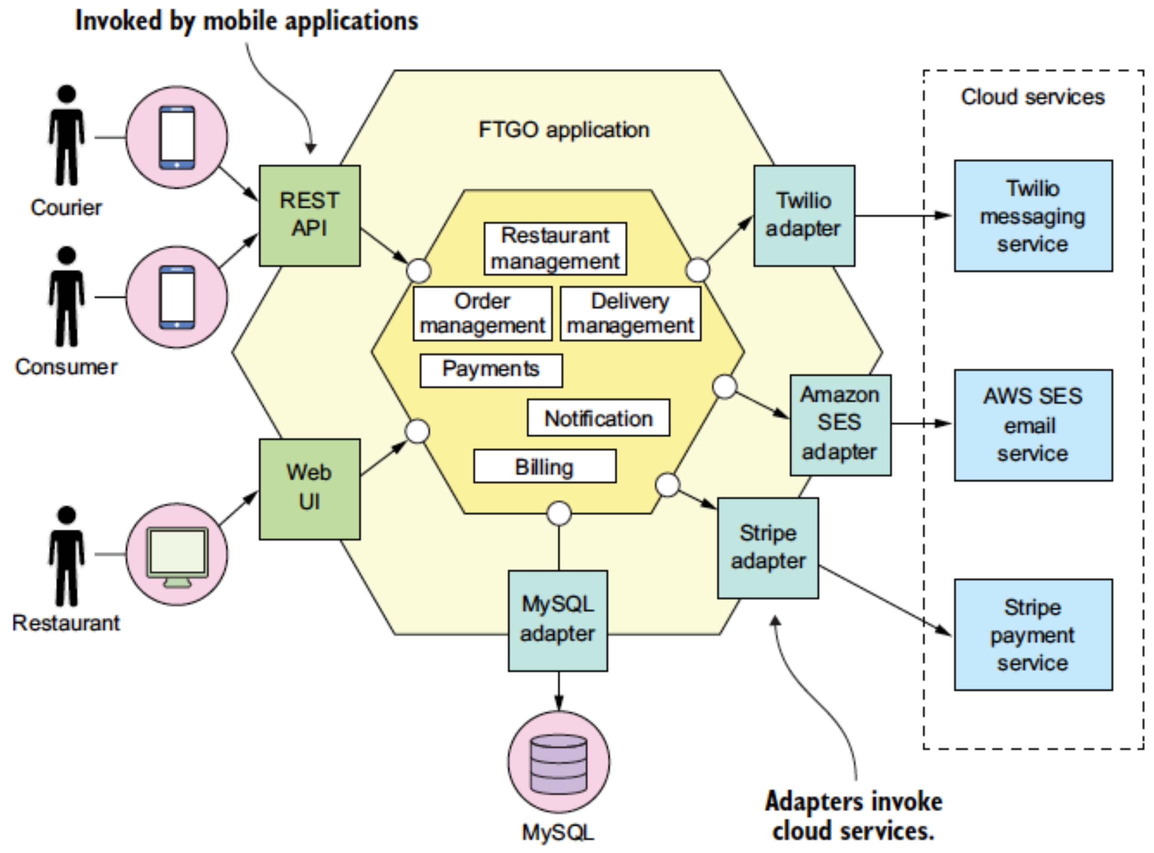
1.1 Motivadores de decisión de negocio y de tecnología

Raúl Estrada

Noviembre 2020

Architecture



Benefits of monolith

- **Simple to develop**—IDEs and other developer tools are focused on building a single application.
- **Easy to make radical changes to the application**—You can change the code and the database schema, build, and deploy.
- **Straightforward to test**—The developers wrote end-to-end tests that launched the application, invoked the REST API, and tested the UI with Selenium.
- **Straightforward to deploy**—All a developer had to do was copy the WAR file to a server that had Tomcat installed.
- **Easy to scale**—And ran multiple instances of the application behind a load balancer.

Complexity intimidates

- A major problem with a monolith application is that it’s too complex.
- It’s too large for any developer to fully understand.
- As a result, fixing bugs and correctly implementing new features have become difficult and time consuming.
- Deadlines are missed.
- If the code base is difficult to understand, a developer won’t make changes correctly.
- Each change makes the code base incrementally more complex and harder to understand.

Development is slow

- As well as having to fight overwhelming complexity, monolith developers find day-to-day development tasks slow.
- The large application overloads and slows down a developer’s IDE.
- Building a monolith application takes a long time.
- Moreover, because it’s so large, the application takes a long time to start up.
- As a result, the edit-build-run-test loop takes a long time, which badly impacts productivity.

Deployment path is long

- Another problem with the monolith application is that deploying changes into production is a long and painful process.
- The team typically deploys updates to production once a month, usually late on a Friday or Saturday night.
- Another reason it takes so long to get changes into production is that testing takes a long time.
- The code base is complex and the impact of a change isn’t well understood, developers and the Continuous Integration (CI) server must run the entire test suite.

Scaling is difficult

- The monolith team also has problems scaling its applications.
- That’s because different application modules have conflicting resource requirements.

A reliable monolith is a challenge

- Another problem with a monolith application is the lack of reliability.
- As a result, there are frequent production outages.
- One reason it's unreliable is that testing the application thoroughly is difficult, due to its large size.
- This lack of testability means bugs make their way into production.
- To make matters worse, the application lacks fault isolation, because all modules are running within the same process.

Monolith are locked into tech stack

- The final aspect of monolithic is that the architecture forces them to use a technology stack that’s becoming increasingly obsolete.
- The monolithic architecture makes it difficult to adopt new frameworks and languages.
- It would be extremely expensive and risky to rewrite the entire monolithic application so that it would use a new and presumably better technology.

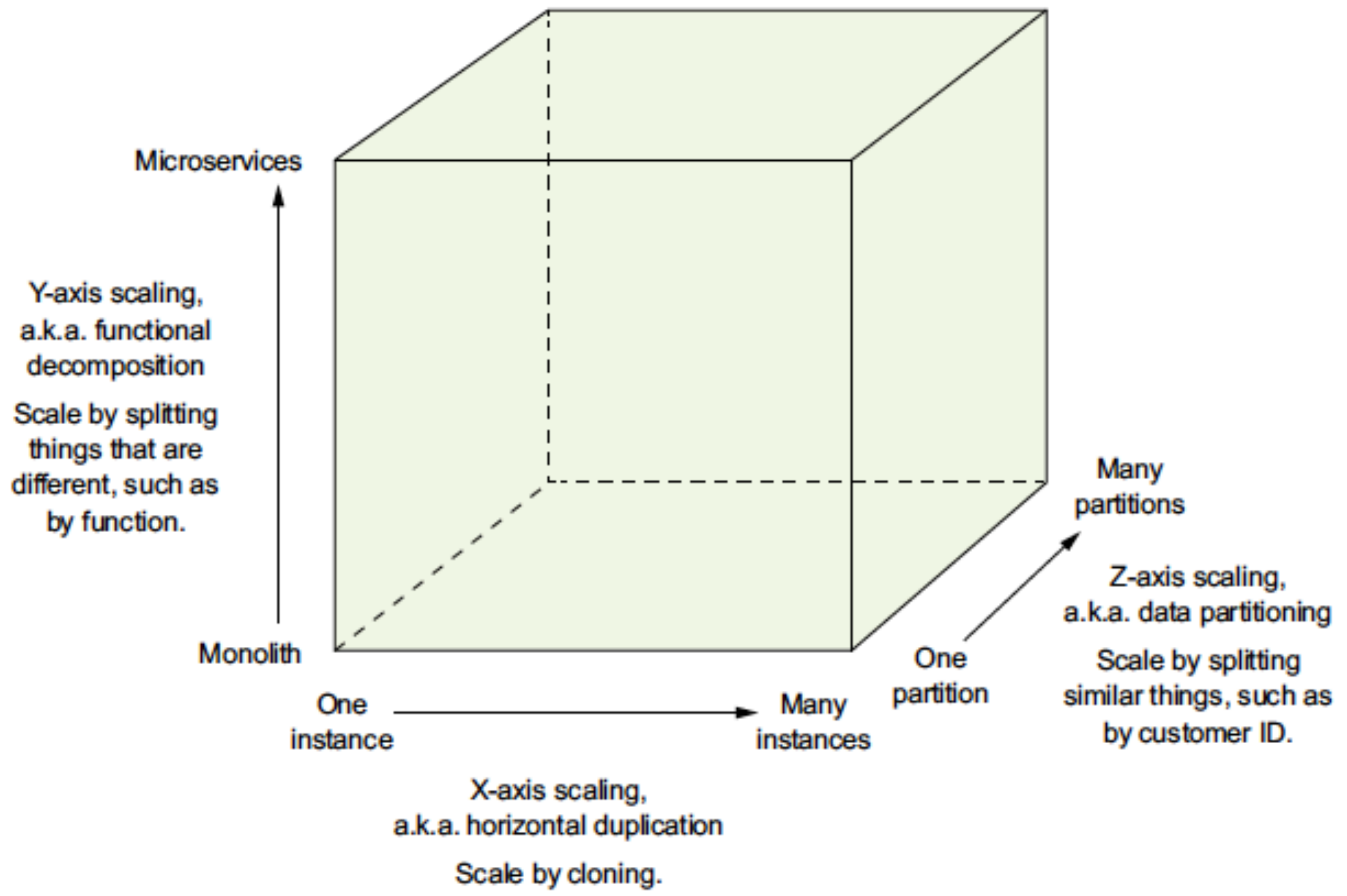
Scale cube and microservices

The book: Martin Abbott and Michael Fisher’s The Art of Scalability (Addison-Wesley, 2015) describes a useful, three-dimensional scalability model: the **scale cube**

The scale cube defines three separate ways to scale an application:

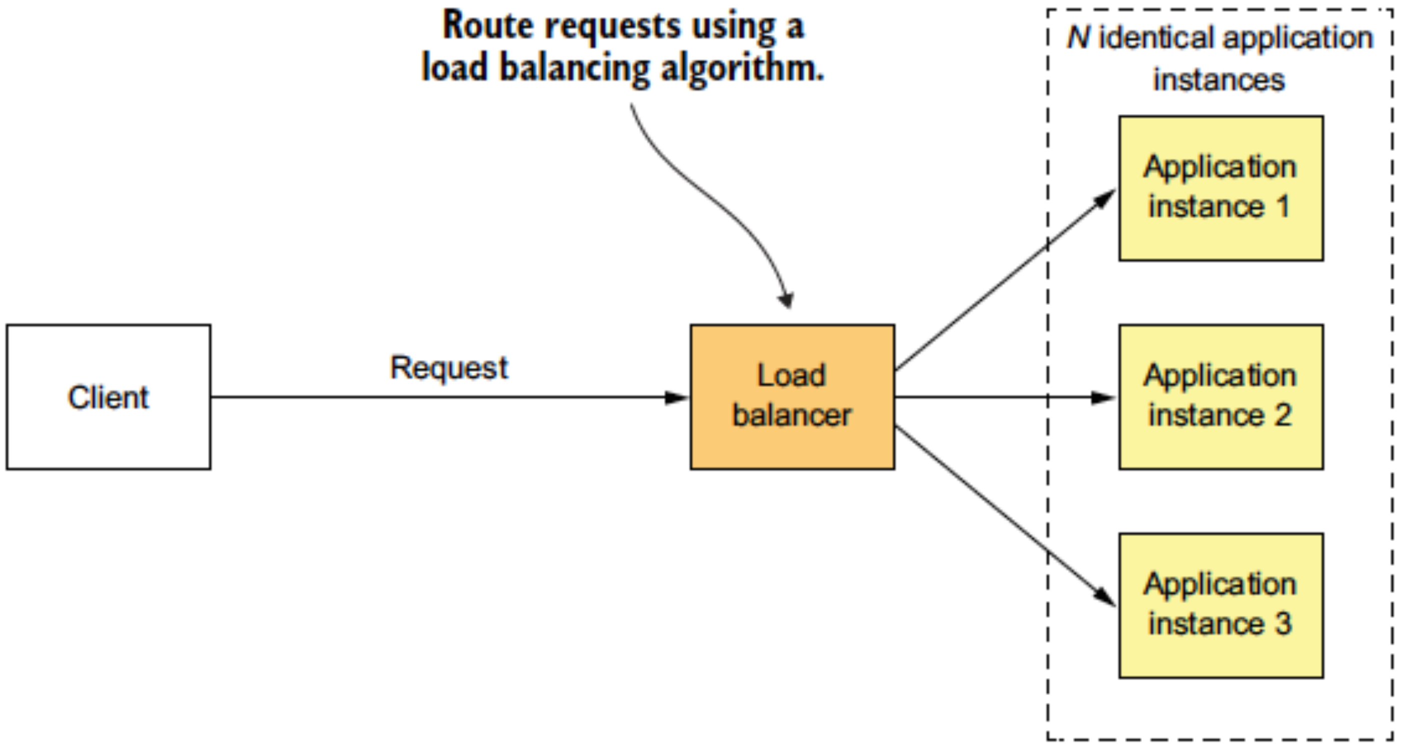
- X-axis scaling load balances requests across multiple, identical instances;
- Z-axis scaling routes requests based on an attribute of the request;
- Y-axis functionally decomposes an application into services.

Scale cube


Microservices
Y-axis scaling, a.k.a. functional decomposition
Scale by splitting things that are different, such as by function.
Monolith
One instance
Many instances
X-axis scaling, a.k.a. horizontal duplication
Scale by cloning.
One partition
Many partitions
Z-axis scaling, a.k.a. data partitioning
Scale by splitting similar things, such as by customer ID.

X-Axis: Load balancer

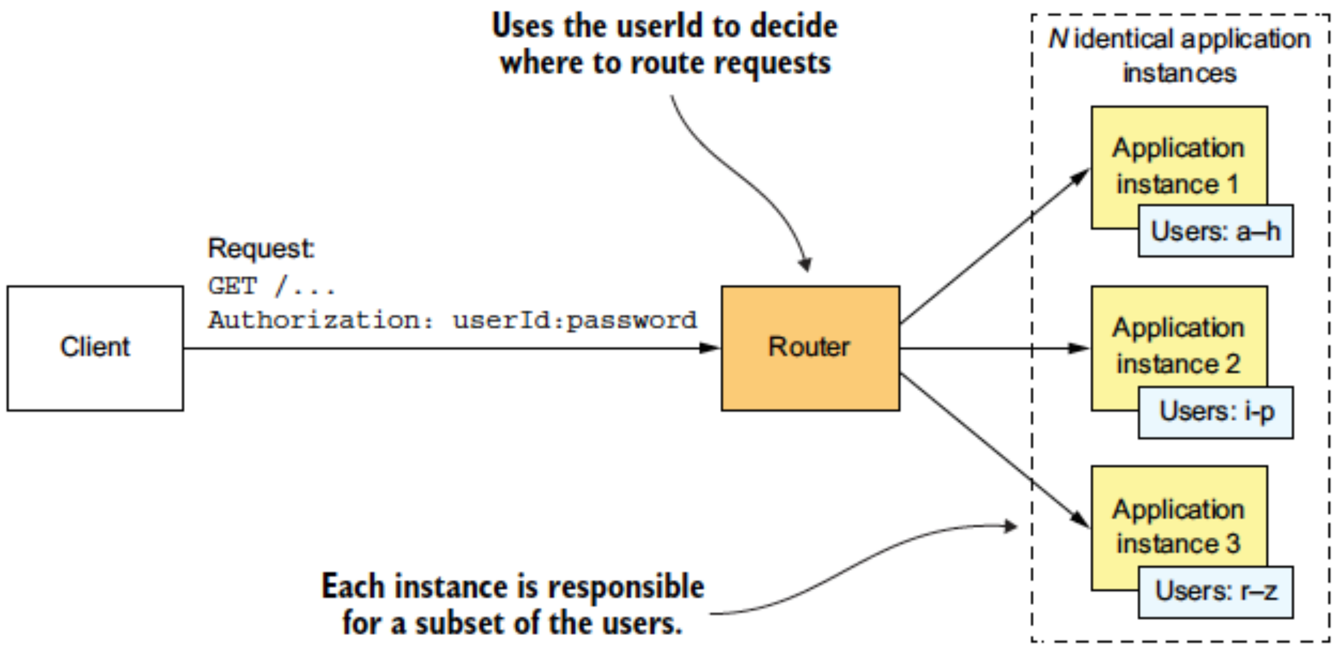
- X-axis scaling is a common way to scale a monolithic application.
- You run multiple instances of the application behind a load balancer.
- The load balancer distributes requests among the N identical instances of the application.
- This is a great way of improving the capacity and availability of an application.



X-axis scaling runs multiple, identical instances of the monolithic application behind a load balancer.

Z-Axis: Based on the request

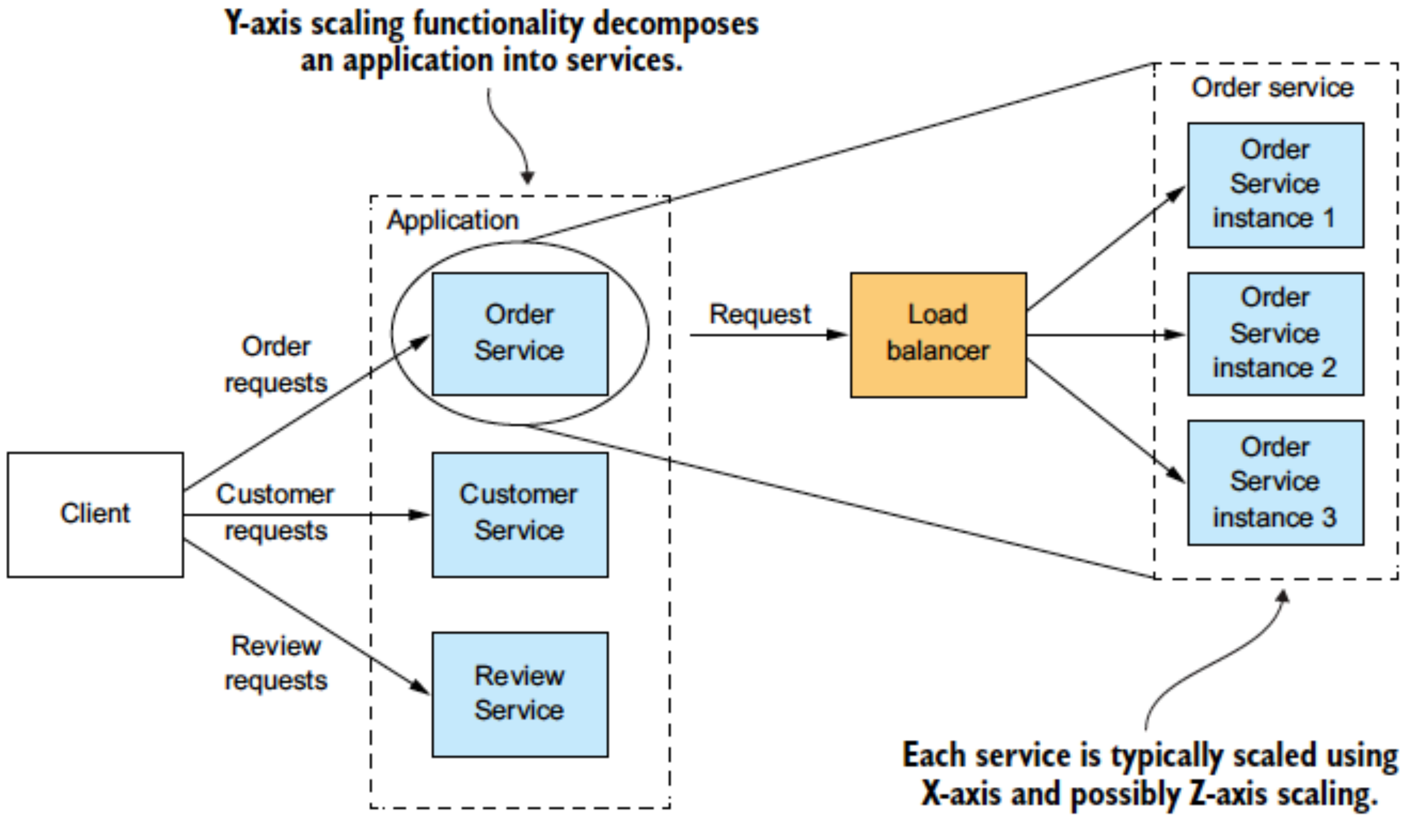
- Z-Axis Scaling routes requests based on an attribute of the request.
- Z-axis scaling also runs multiple instances of the monolith application, but unlike X-axis scaling, each instance is responsible for only a subset of the data.
- The router in front of the instances uses a request attribute to route it to the appropriate instance. An application might, for example, route requests using userId.


Uses the userId to decide where to route requests
N identical application instances
Application instance 1
Users: a–h
Request:
GET /...
Authorization: userId:password
Client
Router
Application instance 2
Users: i-p
Application instance 3
Users: r–z
Each instance is responsible for a subset of the users.

- Z-axis scaling runs multiple identical instances of the monolithic application behind a router, which routes based on a request attribute.
- Each instance is responsible for a subset of the data.

Y-Axis: Functional decomposition

- X- and Z-axis scaling improve the application’s capacity and availability.
- But neither approach solves the problem of increasing development and application complexity.
- To solve those, you need to apply Y-axis scaling, or functional decomposition.
- By splitting a monolithic application into a set of services.



- Y-axis scaling splits the application into a set of services.
- Each service is responsible for a particular function.
- A service is scaled using X-axis scaling and, possibly, Z-axis scaling.

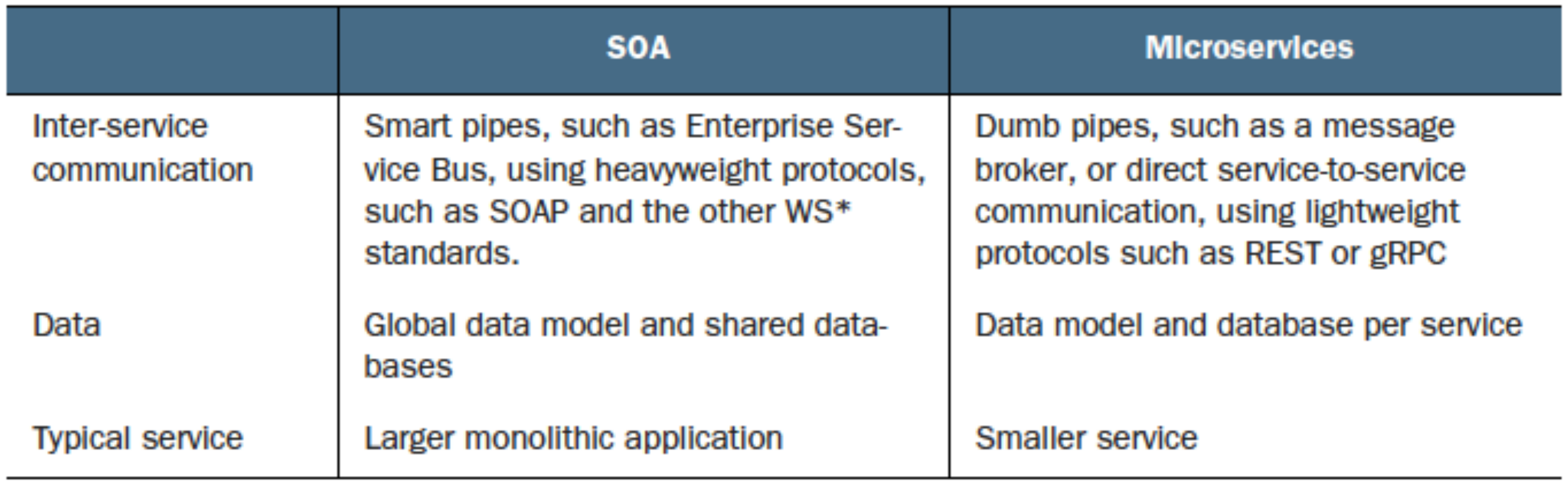
Services

- A service is a mini application that implements narrowly focused functionality, such as order management, customer management, and so on.
- A service is scaled using X-axis scaling, though some services may also use Z-axis scaling.
- For example, the Order service consists of a set of load-balanced service instances.

Microservices

- The high-level definition of microservice architecture (microservices) is an architectural style that functionally decomposes an application into a set of services.
- Note that this definition doesn’t say anything about size.
- Instead, what matters is that each service has a focused, cohesive set of responsibilities.

SOA vs Microservices



	SOA	Microservices
Inter-service communication	Smart pipes, such as Enterprise Service Bus, using heavyweight protocols, such as SOAP and the other WS* standards.	Dumb pipes, such as a message broker, or direct service-to-service communication, using lightweight protocols such as REST or gRPC
Data	Global data model and shared databases	Data model and database per service
Typical service	Larger monolithic application	Smaller service

Benefits of the microservice architecture

The microservice architecture has the following benefits:

- It enables the continuous delivery and deployment of large, complex applications.
- Services are small and easily maintained.
- Services are independently deployable.
- Services are independently scalable.
- The microservice architecture enables teams to be autonomous.
- It allows easy experimenting and adoption of new technologies.
- It has better fault isolation.

Drawbacks of a microservice architecture

The major drawbacks and issues of the microservice architecture:

- Finding the right set of services is challenging.
- Distributed systems are complex, which makes development, testing, and deployment difficult.
- Deploying features that span multiple services requires careful coordination.
- Deciding when to adopt the microservice architecture is difficult.