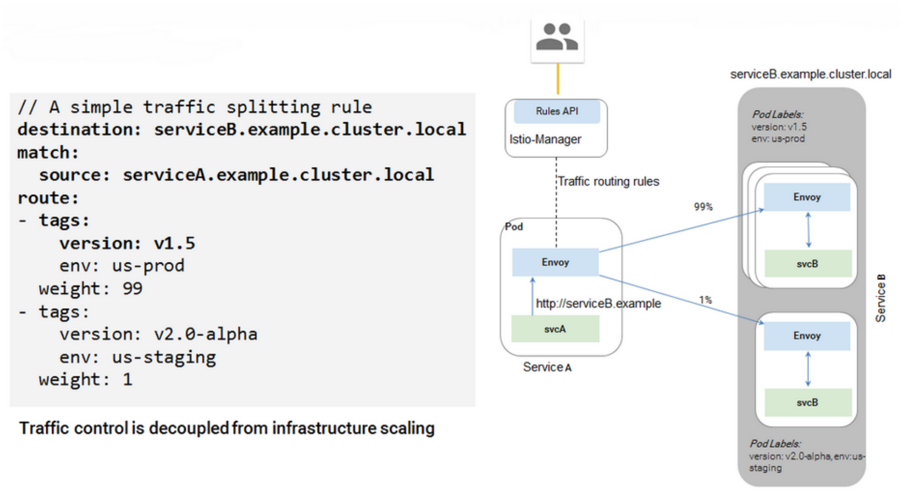
What Istio provides for microservices architectures

Let’s deep dive into few Istio features that demonstrates how the Istio service mesh helps you manage hundreds if not thousands of microservices.

Traffic management benefits

Using Istio’s traffic management model essentially decouples traffic flow and infrastructure scaling, letting operators specify via Pilot what rules they want traffic to follow rather than which specific pods/VMs should receive traffic – Pilot and intelligent Envoy proxies look after the rest. So, for example, you can specify via Pilot that you want 5% of traffic for a particular service to go to a canary version irrespective of the size of the canary deployment, or send traffic to a particular version depending on the content of the request.

Traffic splitting


```
// A simple traffic splitting rule
destination: serviceB.example.cluster.local
match:
  source: serviceA.example.cluster.local
route:
- tags:
    version: v1.5
    env: us-prod
  weight: 99
- tags:
    version: v2.0-alpha
    env: us-staging
  weight: 1
```

Traffic control is decoupled from infrastructure scaling

Discovery and load balancing

Discovery and load balancing

Istio can load balance traffic across instances of a service in a service mesh.

Service registration

Istio assumes the presence of a service registry to keep track of the pods or VMs of a service in the application. It also assumes that new instances of a service are automatically registered with the service registry and unhealthy instances are automatically removed. Platforms such as Kubernetes and Mesos already provide such functionality for container-based applications. A plethora of solutions exist for VM-based applications.

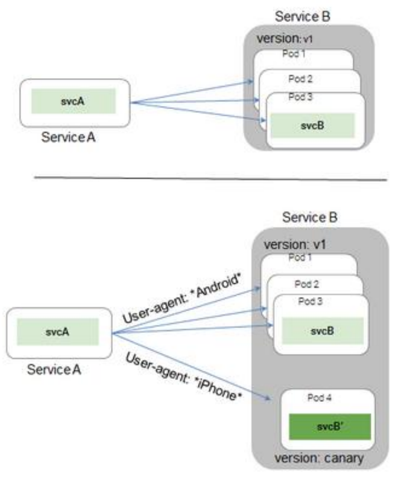
Service discovery

Pilot consumes information from the service registry and provides a platform-agnostic service discovery interface. Envoy instances in the mesh perform service discovery and dynamically update their load balancing pools accordingly.

For more information, see [Discovery & Load Balancing].

Traffic steering

```
// Content-based traffic steering rule
destination: serviceB.example.cluster.local
match:
  httpHeaders:
    user-agent:
      regex: ^(.*?;)?(iPhone)(;.*)?$
precedence: 2
route:
- tags:
    version: canary
```



Content-based traffic steering

Decoupling traffic flow from infrastructure scaling like this allows Istio to provide a variety of traffic management features that live outside the application code. In addition to dynamic [request routing] for A/B testing, gradual rollouts, and canary releases, it also handles [failure recovery] by using timeouts, retries, and circuit breakers, and finally [fault injection] to test the compatibility of failure recovery policies across services. These capabilities are all realized through the Envoy sidecars and proxies deployed across the service mesh.

Handling failures

Envoy provides a set of out-of-the-box opt-in failure recovery features that can be taken advantage of by the services in an application:

- Timeouts
- Bounded retries with timeout budgets and variable jitter between retries
- Limits on number of concurrent connections and requests to upstream services
- Active (periodic) health checks on each member of the load balancing pool
- Fine-grained circuit breakers (passive health checks) that are applied per instance in the load balancing pool

These features can be dynamically configured at runtime through Istio's traffic management rules.

The jitter between retries minimizes the impact of retries on an overloaded upstream service, while timeout budgets ensure that the calling service gets a response (success/failure) within a predictable timeframe.

A combination of active and passive health checks (4 and 5 above) minimizes the chances of accessing an unhealthy instance in the load balancing pool. When combined with platform-level health checks (such as those supported by Kubernetes or Mesos), applications can ensure that unhealthy pods, containers, or VMs can be quickly removed from the service mesh, which minimizes the request failures and impact on latency.

Together, these features enable the service mesh to tolerate failing nodes and prevent localized failures from cascading instability to other nodes.

Fault injection

While Envoy sidecar and proxy provides a host of failure recovery mechanisms to services running on Istio, it is still imperative to test the end-to-end failure recovery capability of the application as a whole. Misconfigured failure recovery policies (for example, incompatible or restrictive timeouts across service calls) can result in continued unavailability of critical services in the application, resulting in poor user experience.

Istio enables protocol-specific fault injection into the network instead of killing pods, which delays or corrupts packets at the TCP layer. The rationale is that the failures observed by the application layer are the same regardless of network level failures, and that more meaningful failures can be injected at the application layer (for example, HTTP error codes) to exercise the resilience of an application.

Operators can configure faults to be injected into requests that match specific criteria. Operators can further restrict the percentage of requests that should be subjected to faults. Two types of faults can be injected: delays and aborts. Delays are timing failures, mimicking increased network latency, or an overloaded upstream service. Aborts are crash failures that mimic failures in upstream services. Aborts usually manifest in the form of HTTP error codes or TCP connection failures.

Mutual TLS Authentication

The aim of Citadel is to enhance the security of microservices and their communication without requiring service code changes. Citadel is responsible for these items:

- Providing each service with a strong identity that represents its role to enable interoperability across clusters and clouds
- Securing service-to-service communication and end-user-to-service communication

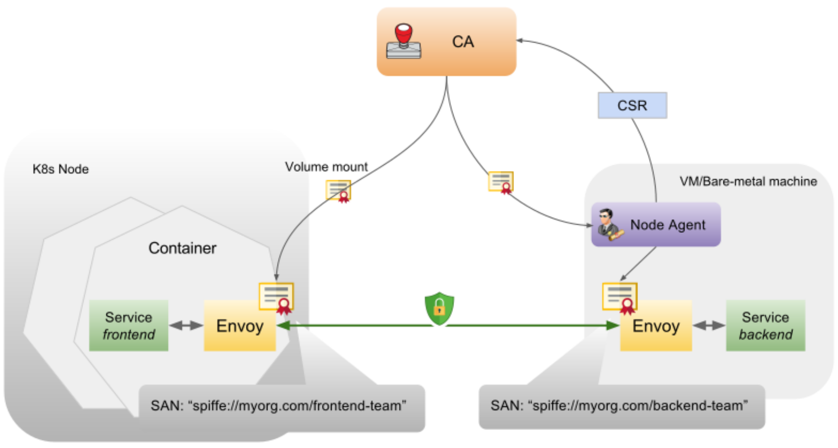
Providing a key management system to automate key and certificate generation, distribution, rotation, and revocation

Architecture

The following diagram shows Citadel's architecture, which includes three primary components:

- Identity
- Key management
- Communication security

This diagram describes how Citadel is used to secure the service-to-service communication between the service frontend that is running as the service account frontend-team and service back end that is running as the service account backend-team. Istio supports services running on both Kubernetes containers and VM or bare-metal machines.



As illustrated in the diagram, Citadel uses secret volume mount to deliver keys/certs from Istio CA to Kubernetes containers. For services running on VM or bare-metal machines, a node agent is used, which is a process running on each VM or bare-metal machine. It generates the private key and CSR (certificate signing request) locally, sends the CSR to the Istio CA for signing, and delivers the generated certificate together with the private key to Envoy.

For more information on Citadel and its components and workflow, see Mutual TLS Authentication.

Automate continuous integration

Early error detection, early system integration, and improved collaboration are the hallmarks of a high-quality product development process. One outcome is greater predictability in delivery schedules.

The effort required to integrate a system increases exponentially with time. By integrating the system more frequently, integration issues are identified earlier, when they are easier to fix, and the overall integration effort is reduced. The result is a higher quality product and more predictable delivery schedules.
Eric Minick, Offering manager for Hybrid Cloud DevOps

Integrate every change to minimize merge conflicts.

Activities
Continuous integration (CI) is implemented through the following activities:

- Changes are delivered and accepted by team members throughout the development day.
- Developers deliver their changes and perform personal builds and unit tests before making the changes available to the team.
- Change sets from all developers are integrated in a team workspace, and then built and unit tested frequently. This should happen at least daily, but ideally it happens any time a new change set is available.

The first activity ensures that any technical debt from conflicting changes is resolved as the changes occur. The second activity identifies integration issues early so that they can be corrected while the change is still fresh in the developer's mind. The third activity ensures that individual developer changes that are introduced to the team have a minimum level of validation through the build and unit testing, and that the changes are made to a configuration that is known to be good and tested before the new code is available.
The ultimate goal of CI is to integrate and test the system on every change to minimize the time between injecting a defect and correcting it.
Benefits of CI
CI provides the following benefits:

- Improved feedback. CI shows constant and demonstrable progress.
- Improved error detection. CI can help you detect and address errors early, often minutes after they've been injected into the product. Effective CI requires automated unit testing with appropriate code coverage.
- Improved collaboration. CI enables team members to work together safely. They know that they can make a change to their code, integrate the system, and determine quickly whether or not their change conflicts with others.
- Improved system integration. By integrating continuously throughout your project, you know that you can actually build the system, thereby mitigating integration surprises at the end of the lifecycle.
- Reduced number of parallel changes that need to be merged and tested.
- Reduced number of errors found during system testing. All conflicts are resolved before making new change sets available, and the resolution is done by the person who is in the best position to resolve them.
- Reduced technical risk. You always have an up-to-date system to test against.
- Reduced management risk. By continuously integrating your system, you know exactly how much functionality you have built to date, thereby improving your ability to predict when and if you are actually going to be able to deliver the necessary functionality.

Get started with CI
If the team is new to CI, it is best to start small and then incrementally add practices. For example, start with a simple daily integration build and incrementally add tests and automated inspections, such as code coverage, to the build process. As the team begins to adopt the practices, increase the build frequency. The following practices provide guidance in adopting CI.
Developer practices
As part of a CI approach, developers follow these practices:

- Make changes available frequently. For CI to be effective, code changes need to be small, complete, cohesive, and available for integration. Keep change sets small so that they can be completed and tested in a relatively short time span.
- Don't introduce errors. Test your changes by using a private build and unit testing before making your changes available.
- Fix broken builds immediately. When a problem is identified, fix it as soon as possible, while it is still fresh in your mind. If the problem cannot be quickly resolved, back out the changes instead of completing them.

Integration practices

A build is more than a compilation. A build consists of compilation, testing, inspection, and deployment. Provide feedback as quickly and as often as possible. Automate the build process so that it is fast and repeatable. In this way, issues are identified and conveyed to the appropriate person for resolution as quickly as possible. Test with build. Include automated tests with the build process and provide results immediately to the team.

Automation

Consider taking these measures to increase automation:

- Commit all of your application assets to the code management (CM) repository so they are controlled and available to the rest of the team. The assets include source code, data definition language source, API definitions, and test scripts.
- Integrate and automate build, deploy, testing, and promotion. Do this for both developer tests and integration tests. Tests must be repeatable and fast.
- Automate feedback from the process to the originator, whether this is the entire team or a developer. Process and resolve feedback, avoiding excess formality, as a part of the backlog process.
- Commit your build scripts to the CM repository so that they are controlled and available to the rest of the team. Both for private builds and integration builds, use automated builds. Builds must be repeatable and fast.
- Invest in a CI server. The goal of CI is to integrate, build, and test the software in a clean environment any time that there is a change to the implementation. Although a dedicated CI server is not essential, it greatly reduces the overhead that is required to integrate continuously and provides the required reporting.

Common pitfalls

As you start to implement CI, remember these potential issues:

- A build process that doesn't identify problems. A build is more than a simple compilation or its dynamic language variations. Sound testing and inspection practices, both developer testing and integration testing, must be adopted to ensure the right amount of coverage.
- Integration builds that take too long to complete. The build process must balance coverage with speed. You don't have to run every system-level acceptance test to meet the intent of CI. Staged builds will provide a useful means to organize testing to get the right balance between coverage and speed.
- Build server measures ignored. Most build servers provide dashboards to measure build results. These results can also be delivered directly to the individual users. Review them to identify trends in applications, components, and architecture that provide an opportunity for improvement.
- Change sets that are too large. Developers must develop the discipline and skills to organize their work into small, cohesive change sets. This practice simplifies testing, debugging, and reporting. It also ensures that changes are made available frequently enough to meet the intention of CI.
- Failure to commit defects to the code management repository. Ensure that developers are performing adequate testing before they make change sets available.

Build and deploy by using continuous delivery

Continuous delivery is a practice by which you build and deploy your software so that it can be released into production at any time.

ntinuous delivery (CD) is a practice by which you build and deploy your software so that it can be released into production at any time. One of the hallmarks of computer science is the shortening of various cycle times in the development and operations process. In the early days of computers when programs were entered in binary with switches and toggles, entering in a program was a time-consuming and error-prone effort. Later, editing was instantaneous, but compile times measured in hours were common for large systems. Within a few years, modern compilers and languages such as Java™ and Ruby had made that a thing of the past, as your code was compiled as quickly as you could save the source file.

The move from continuous integration to continuous delivery

The move away from compilation waits merely shifted the focus of waiting. For many years after that, it was normal for developers to code in isolation on their own aspects of the system while an automated or semi-automated build system ran each night to integrate all of that work. Developers lived in fear of being "the one who broke the build," as a build failure couldn't be resolved until the next night. Many teams had "trophies," such as cardboard cutouts of obnoxious movie characters or funny hats, that were awarded to the people who were responsible for build failure.

That approach began to change with the introduction of continuous integration (CI) tools and practices. When you integrated your code more frequently, the possibility of having a misunderstanding that might lead to a build-breaking problem became less common. In addition, the consequences of breaking a build with a faulty automated test became less severe. Again, the focus shifted. Many teams have implemented CI tools but still do system releases on a quarterly or bi-yearly basis. They still live with the pain of multiple code branches: the code release that they sent to users 5 months ago is still being patched as bugs are fixed, the new code base for the next release is drifting farther away from it, and the possibility of missing new bugs in the new release increases daily.

Frequent small releases: Releases become boring

Why is this true? Why do enterprises and commercial software companies put themselves through the pain and anxiety of the "big bang" release? Probably the biggest reason is inertia. Operations teams carefully defined their operations environments and tweaked them in just the right way to ensure that they are secure, perform well, and are reliable. But as a result, they live in fear of change because change might wreck all of work that went into their carefully constructed environments.

In commercial software, Sales and Marketing teams are used to the twice-per-year training seminars, which they plan their years around. In enterprise development shops, the calendar revolves around pre-planned code freezes, planned vacations to respect those code freezes, and the various audits and checks that are the usual cause of the freezes. What if you turned all of that calendar planning on its head? What if instead of two or four large, disruptive releases, you had much more frequent and smaller releases? You would see several advantages:

- If you change less with each release, the release can break fewer things—that makes the release more predictable and probably easier to roll back.
- If you release more frequently, you vastly reduce the time between concept and rollout—in infrequent releases, the market forces that a feature was designed to address often change by the time it is released.
- You save time, anxiety, and money by having fewer meetings to plan the big-bang releases, less complexity to manage at the time of the releases, and less time spent testing and verifying each release.

The benefits are huge: your team can be more productive, less stressed, and more focused on feature delivery rather than dealing with big, unknown potential changes. In fact, you can go so far as to say that when you do releases often enough, they become predictable and even boring. However, to take advantage of these benefits, you have to embrace a few principles of CD.

Eric Minick, Offering manager for Hybrid Cloud DevOps

Release code frequently using an automated delivery pipeline.

Principles of continuous delivery

- Every change must be releasable: That goes almost without saying, but it hides a deep set of practices that influence the way your development and operations teams interact and join together. If every change is releasable, it has to be entirely self-contained. That includes things like user documentation, operations runbooks, and information about exactly what changed and how for audits and traceability. No one gets to procrastinate.

- Code branches must be short-lived: A practice from CI that applies—especially when you augment CI with CD—is the notion of short-lived code branches. If you branch your code from the main trunk, that branch must live for a only a short period of time before it is merged back into the trunk in preparation for the next release. If your releases are weekly or daily, the amount of time that a developer or team can spend working in a branch is limited greatly.
- Deliver through an automated pipeline: The real trick to achieving CD is the use of an automated delivery pipeline. A well-constructed delivery pipeline can ensure that all of your code releases are moved into your test and production environments in a repeatable, orderly fashion.
- Automate almost everything: Just as the secret to CD is in assembling a reliable delivery pipeline, the key to building a good delivery pipeline is to automate nearly everything in your development process. Automate not only builds and code deployments, but even the process of constructing new development, test, and production environments. If you get to the point of treating infrastructure as code, you can treat infrastructure changes as one more type of code release that makes its way through the delivery pipeline.
- Aim for zero downtime: To ensure the availability of an application during frequent updates, teams can implement blue-green deployments. In a blue-green deployment, when a new function is pushed to production, it is deployed to an instance that isn't the actual running instance. After the new application instance is validated, the public URL is mapped to the new instance of the application.

Microservices vs SOA: How to start an argument

kim.clark@uk.ibm.com
Published on February 9, 2017 / *Updated on April 10, 2017*

5

Apparently simple questions like “how are microservices related to SOA?” are a common cause of heated arguments. Let’s assume we’ve at least managed to separate the core concepts of APIs and microservices discussed in our previous post we’re now moving on to some much more complex questions. Are microservices an evolution of SOA, or are they completely different concepts?

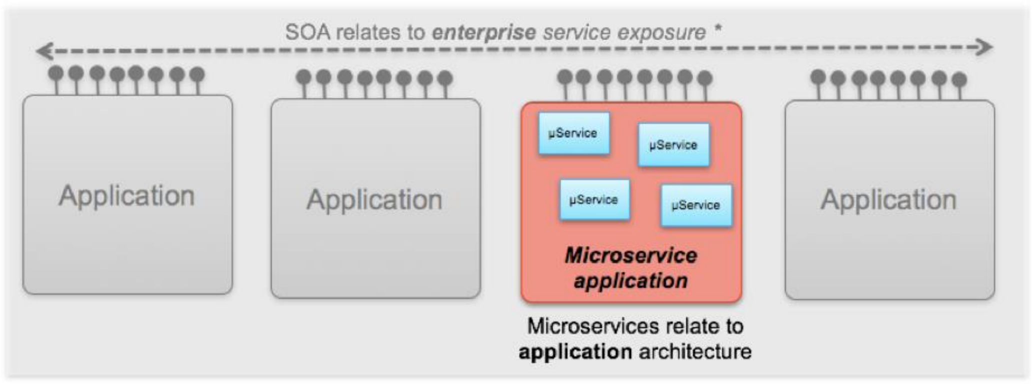
The article Microservices, “SOA, and APIs: Friends or enemies?” discusses this in some detail but for those who don’t want to wade through 15 pages of text, or for that matter the related full webinar, we’ve created a quick 10 minute video and this short post to guide you through some of the key points.

What is microservices architecture?

Microservice architecture might have been better named “micro-component” architecture, because it’s really about breaking applications up into smaller pieces. The aim of a microservice architecture is to completely decouple application components from one another. This is typically with the aim of improving agility; the velocity at which functionality can be delivered. This is primarily because these components should be well decoupled and can therefore be changed independently. Many other potential benefits follow from that including opportunities for more elastic individual scalability, especially in cloud native deployments, and more creative resilience models enabling each component to provide the its own unique up-time model based on its specific business needs. Microservices architecture is a new, more flexible way to build applications.

Could we just say that SOA and microservices relate to different scopes?

A very simplistic view would be that services oriented architecture (SOA) relates to exposing business functions and data via standardized interfaces for re-use across the enterprise. If we then compare this to our definition of microservices, they are an alternative way of building applications.

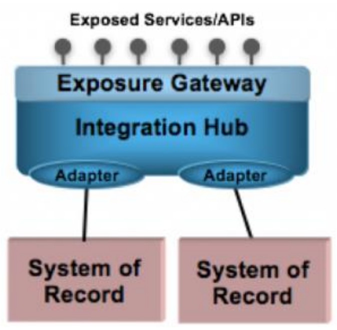

SOA relates to enterprise service exposure *
Application
Application
μService
μService
μService
μService
Microservice application
Application
Microservices relate to application architecture

With this view, microservices and SOA appear to be completely separate concepts, one at the enterprise scope and the other at the application scope. You well might wonder what all the fuss is about!

What was SOA really about, integration or component decomposition?

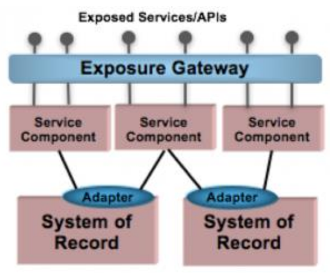
The real challenge is that there are very different views out there on what SOA was really about. Let's take a look at how two very different views could evolve.

For large enterprises, with a multitude of aging systems of record, an SOA programme is likely to spend a significant amount of time solving hard integration problems in order to surface complex back end systems of record as re-usable interfaces. After a while, it may start to feel like SOA was all about integration.



However, the original aims of SOA were broader. The desire was to re-landscape the systems of record into business services more applicable for the applications of the future.

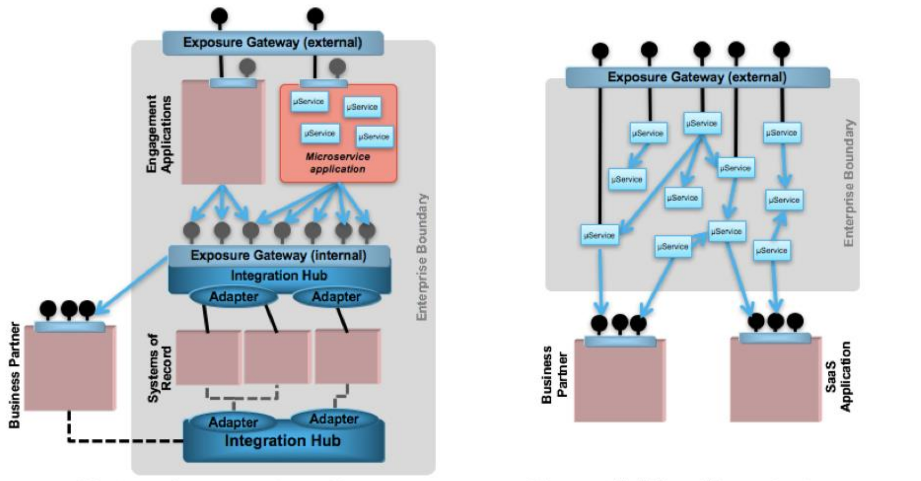
For a perhaps smaller company with fewer, more modern systems of record, the integration aspect of SOA might be comparatively trivial, and most of the time might be spent on this re-landscaping, probably creating an intermediate layer of "service components" that represent those valuable business services.



As technology moves forward, and we create more lightweight runtimes and methods for building those components, we can create smaller, more independent service components. You can see where this is going. We could call these "microservice components", and now suddenly microservices looks more like an evolution of SOA rather than something completely different to it. Both view have merit, depending on your definition of SOA in the first place.

How pervasive are microservices?

The other reason for the wildly differing opinions is how much of your estate is likely to become microservices based. In a mature large enterprise such as a bank or insurance company with IT systems created and acquired over multiple decades, microservices architecture is just another option for now to build an application. Microservices not going to take over a large enterprise's IT landscape any time soon any more than SOA did. It's just an alternative application architecture, well suited to a particular set of solutions.



Mature large enterprise
Microservices are just one style of application
Exposing services is an *integration and data* challenge

Green field online start-up
Much of landscape could be microservice based
The landscape *is* a (micro)service oriented architecture

Now if we compare that to the other extreme, a green field online start-up. They might use microservices as their fundamental architectural style, and everything they build, is done in that way. In their situation, they could describe their entire estate as "(micro)services oriented architecture". We can see now that for this company, microservices could look very much like an evolution of service oriented architecture, and they might well see it as an enterprise wide architectural pattern.

We should recognize that this second situation isn't limited to small start-up. However, it is inevitably more common in "modern" companies with less IT legacy.

Some microservices principles are really different to SOA

It's also worth knowing that microservices architecture introduces some principles that will be pretty surprising to someone from an SOA background. Here are some key examples:

- **Re-use**: Microservices is about creating small truly independent components. So for example, for SOA re-use was one of the primary perceived benefits. The ability to re-use data and function from one application in another. However, for microservice components, re-using a component, especially at runtime, and perhaps even at design time results in dependencies that make the resulting solution fragile, so re-use is often discouraged.
- **Inter-communication**: SOA typically resulted in the exposure of synchronous web services such as SOAP/HTTP. In microservices, a synchronous protocol such as HTTP, whether SOAP or REST based causes a real-time dependency between components, so messaging is the preferred

transport between microservices wherever possible. REST is used, but generally reserved just for exposing the microservice beyond the bounds of the application, or to the outside world.

There's more too, for example with issues relating to service discovery, and data duplication and more. For all their similarities, when it comes to the detail the two are very different in some fundamental ways.

Microservices are certainly more focused on application architecture. Whether you can use those concepts at the enterprise level strongly depends on the shape and size of your enterprise. SOA is unquestionably an enterprise level concept. You can certainly bring microservices techniques in to help how you build that out, but be careful when the two overlap as some of the core principles are very different.

How to start an argument

I promised you I'd tell you how to start an argument! Debates typically get rolling when things are being compared that don't match. For example asking people to compare "Microservices and SOA" is itself misleading as a microservice is a component and SOA is, well, an architecture obviously. Now if we were to compare microservices architecture with service oriented architecture, we'd at least have a fair comparison, and we could discuss whether each of them are application architecture or enterprise architecture.

Final thoughts

The core concepts for this were taken from the article "SOA, and APIs: Friends or enemies?" so feel free to get the full story from there. There is also a full length webinar covering the broader points in that article.

In future posts we'll go on to look at how the products within IBM's hybrid integration portfolio are typically used to complement SOA and microservices architecture, and how to decide what is most appropriate for your organization.