Microservices applications

A very small number of developers recognized the need for new thinking very early on and started working on the evolution of a new architecture, called microservices, early in 2014.

Early pioneers

A few individuals took a forward leap in moving away from monolithic to small manageable services adoption in their respective companies. Some of the most notable of these people include Jeff Bezos, Amazon's CEO, who famously implemented a mandate for Amazon (https://bit.ly/2Hb3NI5 (https://bit.ly/2Hb3NI5)) in 2002. It stated that all employees have to adopt a service interface methodology where all communication calls would happen over the network. This daring initiative replaced the monolith with a collection of loosely coupled services. One nugget of wisdom from Jeff Bezos was two-pizza teams – individual teams shouldn't be larger than what two pizzas can feed. This colloquial wisdom is at the heart of shorter development cycles, increased deployment frequency, and faster time to market.

Netflix adopted microservices early on. It's important to mention Netflix's **Open Source Software Center** (**OSS**) contribution through https://netflix.github.io (https://netflix.github.io). Netflix also created a suite of automated open source tools, the Simian Army (https://github.com/Netflix/SimianArmy (https://github.com/Netflix/SimianArmy)), to stress-test its massive cloud infrastructure. The rate at which Netflix has adopted new technologies and implemented them is phenomenal.

Lyft adopted microservices and created an open source distributed proxy known as Envoy (https://www.envoyproxy.io/ (https://www.envoyproxy.io/)) for services and applications, and would later go on to become a core part of one of the most popular service mesh implementations, such as Istio and Consul.

Though this book is not about developing microservices applications, we will briefly discuss the microservices architecture so that it is relevant from the perspective of a service mesh.

Since early 2000, when machines were still used as bare metal, three-tier monolithic applications ran on more than one machine, leading to the concept of distributed computing that was very tightly coupled. Bare metal evolved into VMs and monolithic applications into SOA/ESB with an API gateway. This trend continued until 2015 when the advent of containers disrupted the SOA/ESB way of thinking toward a self-contained, independently managed service. Due to this, the term *microservice* was coined.

The first mention of microservice as a term was used in a workshop of software architects in 2011 (https://bit.ly/1KljYiZ (https://bit.ly/1KljYiZ)) when they used the term microservice to describe a common architectural style as a fine-grained SOA.

Chris Richardson created https://microservices.io (https://microservices.io) in January 2014 to document architecture and design patterns.

James Lewis and Martin Fowler published their blog post (https://martinfowler.com/articles/microservices.html) (https://martinfowler.com/articles/microservices.html)) about microservices in March 2014, and this blog post popularized the term microservices.

The microservices boom started with easy containerization that was made possible by Docker and through a *de facto* container orchestration platform known as Kubernetes, which was created for distributed computing.

What is a microservice?

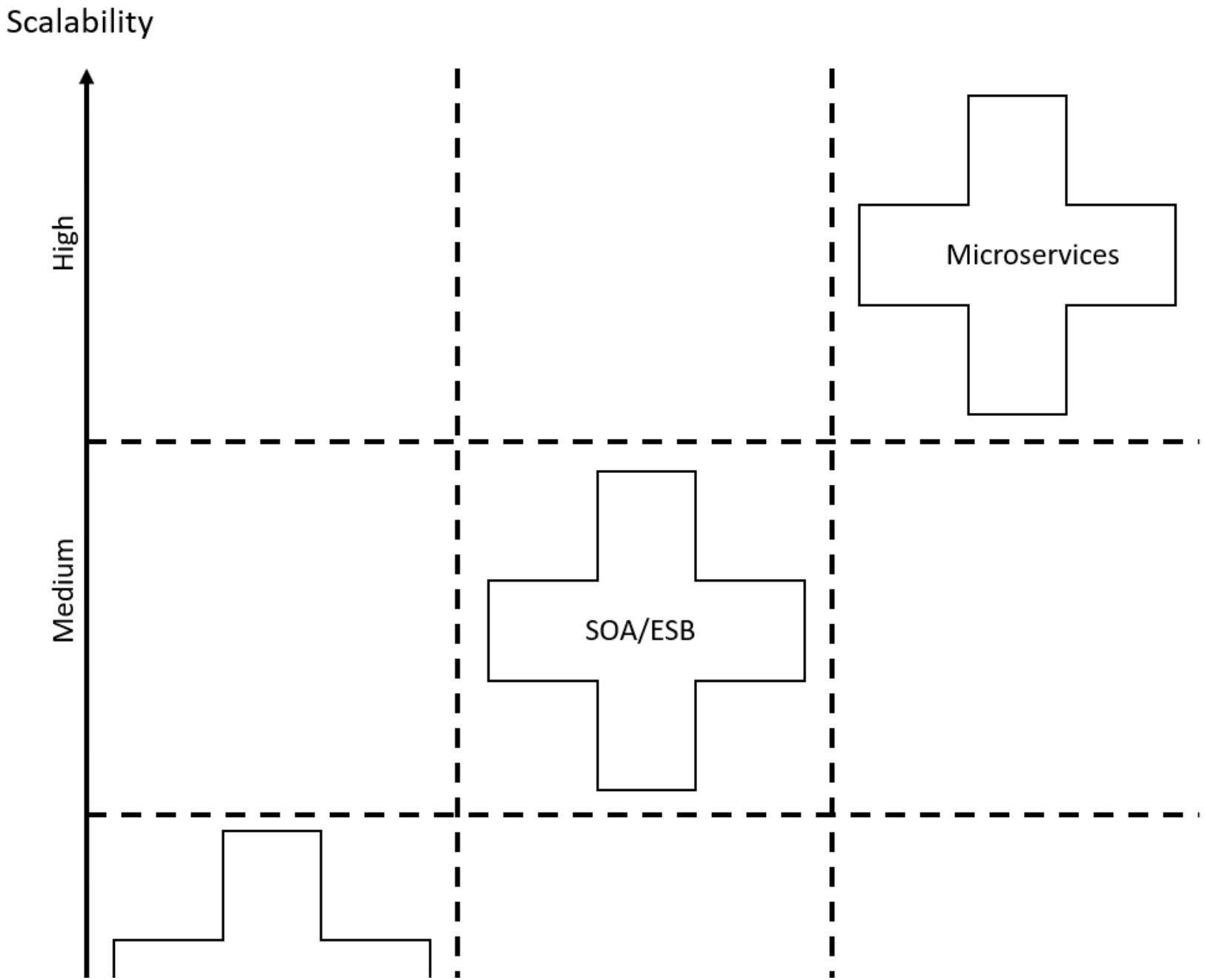
The natural transition of SOA/ESB is toward microservices, in which services are decoupled from a monolithic ESB. Let's go over the core points of microservices:

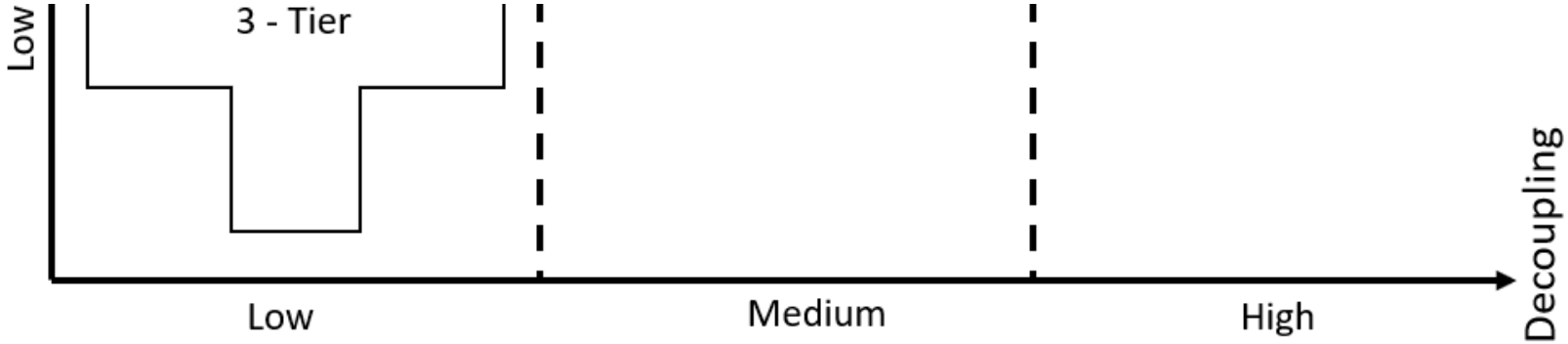
- Each service is autonomous, which is developed and deployed independently.
- Each microservice can be scaled independently in relation to others if it receives more traffic without having to scale other microservices.
- Each microservice is designed based on the business capabilities at hand so that each service serves a specific business goal with a simple time principle that it does only one thing, and does it well.
- Since services do not share the same execution runtime, each microservice can be developed in different languages or in a polyglot fashion, providing agility in which developers pick the best programming language to develop their own service.
- The microservices architecture eliminated the need for a centralized ESB. The business logic, including inter-service communication, is done through smart endpoints and dumb pipes. This means that the centralized business logic of ESBs is now distributed among the microservices through smart endpoints, and a primitive messaging system or a dumb pipe is used for service-to-service communication using a lightweight protocol such as REST or gRPC.

The evolution of SOA/ESB to the microservices pattern was mainly influenced by the idea of being able to adapt to smaller teams that are independent of each other and to provide a self-service model for the consumption of services that were created by smaller teams. At the time of writing, microservices is a winning pattern that is being adopted by many enterprises to modernize their existing monolithic application stack.

Evolution of microservices

The following diagram shows the evolution of the application architecture from a three-tier architecture to SOA/ESB and then to microservices in terms of flexibility toward scalability and decoupling:





Tiered Vs SOA Vs Microservices

Credit: Paolo Maresca

Microservices have evolved from being tiered and the SOA architecture and are becoming the accepted pattern for building modern applications. This is due to the following reasons:

- Extreme scalability
- Extreme decoupling
- Extreme agility

These are key points regarding the design of a distributed scalable application where developers can pick the best programming language of their choice to develop their own service.

A major differentiation between monolithic and microservices is that, with microservices, the services are loosely coupled, and they communicate using dumb pipe or low-level REST or gRPC protocols. One way to achieve loose coupling is through the use of a separate data store for each service. This helps services isolate themselves from each other since a particular service is not blocked due to another service holding a data lock. Separate data stores allow the microservices to scale up and down, along with their data stores, independently of all the other services.

It is also important to point out the early pioneers in microservices, which we will discuss in the next section.

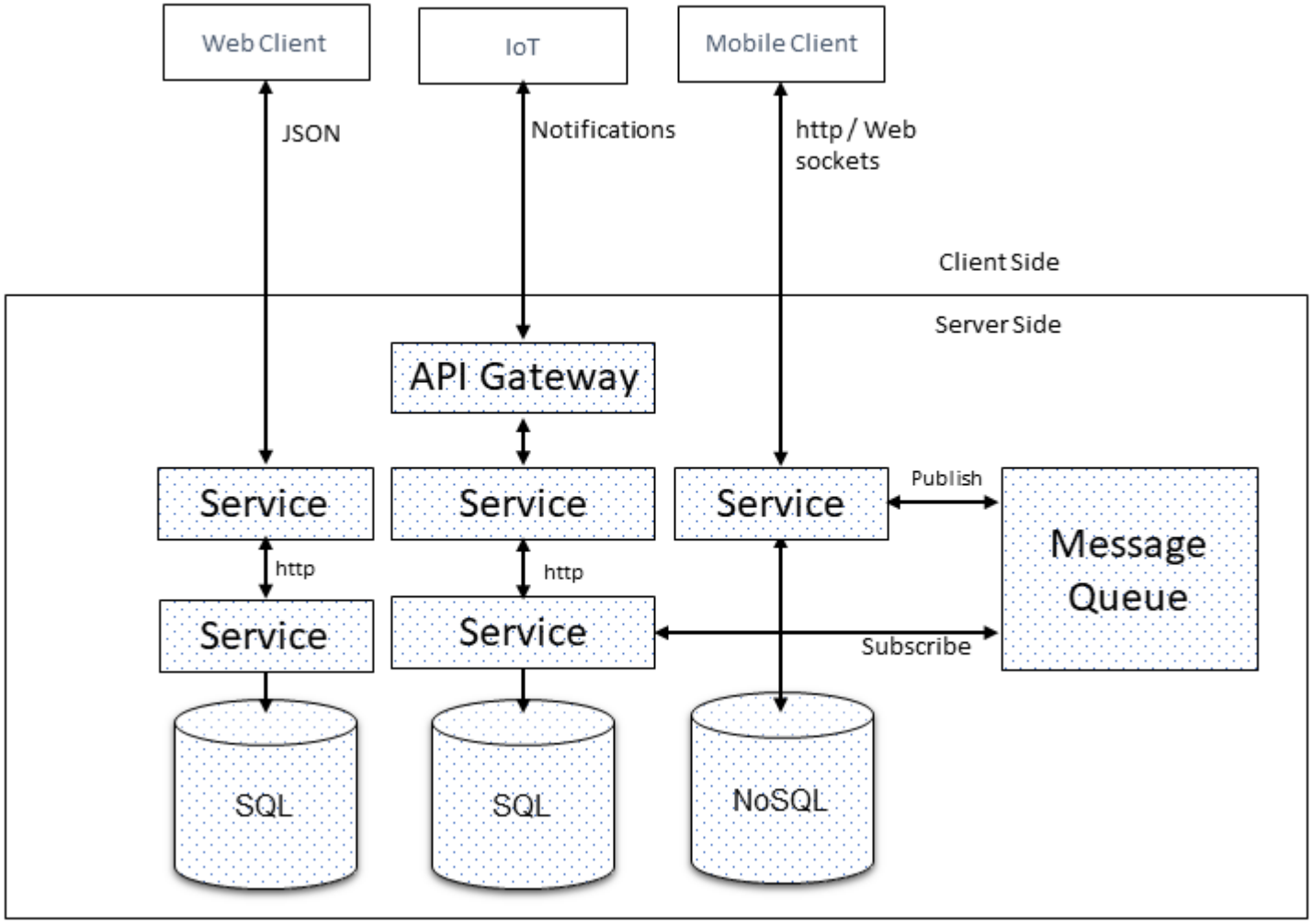
Microservices architecture

The aim of a microservice architecture is to completely decouple app components from one another so that they can be maintained, scaled, and more. It's an evolution of the app architecture, SOA, and publishing APIs:

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

In Martin Fowler's paper, he states that the microservice architecture would have been better named the micro-component architecture because it is really about breaking apps up into smaller pieces (micro-components). For more information, see *Microservices,* by Martin Fowler, at https://martinfowler.com/articles/microservices.html (https://martinfowler.com/articles/microservices.html). Also, check out Kim Clark's IBM blog post on microservices at https://developer.ibm.com/integration/blog/2017/02/09/microservices-vs-soa (https://developer.ibm.com/integration/blog/2017/02/09/microservices-vs-soa), where he argues microservices as micro-components.

The following diagram shows the microservice architecture in which different clients consume the same services. Each service can use the same/different language and can be deployed/scaled independently of each other:



Each microservice runs its own process. Services are optimized for a single function and they must have one, and only one, reason to change. The communication between services is done through REST APIs and message brokers. The CICD is defined per service. The services evolve at a different pace. The scaling policy for each service can be different.

Benefits and drawbacks of microservices

The explosion of microservices is not an accident, and it is mainly due to rapid development and scalability:

- **Rapid development**: Develop and deploy a single service independently. Focus only on the interface and the functionality of the service and not the functionality of the entire system.
- **Scalability**: Scale a service independently without affecting others. This is simple and easy to do in a Kubernetes environment.

The other benefits of microservices are as follows:

- Each service can use a different language (better polyglot adaptability).
- Services are developed on their own timetables so that the new versions are delivered independently of other services.
- The development of microservices is suited for cross-functional teams.
- Improved fault isolation.
- Eliminates any long-term commitment to a technology stack.

However, the microservice is not a panacea and comes with drawbacks:

- The complexity of a distributed system.
- Increased resource consumption.
- Inter-service communication.
- Testing dependencies in a microservices-based application without a tool can be very cumbersome.
- When a service fails, it becomes very difficult to identify the cause of a failure.
- A microservice can't fetch data from other services through simple queries. Instead, it must implement queries using APIs.
- Microservices lead to more Ops (operations) overheads.

There is no perfect silver bullet, and technology continues to emerge and evolve. Next, we'll discuss the future of microservices.

Future of microservices

Microservices can be deployed in a distributed environment using a container orchestration platform such as Kubernetes, Docker Swarm, or an on-premises **Platform as a Service** (**PaaS**), such as Pivotal Cloud Foundry or Red Hat OpenShift.

Service mesh helps reduce/overcome the aforementioned challenges and overheads on Ops, such as the operations overhead for manageability, serviceability, metering, and testing. This can be made simple by the use of service mesh providers such as Istio, Linkerd, or Consul.

As with every technology, there is no perfect solution, and each technology has its own benefits and drawbacks regarding an individual's perception and bias toward a particular technology. Sometimes, the drawbacks of a particular technology outweigh the benefits they accrue.

In the last 20 years, we have seen the evolution of monolithic applications to three-tier ones, to the adoption of the SOA/ESB architecture, and then the transition to microservices. We are already witnessing a framework evolution around microservices using service mesh, which is what this book is based on.

‹ Previous Section (/book/web_development/9781789615791/2/ch02lvl1sec04/monolithic-applications)

Next Section › (/book/web_development/9781789615791/2/ch02lvl1sec06/summary)
