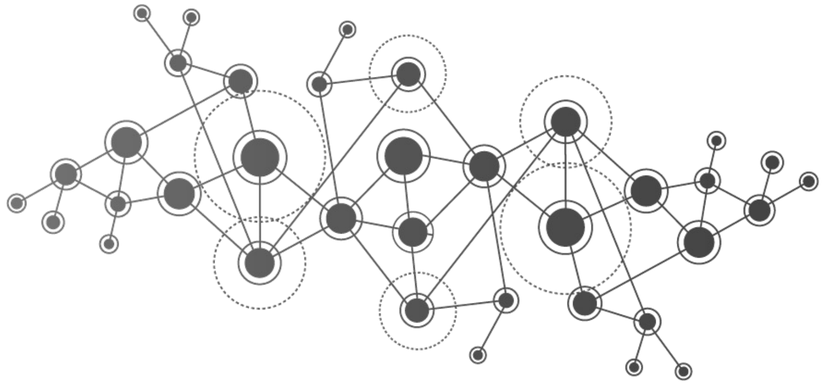
UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA



Intelligent Internet of Things

Monolithic vs Microservice oriented Architectures for the IoT

Prof. Marco Picone

A.A 2023/2024

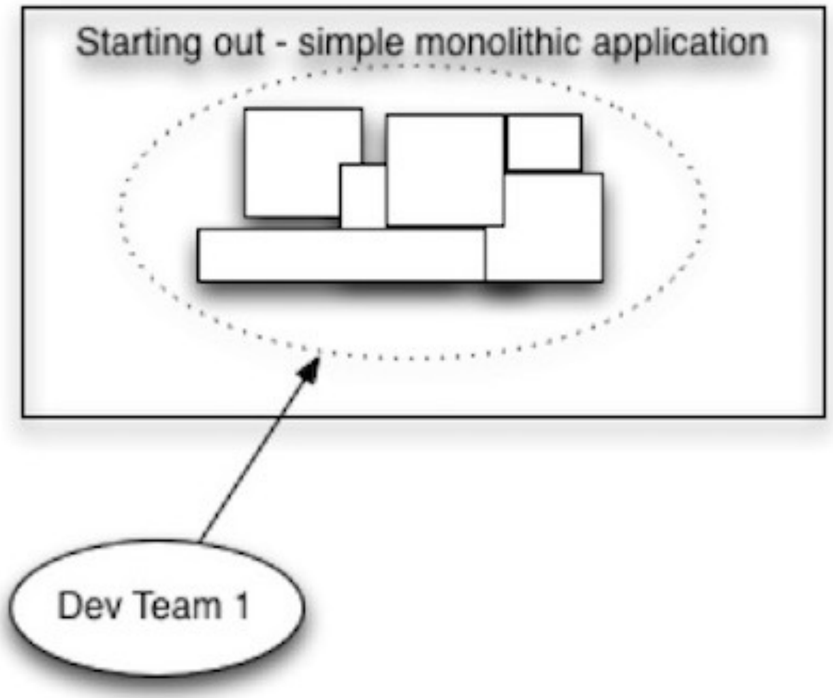
Monolithic vs Microservice oriented Architectures for the IoT

- Microservices
- Service Mesh
- Microservices composition and orchestration in Distributed Environments
- IoT and Microservices
- Sidecar Object Pattern

2023 - Marco Picone, Ph.D. (marco.picone@unimore.it)

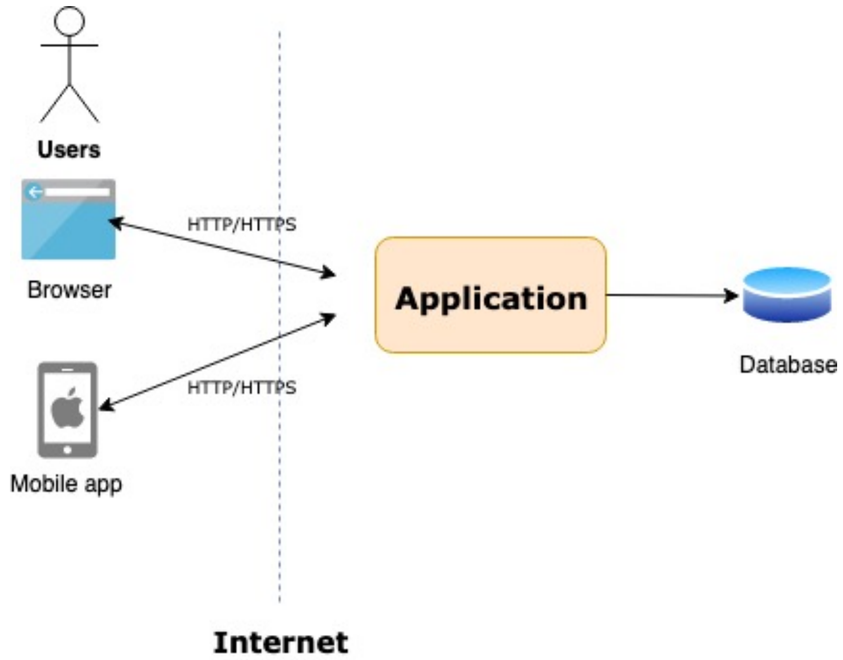
Monoliths

- Traditionally, software developers created large, monolithic applications
- A single monolith would contain all the code for all the business activities an application performed. **As the application's requirements grew, of course, so did the monolith**
- From a Development point of view -> you write a simple application, which is developed and managed by a single team



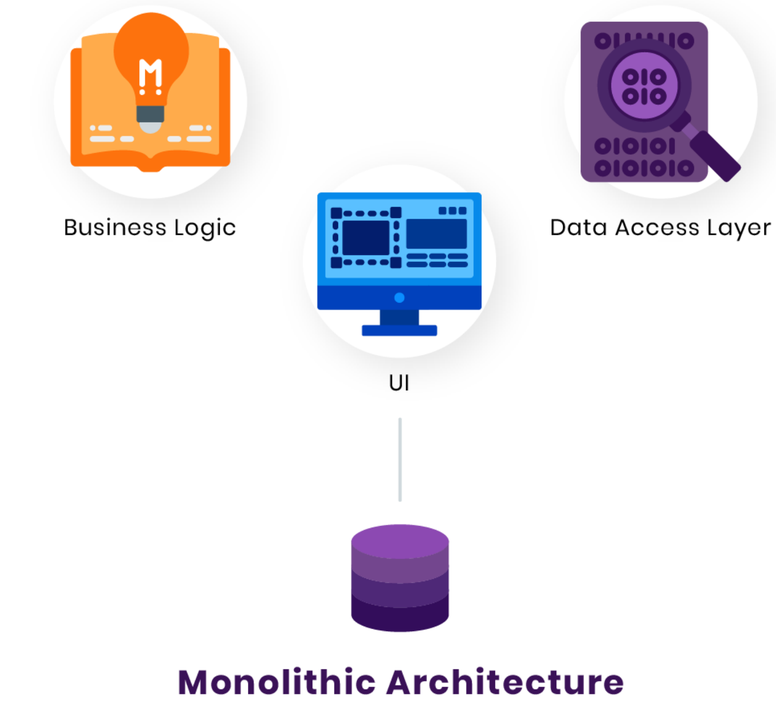
The Monolith approach

- The code might be exceptionally well organized, but overall, it’s still just **one big black box**
- the inner workings of a monolith are opaque, or hard to see, instead of transparent.
- New developers will take time and effort to find code that needs to be adjusted and/or extended
- It’s all connected in one piece, **so anything that goes wrong in one place usually affects the entire structure**



The Monolith Architecture

- A monolithic architecture is comfortable for small teams to work with, which is why many startups choose this approach when building an app
- Components of monolithic software are interconnected and interdependent, which helps the software be self-contained
- This architecture is a traditional solution for building applications, but some developers find it outdated. However, we believe that a monolithic architecture is a perfect solution in some circumstances



Monolithic Architecture

The Monolith Architecture

- When developing a server-side application you can start it with a modular hexagonal or layered architecture which consists of different types of components:
  - **Presentation** — responsible for handling HTTP requests and responding with either HTML or JSON/XML (for web services APIs).
  - **Business logic** — the application's business logic.
  - **Database access** — data access objects responsible for access the database.
  - **Application integration** — integration with other services (e.g. via messaging or REST API).
- The code inside the monolith can follow software architecture and code organization patterns like model-view controller (MVC) or model view view model (MVVM).

Monolithic Architecture - Benefits

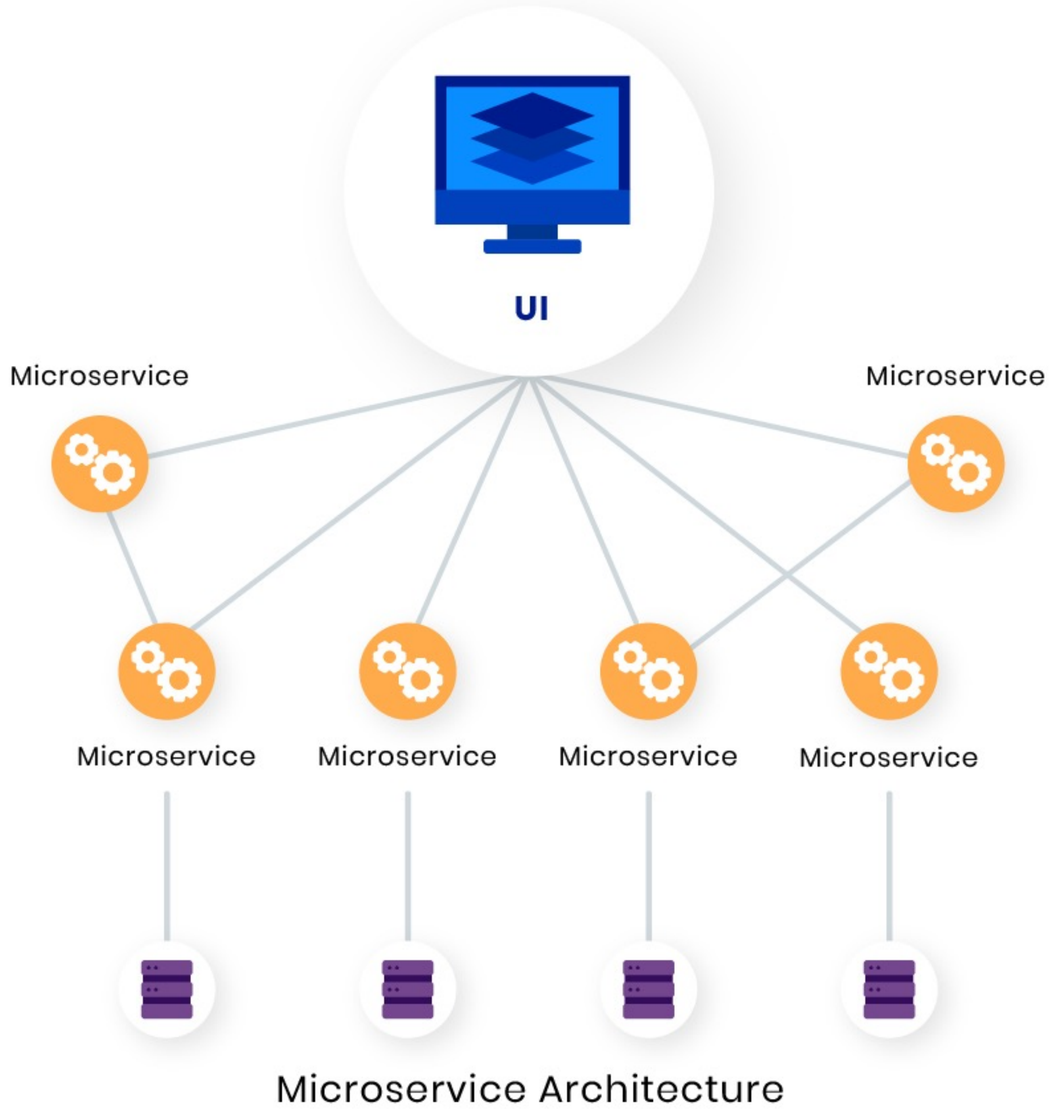
- Simple to **develop**.
- Simple to **test**. It is possible for example to end-to-end testing by simply launching the application and testing the UI
- Simple to **deploy**. You just have to copy the packaged application to a server.
- **Simple to scale horizontally by running multiple copies behind a load balancer**.
- Better performance (avoiding several communication among microservices)

Monolithic Architecture - Drawbacks

- This simple approach has a limitation in **size** and **complexity**.
- **Application is too large and complex** to fully understand and made changes fast and correctly.
- **The size of the application** can slow down the start-up time.
- You must **redeploy the entire application** on each update.
- **Impact of a change** is usually not very well understood which leads to do extensive manual testing.
- **Continuous deployment** is difficult.
- Monolithic applications can also be difficult to **scale** when different modules have conflicting resource requirements.
- Another problem with monolithic applications is **reliability**. Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the availability of the entire application.
- Monolithic applications has a **barrier to adopting new technologies**. Since changes in frameworks or languages will affect an entire application it is extremely expensive in both time and cost.

Microservice Architecture

- Microservice is a type of service-oriented software architecture that focuses on building a series of autonomous components that make up an app
- The idea is to split your application into a set of smaller, interconnected services instead of building a single monolithic application.
- Each microservice is a small application that has its own architecture consisting of business logic along with various adapters
- Some microservices would expose a REST, RPC or message-based API and most services consume APIs provided by other services. Other microservices might implement a web UI.



Microservice Approach - Definition

> "A particular way of designing software applications as suites of independently deployable services"
>
> — Martin Fowler (2014)

> The golden rule: can you make a change to a service and deploy it by itself without changing anything else?
>
> — Sam Newman, Building Microservices

Microservices without independent modules are just chaos.

Without independent components, we will create a highly coupled, low-cohesion **Frankenstein** that requires a lot of work to deliver, run, and maintain.

The main design goal is for each service to solve each business goal in the most optimal way

https://medium.com/swlh/monolithic-vs-micro-services-and-all-in-between-7d496408ad02

Microservice - Benefits

- It **tackles** the problem of **complexity** by **decomposing** application **into a set of manageable services** which are much **faster to develop**, and much **easier to understand and maintain**
- It enables **each service to be developed independently** by a team that is focused on that service.
- It **reduces barrier of adopting new technologies** since the developers are free to choose whatever technologies make sense for their service and not bounded to the choices made at the start of the project.
- Microservice architecture enables each microservice to be **deployed independently**. As a result, it makes **continuous deployment possible for complex applications**
- Microservice architecture enables **each service to be scaled independently**

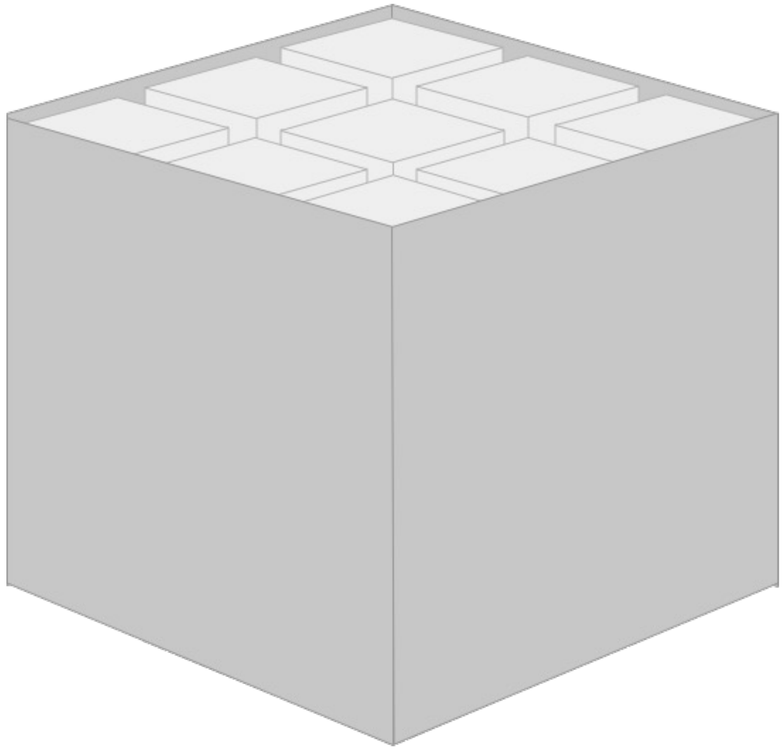
Microservice - Drawbacks

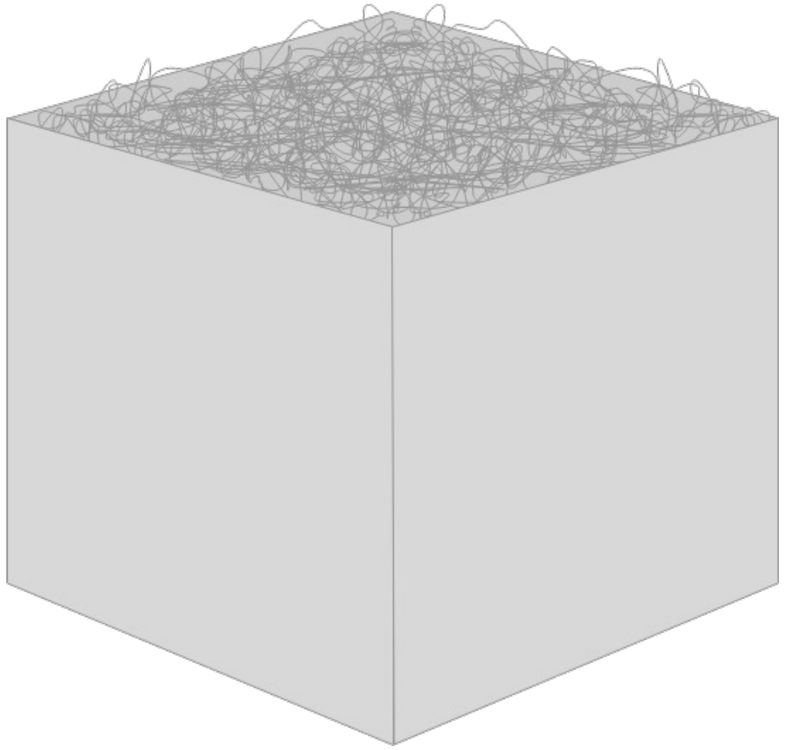
- Microservices architecture adding a complexity to the project just by the fact that a microservices application is a distributed system with communication requirements and error handling among services
- Microservices has the partitioned database architecture. Business transactions that update multiple business entities in a microservices-based application need to update multiple databases owned by different services
- Testing a microservices application is also much more complex then in case of monolithic application
- It is more difficult to implement changes that involves multiple services. You need to carefully plan and coordinate the rollout of changes to each of the services.
- Deploying a microservices-based application is also more complex. Microservice application typically consists of a large number of services. Each service will have multiple runtime instances. And each instance need to be configured, deployed, scaled, and monitored

How to start ? Monolith or Microservices ?

How to start ? Monolith or Microservices ?

How to start ? Monolith or Microservices ?

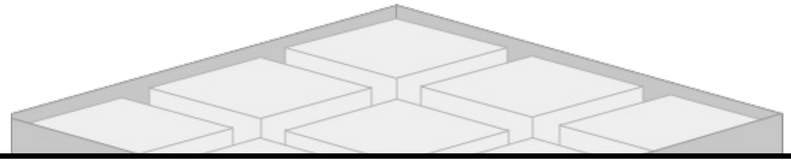




Hope vs. Reality

https://martinfowler.com/articles/dont-start-monolith.html

How to start ? Monolith or Microservices ?



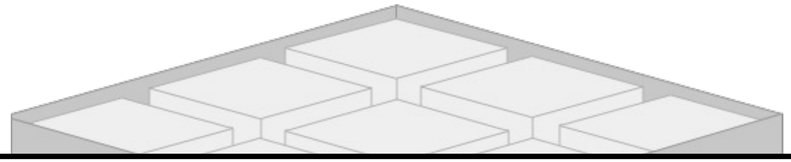
You may start designing a Monolith application with a set of nicely separated small services hiding in your bigger picture and just waiting to be extracted in order to move to a Microservices architecture.

In reality, it is extremely hard to avoid creating lots of connections, planned and unplanned.
The whole point of the microservices approach is to make it hard to create something like this.

Hope vs. Reality

https://martinfowler.com/articles/dont-start-monolith.html

How to start ? Monolith or Microservices ?



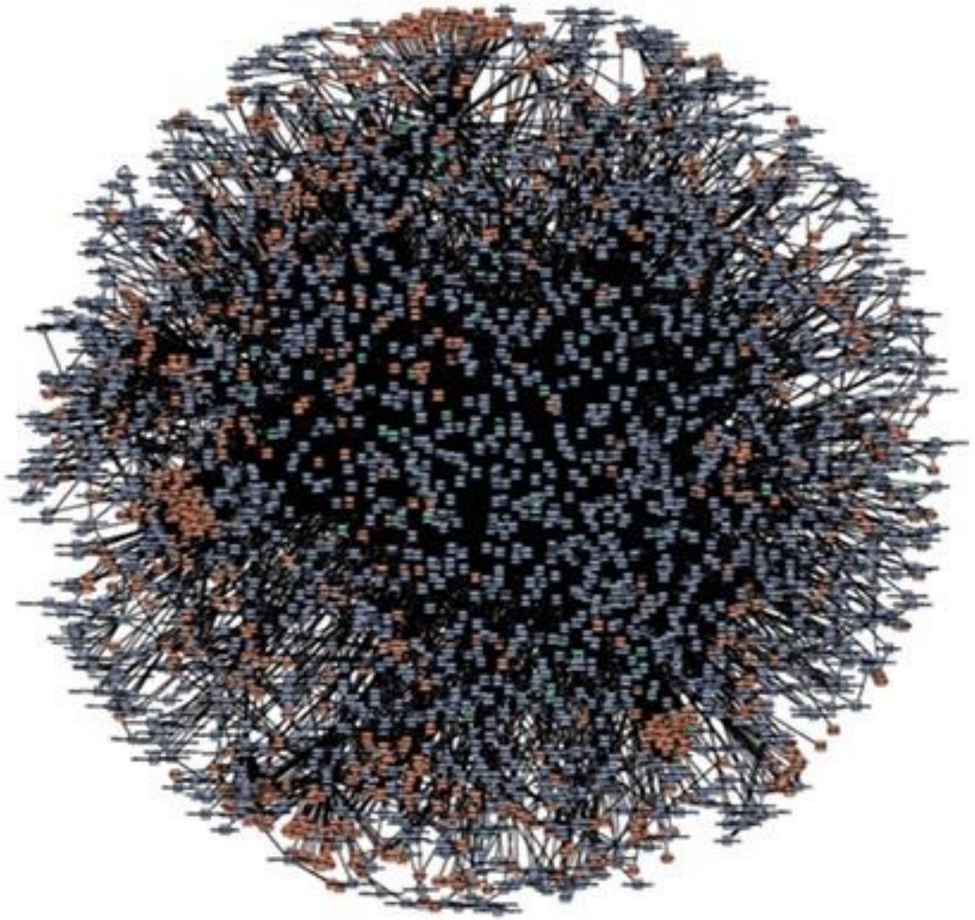
If it is reasonable (and the project large enough), when you start, you should think about the subsystems you build, and build them as independently of each other as possible

Starting since the beginning with an approach where you split your system into smaller parts, and treat each of them as a clearly separated (with their own development, deployment, and delivery cycle, and their own internal architecture), is a very powerful concept that can help you to deliver, maintaining and extend a system.

Hope vs. Reality

https://martinfowler.com/articles/dont-start-monolith.html

Some Complex Microservices Examples

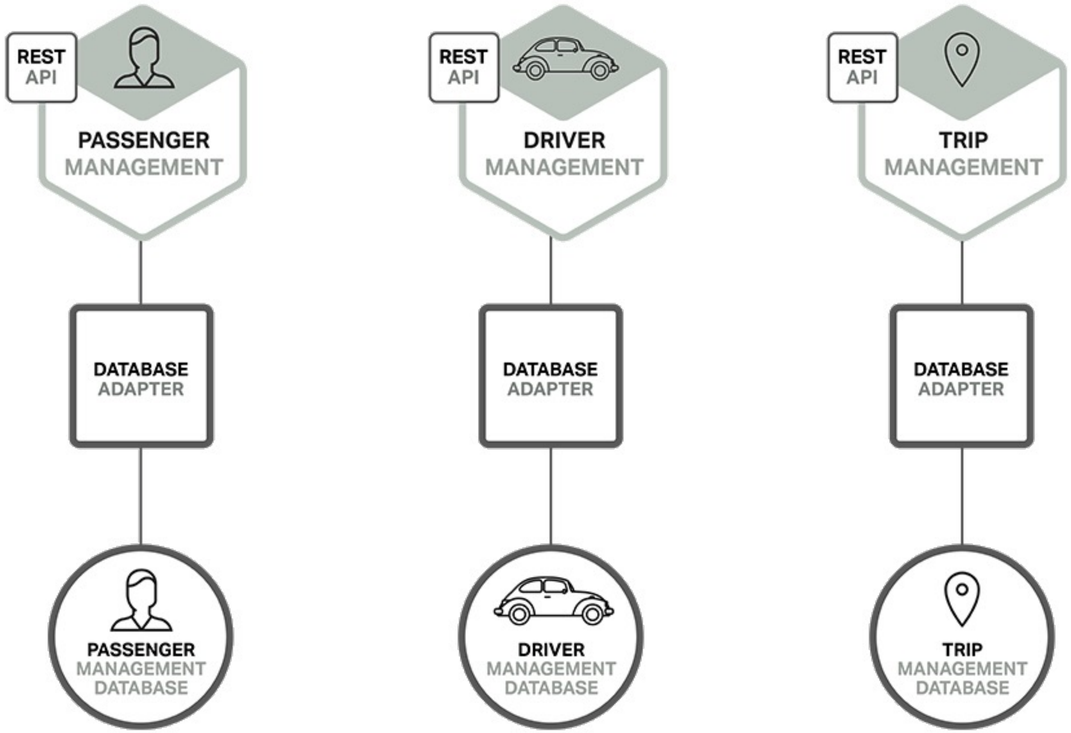


amazon.com

NETFLIX

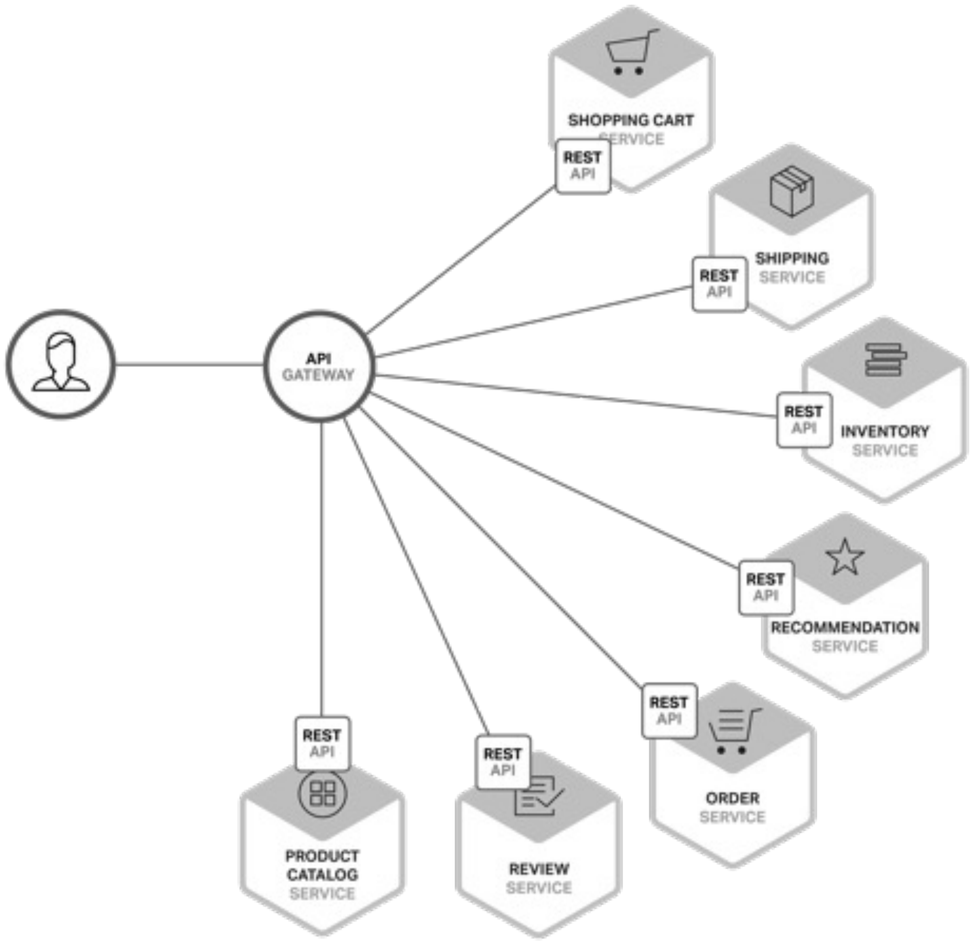
Microservices & Data Management

- The Microservices Architecture pattern significantly impacts the relationship between the application and the database
- Instead of sharing a single database schema with other services, each service has its own database schema
  - This approach is at odds with the idea of an enterprise-wide data model. Also, it often results in duplication of some data
  - Having a database schema per service is essential if you want to benefit from microservices, because it ensures loose coupling



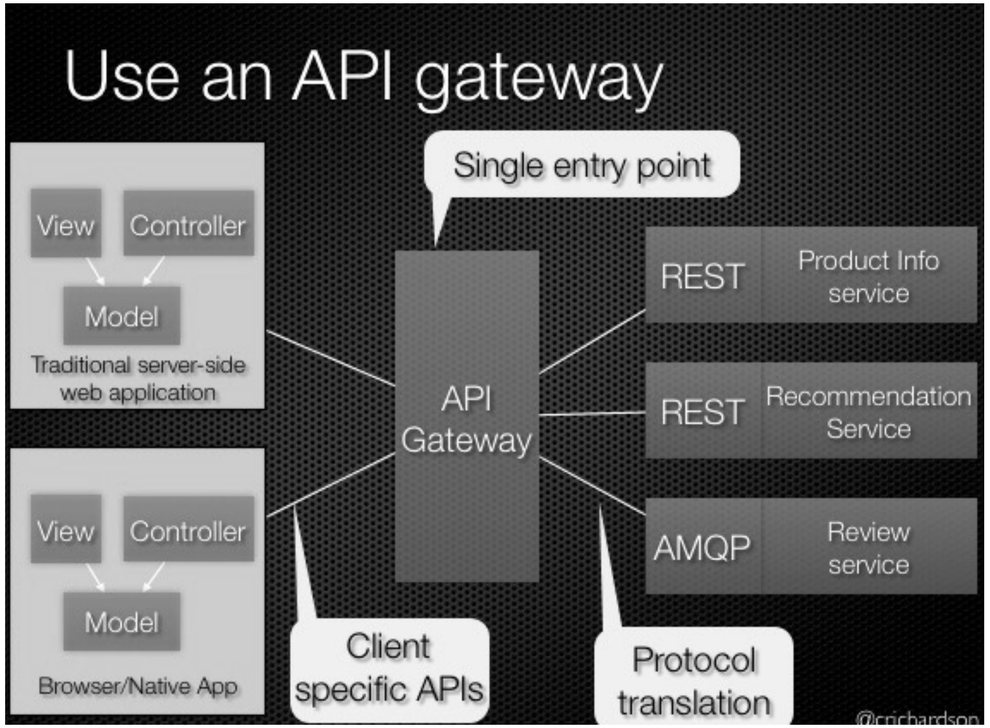
API Gateway Pattern

- Some Microservice and APIs may be exposed to the external world and in particular to external clients such as mobile applications, desktop clients and web apps
- These apps do not have a direct access to the back-end services and microservices
- Communication is mediated by an intermediary known as an API Gateway.
- The API Gateway is responsible for tasks such as load balancing, caching, access control, API metering, and monitoring



API Gateway Pattern

- Implement an API gateway that is the single entry point for all clients
- The API gateway handles requests in one of two ways
  - Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services
  - the API gateway can expose a different API for each client. (the Netflix API gateway runs client-specific adapter code that provides each client with an API that's best suited to its requirement).
- The API gateway might also implement security, e.g. verify that the client is authorized to perform the request



https://medium.com/netflix-techblog/embracing-the-differences-inside-the-netflix-api-redesign-15fd8b3dc49d

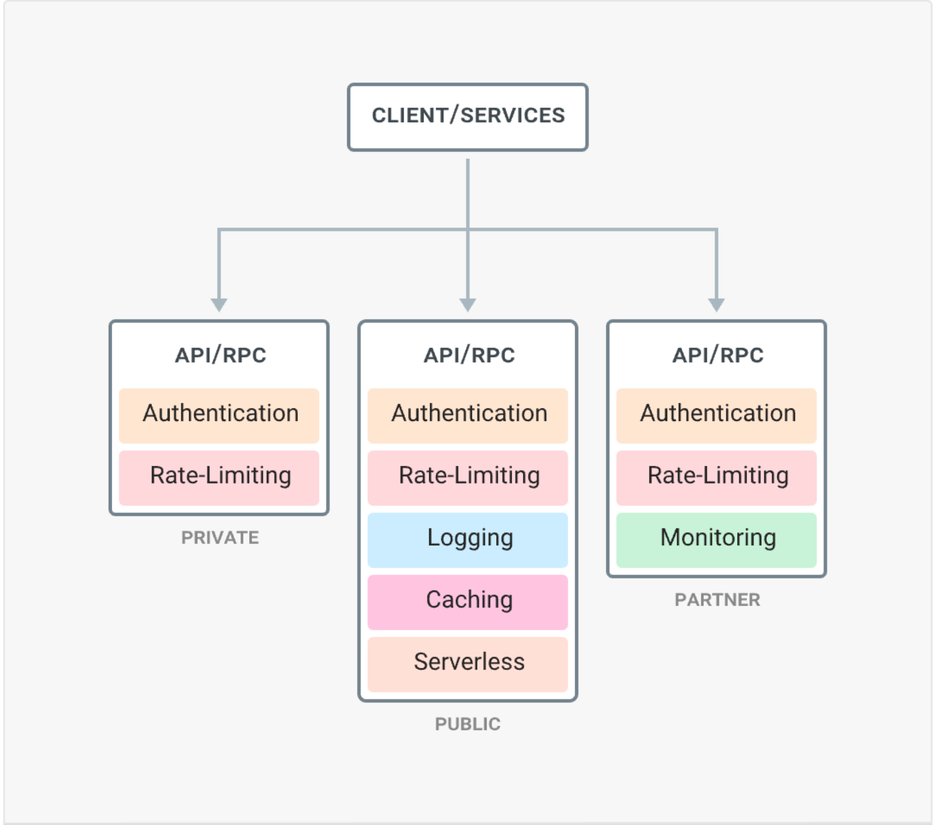
API Gateway Pattern - Benefits

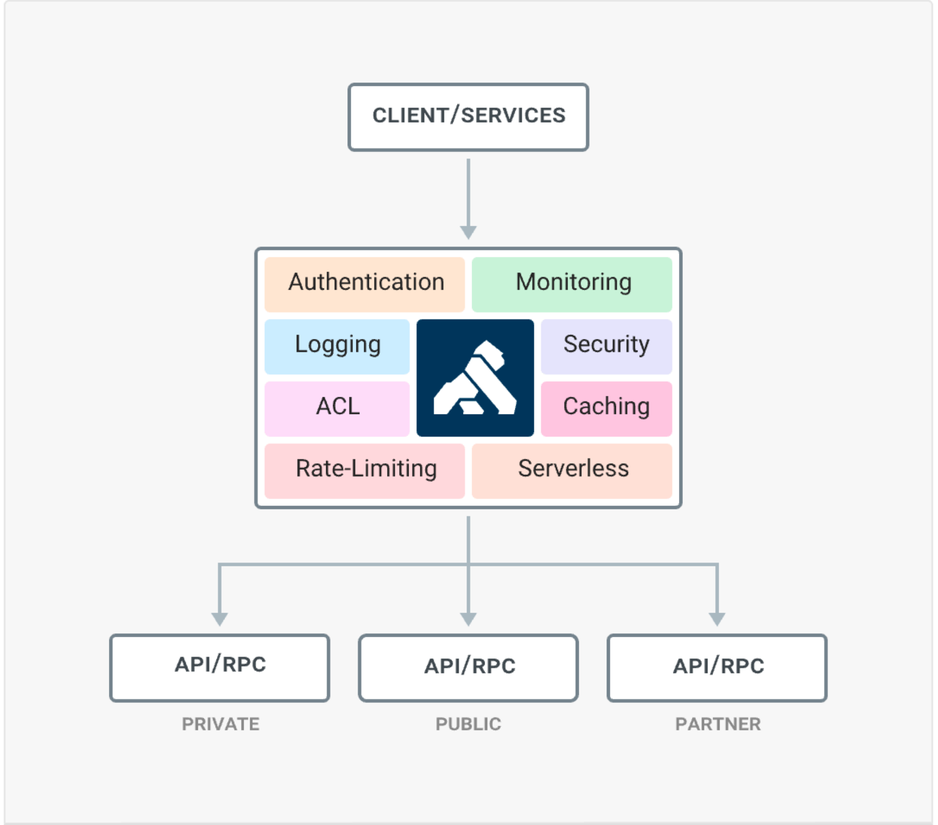
- Insulates the clients from how the application is partitioned into microservices
- Insulates the clients from the problem of determining the locations of service instances
- Provides the optimal API for each client
- Reduces the number of requests/roundtrips
  - For example, the API gateway enables clients to retrieve data from multiple services with a single round-trip. Fewer requests also means less overhead and improves the user experience. An API gateway is essential for mobile applications.
- Simplifies the client by moving logic for calling multiple services from the client to API gateway
- Translates from a “standard” public web-friendly API protocol to whatever protocols are used internally

API Gateway Pattern - Drawbacks & Notes

- Increased complexity - the API gateway is yet another moving part that must be developed, deployed and managed
- Increased response time due to the additional network hop through the API gateway - however, for most applications the cost of an extra roundtrip is insignificant.
- How implement the API gateway ?
- Is it possible to use an existing API gateway ?
- Do I need a custom API Gateway for my Business Logic ?

API Gateway Pattern - Kong Example

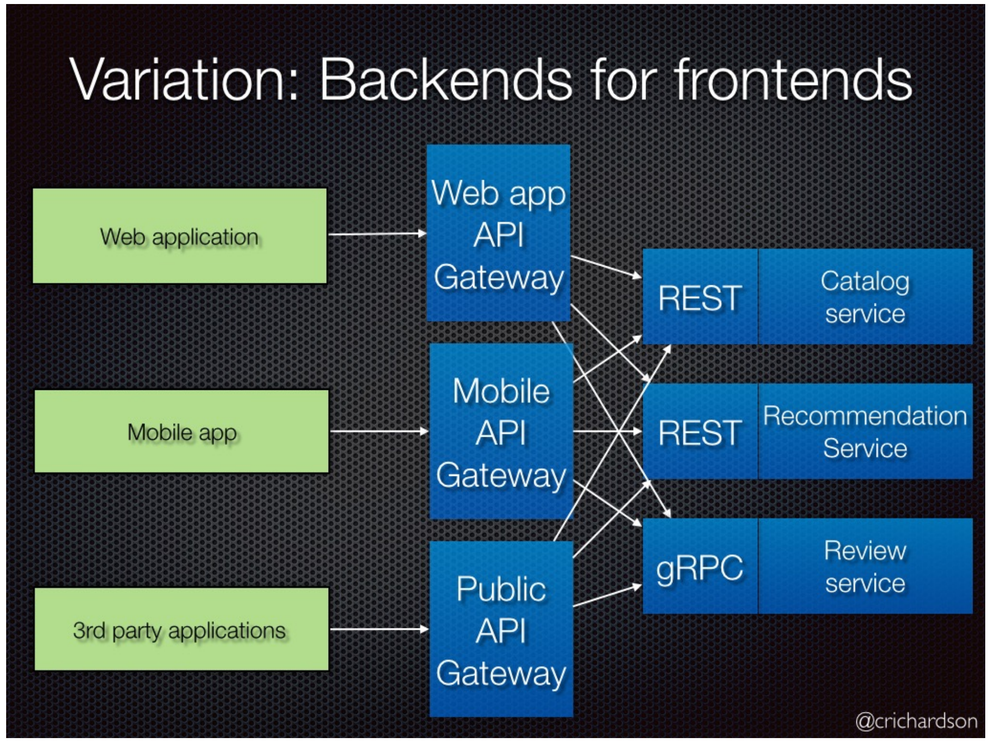




✗ Common functionality is duplicated across multiple services	✓ Kong orchestrates common functionality
✗ Systems tend to be monolithic and hard to maintain	✓ Build efficient distributed architectures ready to scale
✗ Difficult to expand without impacting other services	✓ Expand functionality from one place with a simple command
✗ Productivity is inefficient because of system constraints	✓ Focus on your product and let Kong do the REST

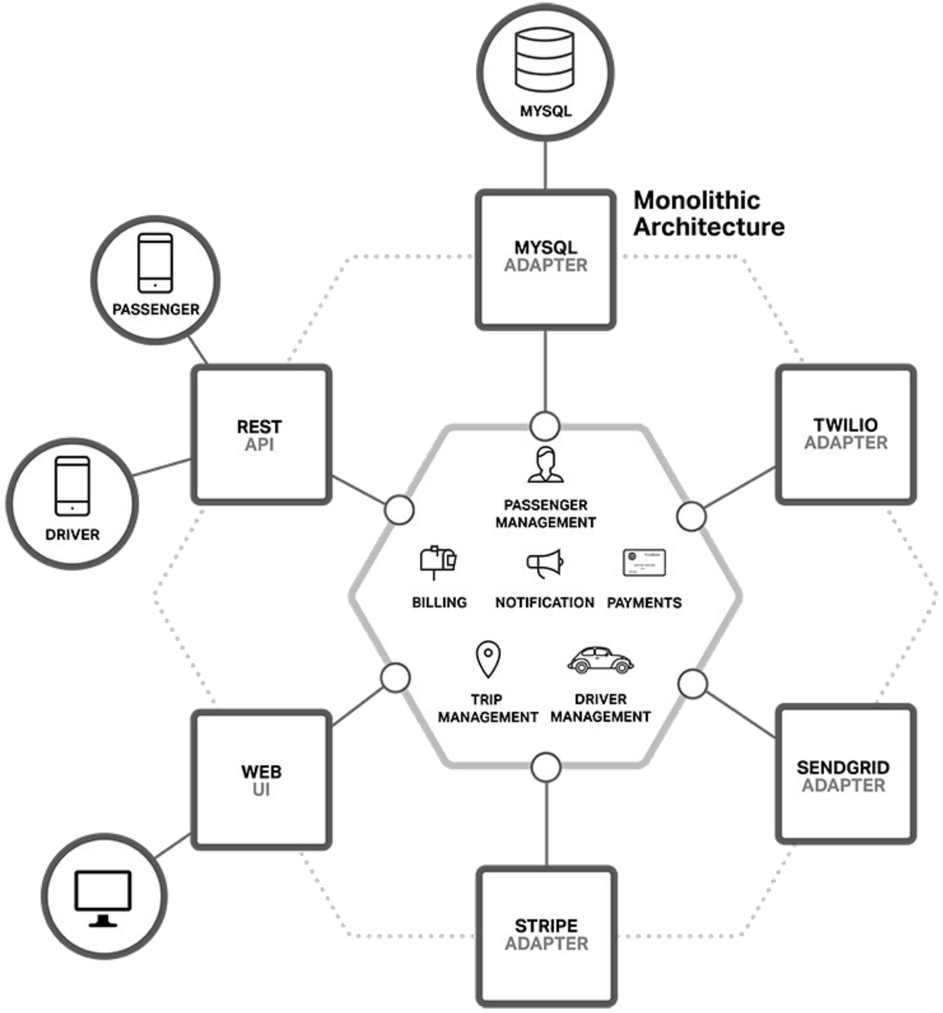
API Gateway Pattern - Backends for Frontends

- A variation of this pattern is the Backends for frontends pattern. It defines a separate API gateway for each kind of client.
- In the target example we have:
  - web application
  - mobile application
  - external 3rd party application.
- There are three different API gateways. Each one is provides an API for its client.



https://medium.com/netflix-techblog/optimizing-the-netflix-api-5c9ac715cf19

Monolith vs Microservices - An Example



- We are creating a new taxi application in order to compete with Uber :)
- At the core of the application is the business logic, which is implemented by modules that define services, domain objects, and events
- Surrounding the core are adapters that interface with the external world (e.g., database access, messaging, web components, API etc …)

https://www.nginx.com/blog/introduction-to-microservices/

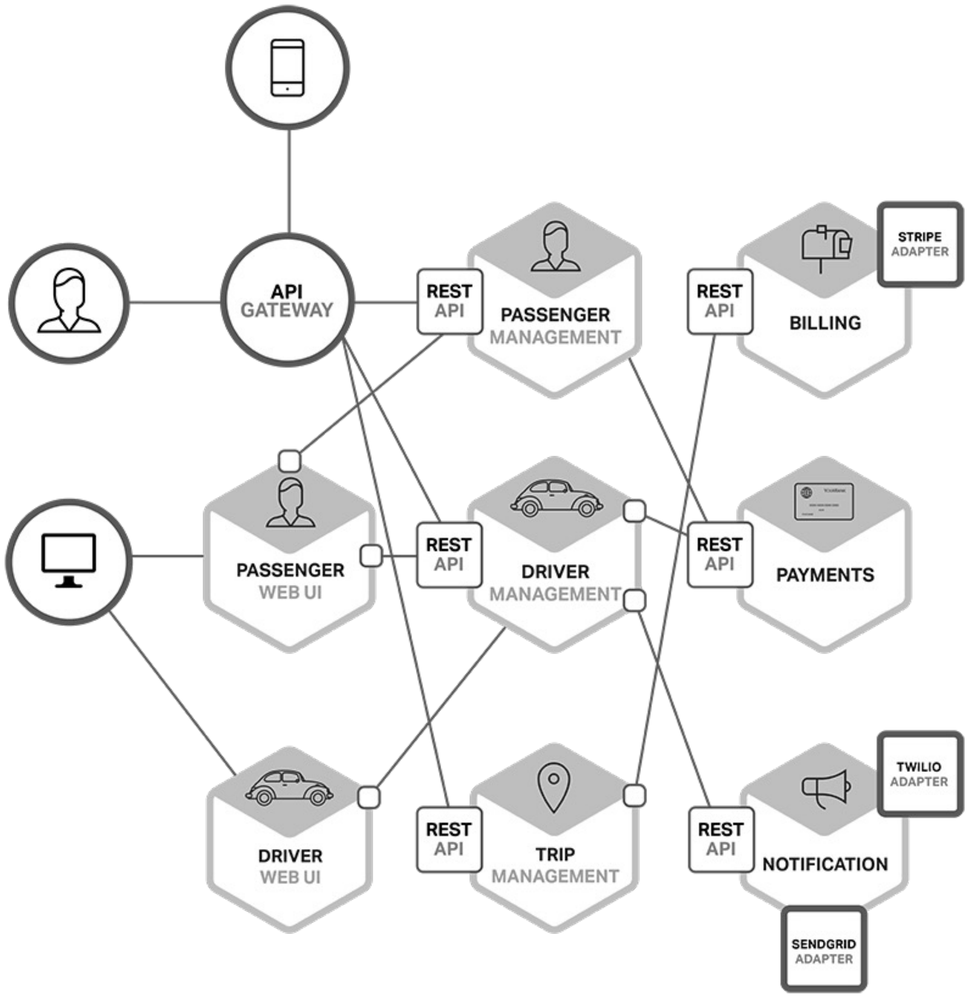
Monolith vs Microservices - An Example

- Monolithic Application Design are extremely common and good is some specific cases
- They are simple to develop since you are focused on building a single application. At the same time they are also simple to test with an end-to-end test suite
- Monolithic applications are also simple to deploy. You just have to copy the packaged application to a server. You can also scale the application by running multiple copies behind a load balancer. **In the early stages of the project it works well**
- Unfortunately, this simple approach has a huge limitation. Successful applications have a habit of growing over time and eventually becoming huge. **After a few years, your small, simple application will have grown into a monstrous monolith**

Monolith vs Microservices - An Example

- Once your application has become a large, complex monolith, your development organization is probably in a world of pain
- Any attempts at agile development and delivery will flounder. One major problem is that the application is overwhelmingly complex
- It’s simply too large for any single developer to fully understand. As a result, fixing bugs and implementing new features correctly becomes difficult and time consuming
- If the codebase is difficult to understand, then changes won’t be made correctly
- Another problem with a large, complex monolithic application is that it is an obstacle to continuous deployment. Today, the state of the art for SaaS applications is to push changes into production many times a day. This is extremely difficult to do with a complex monolith since you must redeploy the entire application in order to update any one part of it

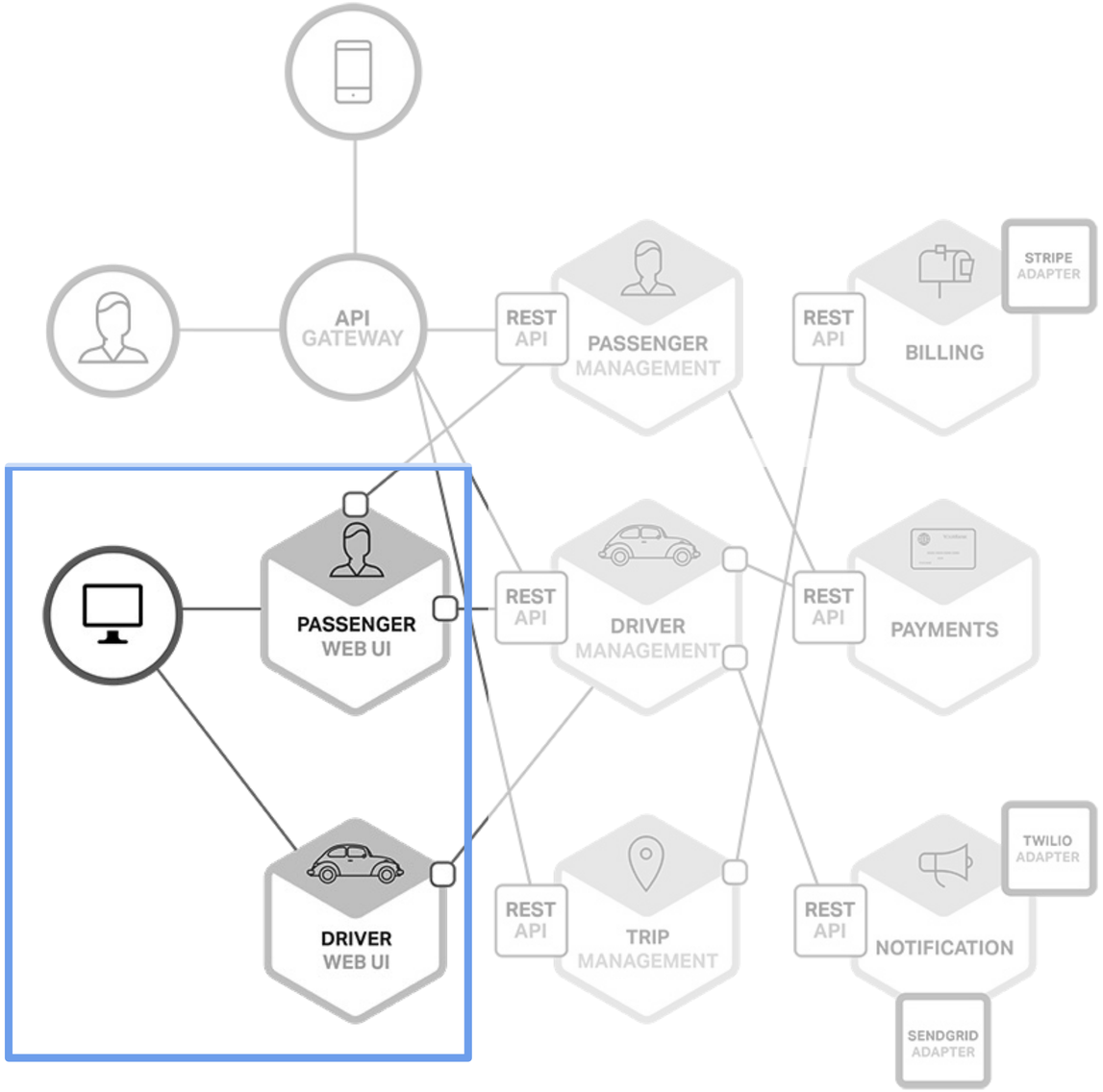
Monolith vs Microservices - An Example



- Instead of building a single monstrous, monolithic application, the idea is to split your application into set of smaller, interconnected services
- A service typically implements a set of distinct features or functionality (e.g., order management, customer management, etc.)
- Each microservice is a mini-application that has its own architecture consisting of business logic along with various adapters
- Some microservices would expose an API that’s consumed by other microservices or by the application’s clients. Other microservices might implement a web UI. At runtime, each instance is often a cloud VM or a Docker container
- Each functional area of the application is now implemented by its own microservice.

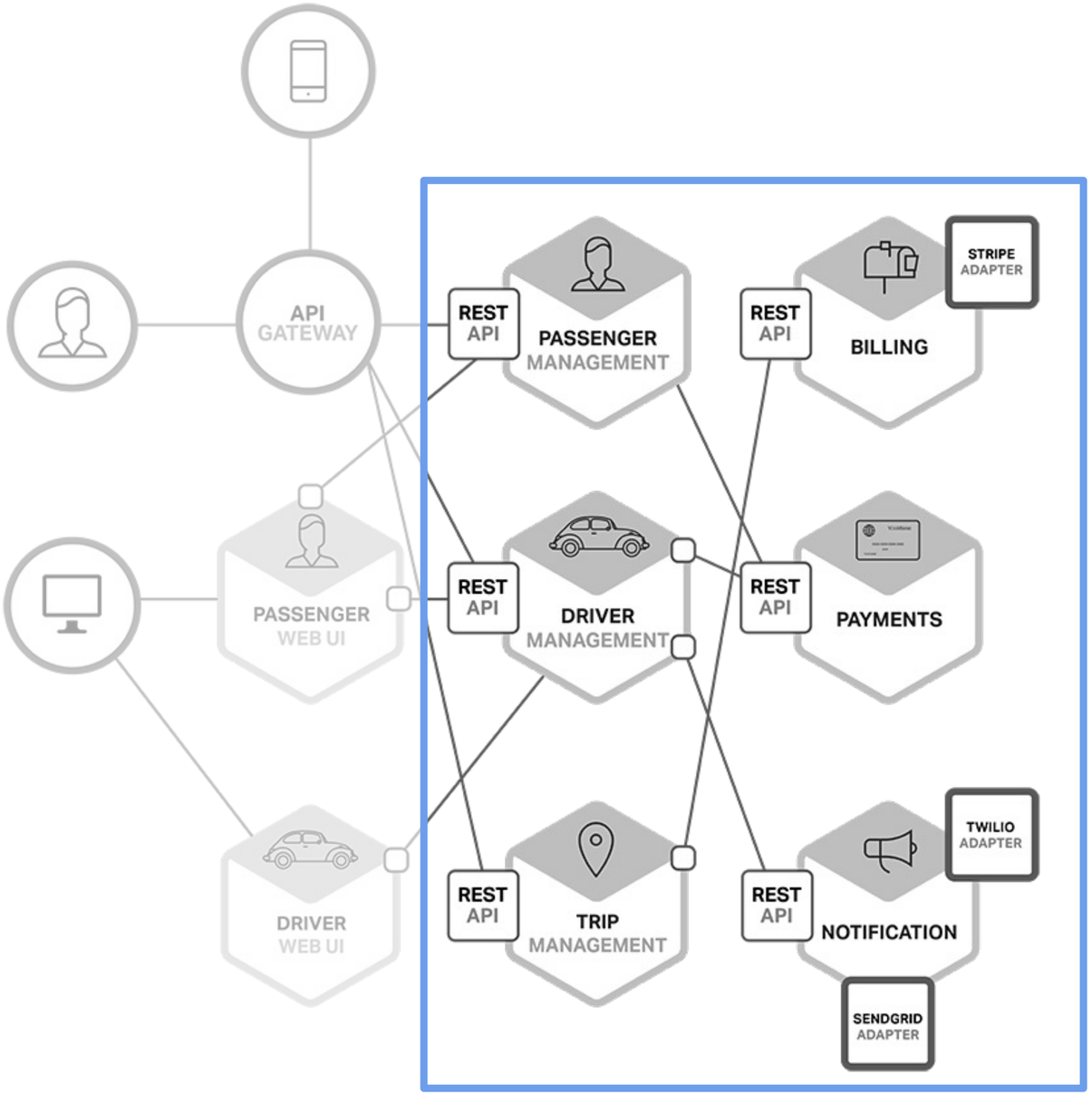
Monolith vs Microservices - An Example

- Web application is split into a set of simpler web applications. Such as one for the current example:
  - Passengers Web UI
  - Drivers Web UI
- This makes it easier to deploy distinct experiences for specific users, devices, or specialized use cases



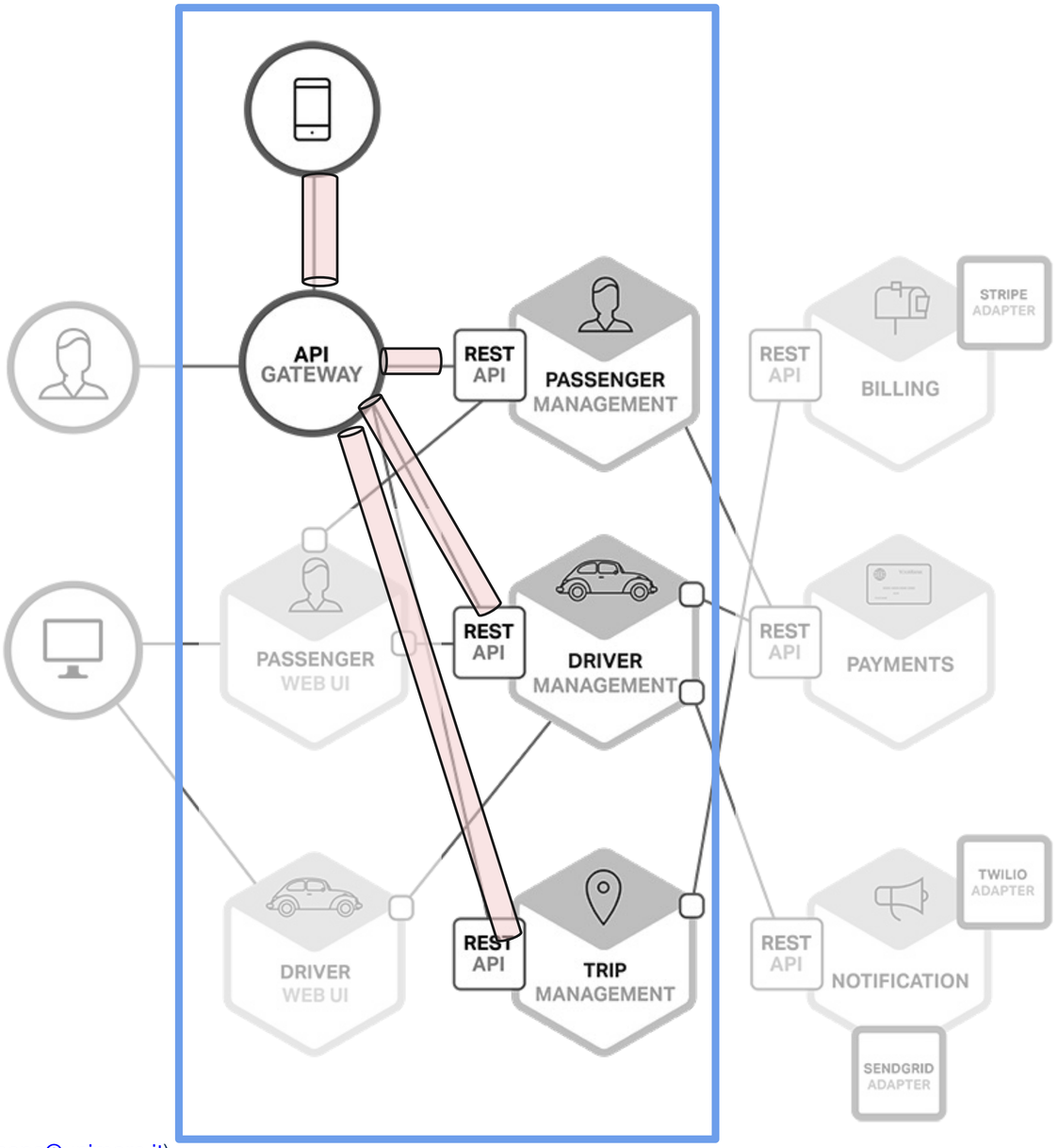
Monolith vs Microservices - An Example

- Each backend service exposes a REST API and most services consume APIs provided by other services. For example:
  - Driver Management uses the Notification server to tell an available driver about a potential trip
  - The UI services invoke the other services in order to render web pages



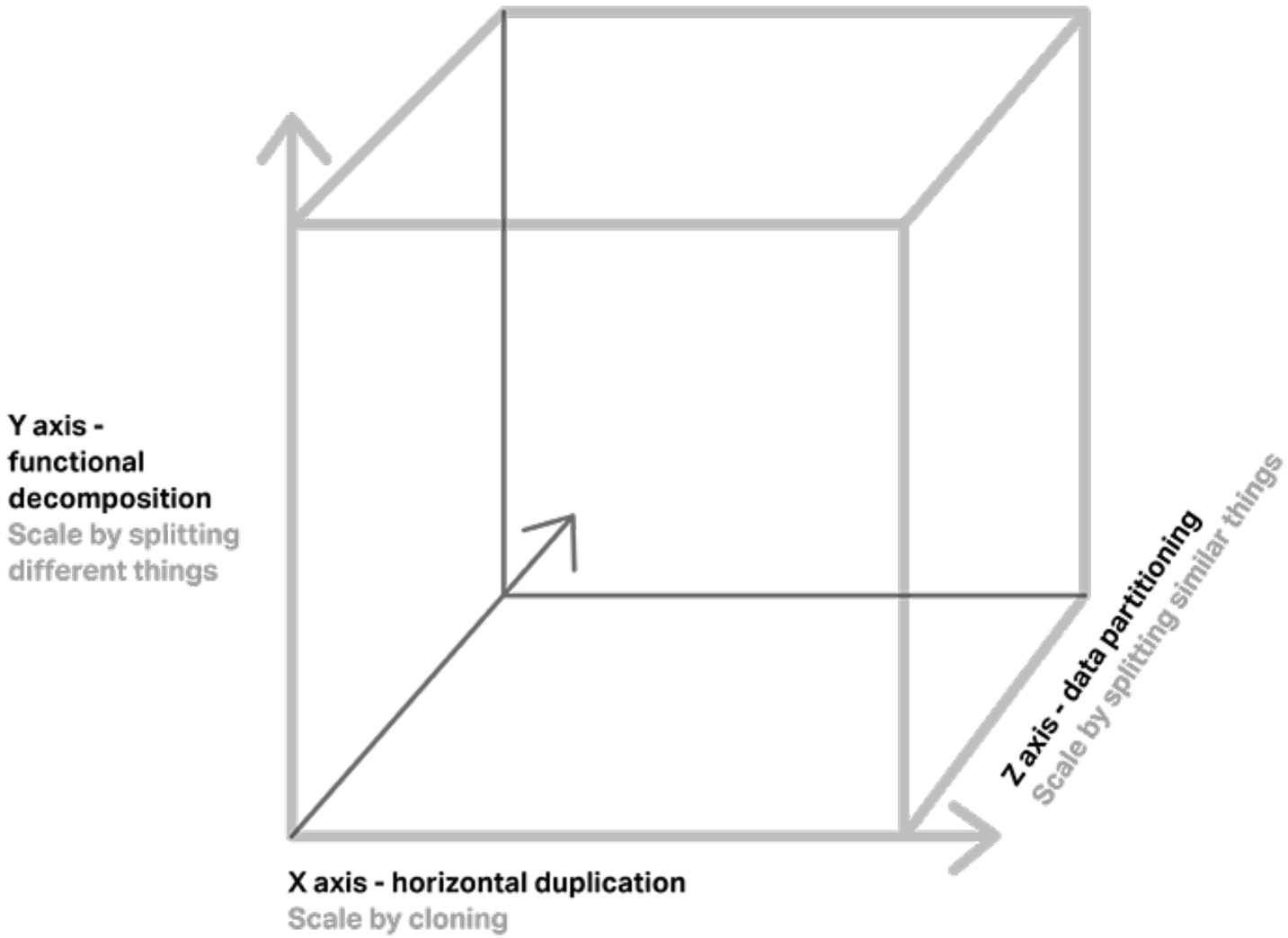
Monolith vs Microservices - An Example

- Some REST APIs are also exposed to the mobile apps used by the drivers and passengers.
  - The apps don’t, however, have direct access to the backend services
  - Communication is mediated by an intermediary API Gateway.
- Services might also use asynchronous, message-based communication. Inter-service communication will bring significant value to a distributed Microservice Architecture



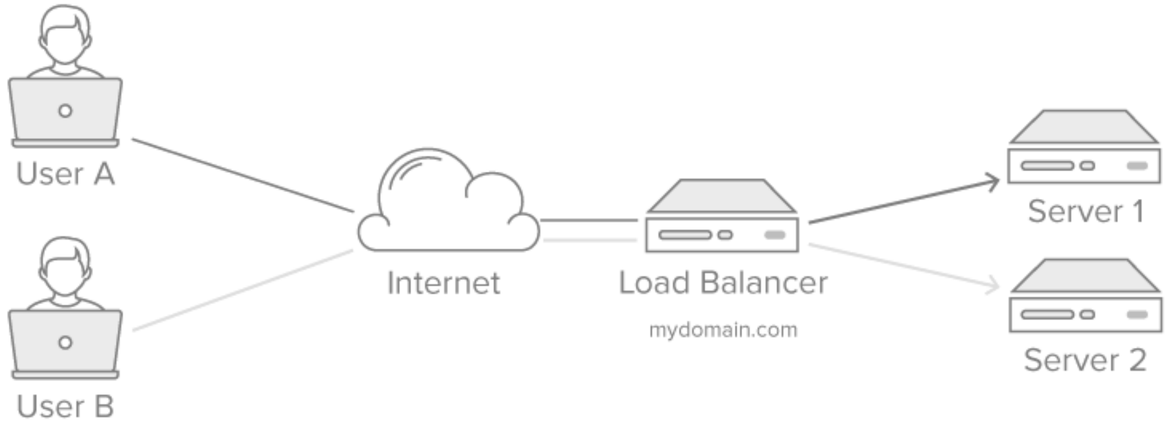
Microservices & Scalability

- We have different type of Scalability
  - (X) Horizontal Duplication - Scale by cloning
  - (Y) Functional Decomposition - Scale by splitting different things
  - (Z) Data Partitioning - Scale by splitting similar things
- **The Microservices Architecture pattern corresponds to the Y-axis scaling of the Scale Cube** [15], which is a 3D model of scalability from the excellent book The Art of Scalability [16]



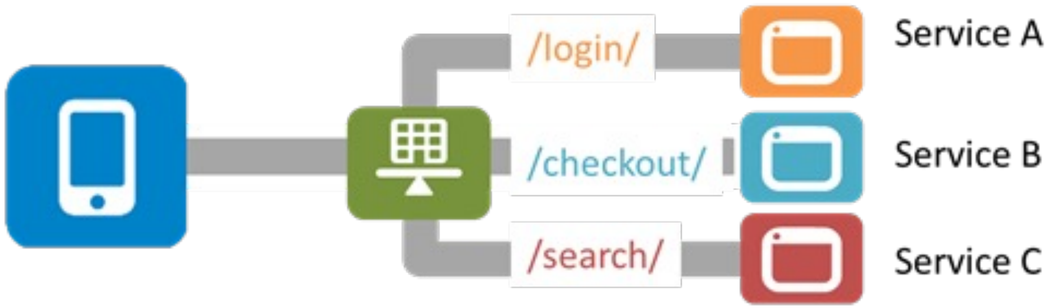
X-Axis Scaling

- X-axis scaling consists of running multiple copies of an application behind a load balancer. If there are N copies then each copy handles 1/N of the load. This is a simple, commonly used approach of scaling an application.
- One drawback of this approach is that because each copy potentially accesses all of the data, caches require more memory to be effective. Another problem with this approach is that it does not tackle the problems of increasing development and application complexity.

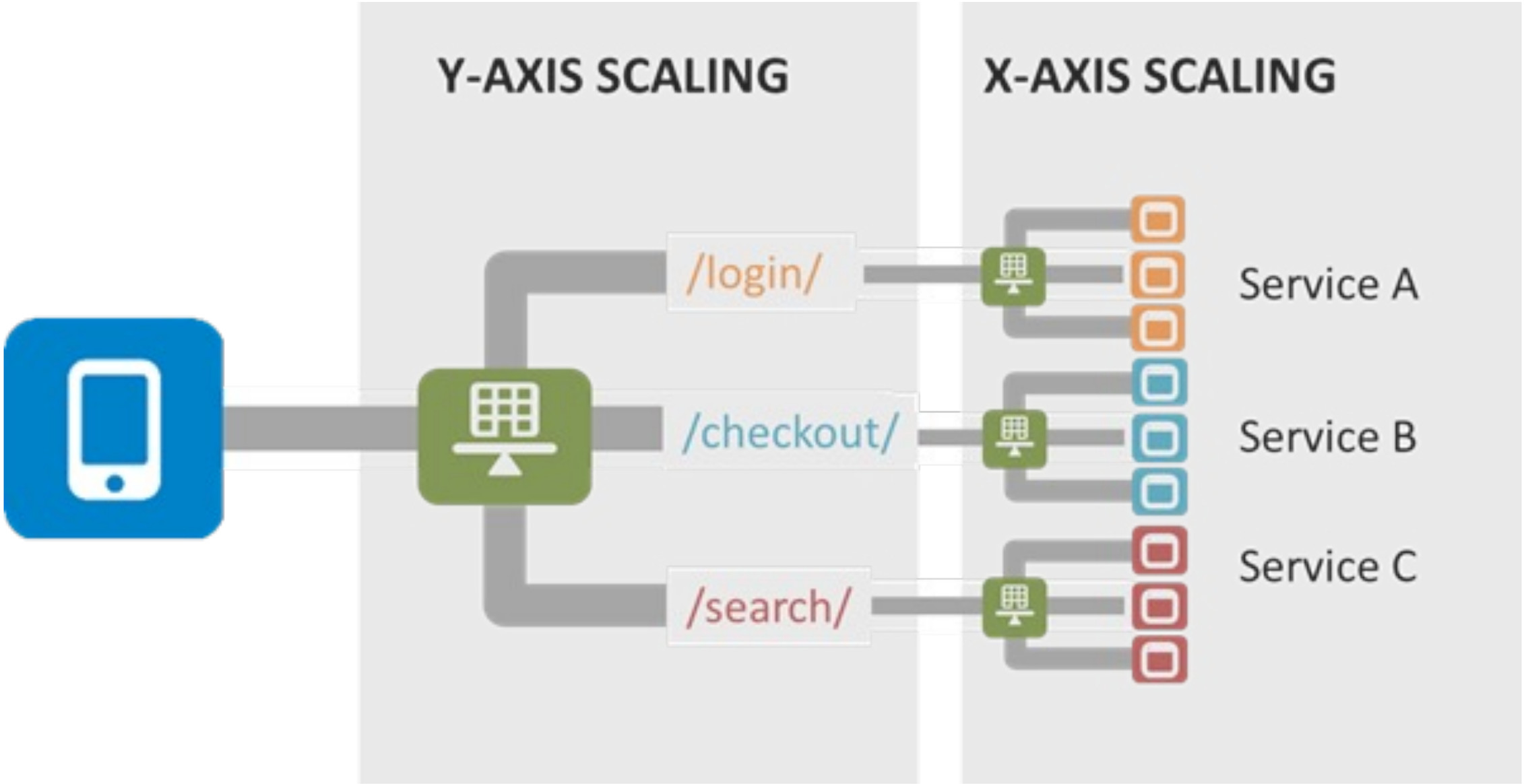


Y-Axis Scaling

- Y-axis axis scaling splits the application into multiple, different services.
- Each service is responsible for one or more closely related functions.
- There are a couple of different ways of decomposing the application into services:
  - One approach is to use verb-based decomposition and define services that implement a single use case such as “**checkout**”.
  - The other option is to decompose the application by noun and create services responsible for all operations related to a particular entity such as “**customer management**”.
  - An application might use a combination of verb-based and noun-based decomposition.



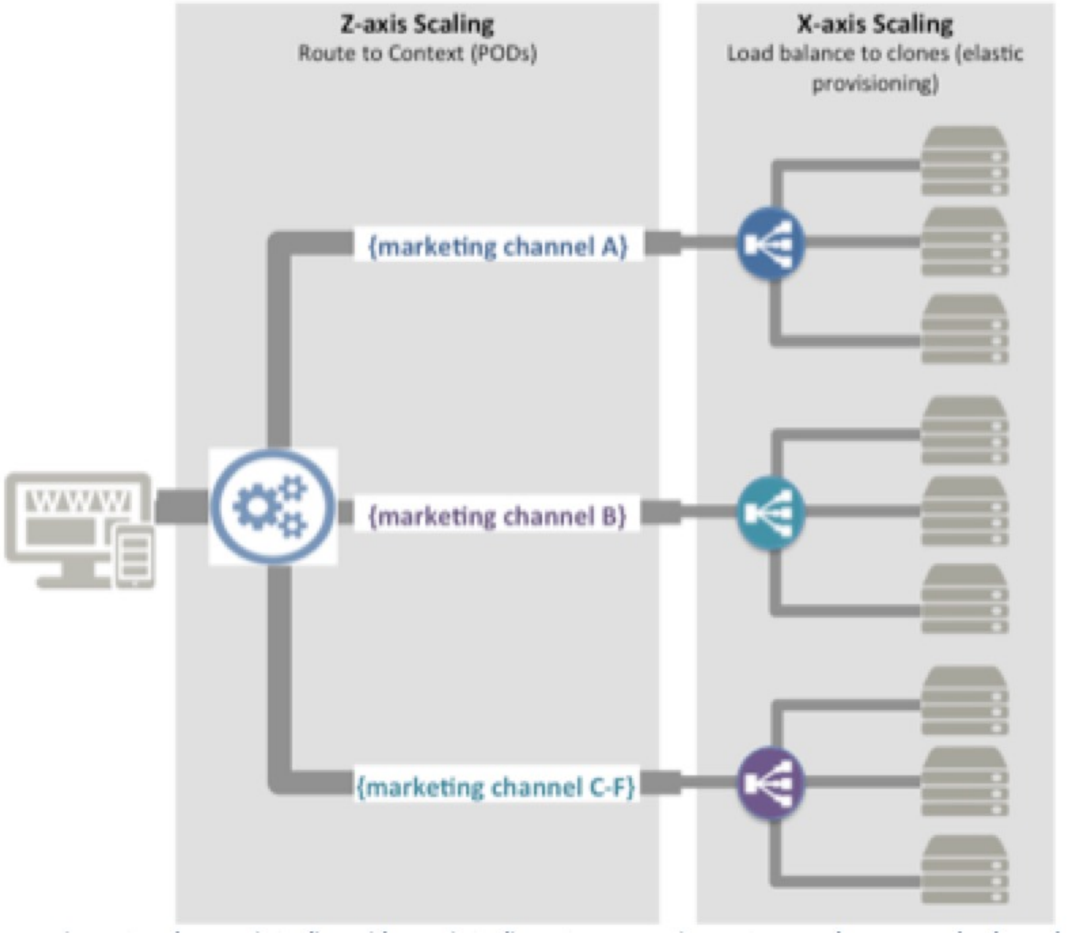
Y-Axis & X-Axis Scaling

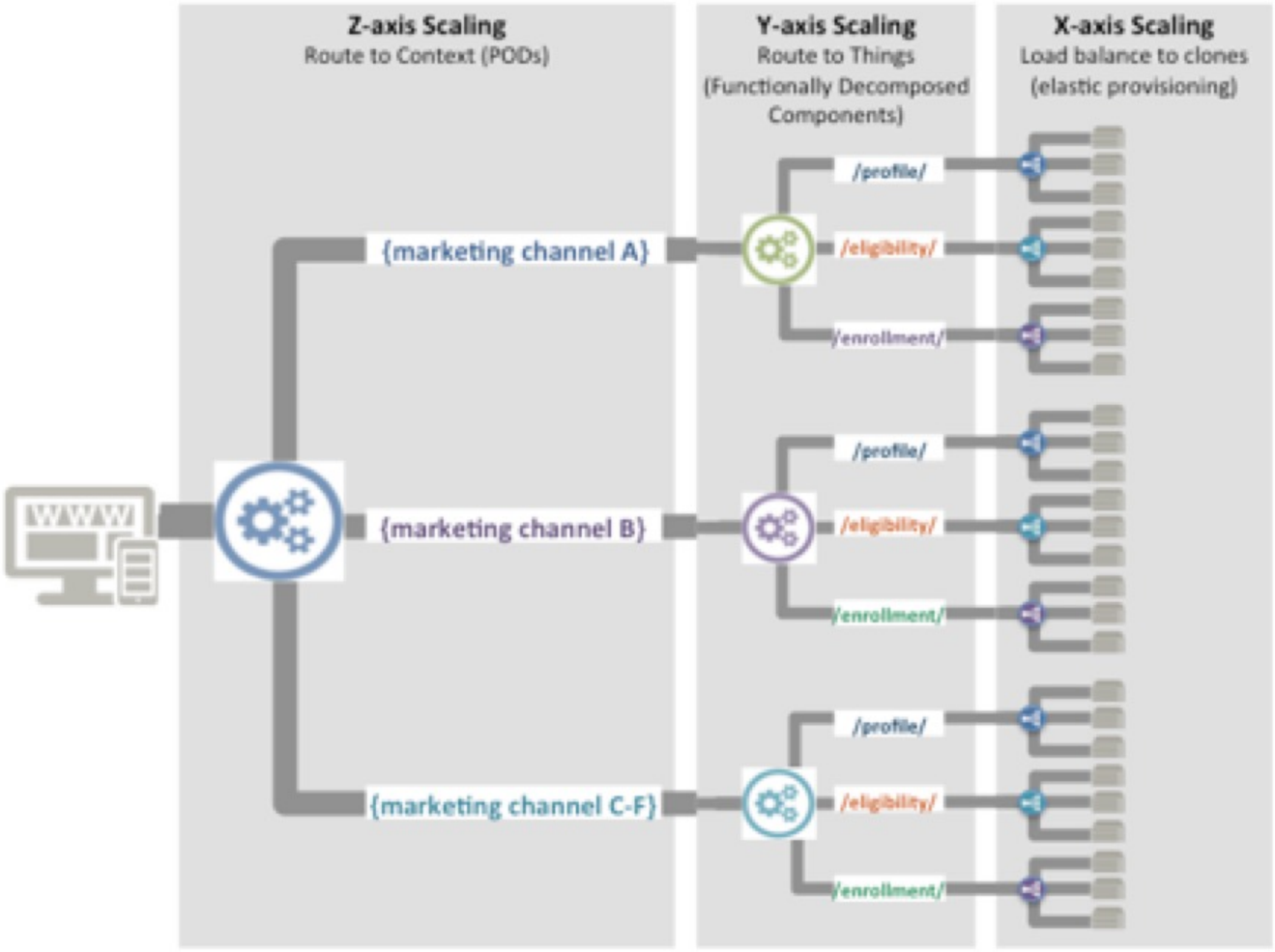


Z-Axis Scaling

- When using Z-axis scaling each server runs an identical copy of the code
- It’s similar to X-axis scaling. The big difference is that each server is responsible for only a subset of the data. Some component of the system is responsible for routing each request to the appropriate server.
- One commonly used routing criteria is an attribute of the request such as the primary key of the entity being accessed. Another common routing criteria is the customer type.
- For example, an application might provide paying customers with a higher SLA than free customers by routing their requests to a different set of servers with more capacity.
- Z-axis splits are commonly used to scale databases. Data is partitioned (a.k.a.

Z-Axis Scaling



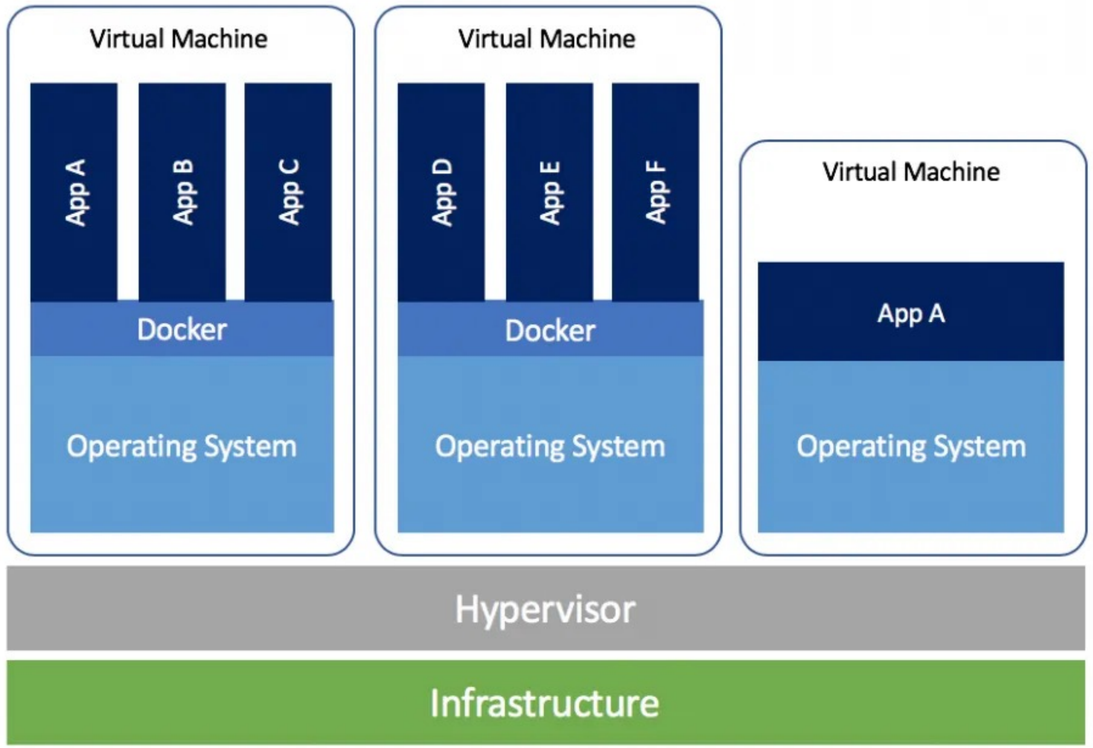


Z-Axis Scaling - Comments

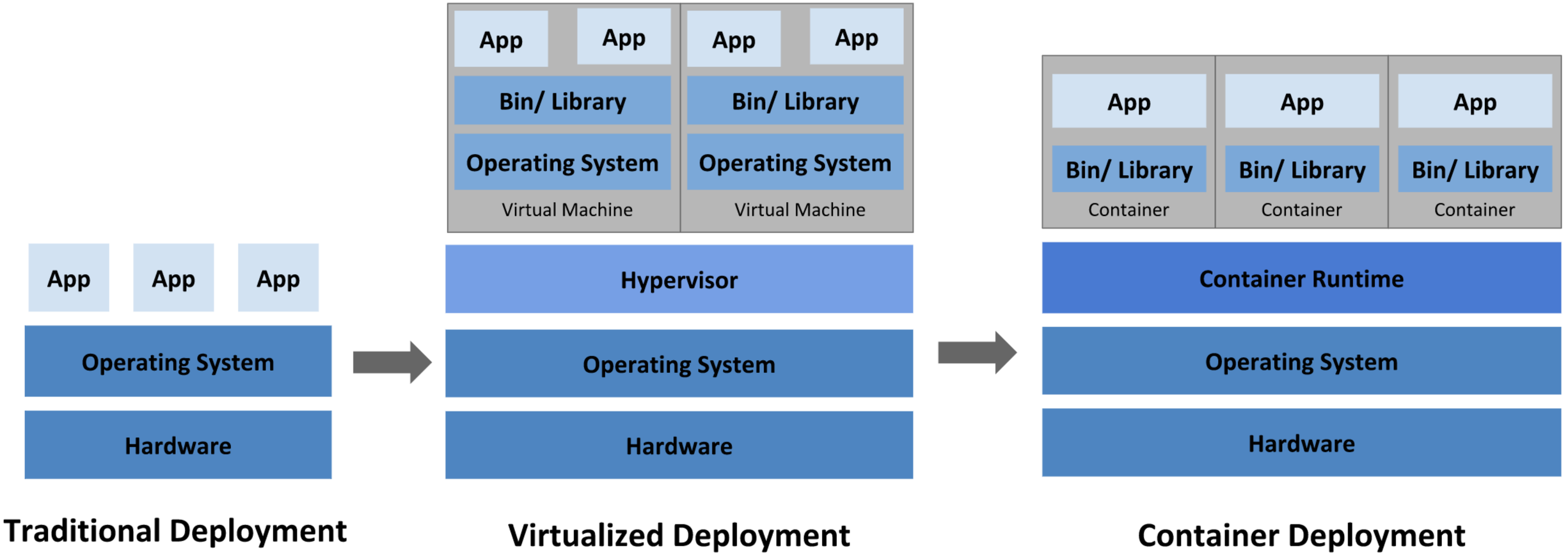
- Z-axis scaling has a number of benefits
  - Each server only deals with a subset of the data.
  - This improves cache utilization and reduces memory usage and I/O traffic.
  - It also improves transaction scalability since requests are typically distributed across multiple servers.
  - Also, Z-axis scaling improves fault isolation since a failure only makes part of the data in accessible.
- Z-axis scaling has some drawbacks
  - Increased application complexity.
  - We need to implement a partitioning scheme, which can be tricky especially if we ever need to repartition the data.
- It doesn’t solve the problems of increasing development and application complexity. To solve those problems we need to apply Y-axis scaling

Microservices - Virtual Machine & Containers

- Virtual Machines offer virtualization of hardware as well as the OS and create an efficient, isolated duplicate of a real machine
- In the case of containers (e.g., based on Docker), only the OS is virtualized and not the hardware, creating a lightweight environment that houses the application and the dependent assets
- When microservices come into the picture, the question—“containerize or virtualize?”



Microservices - Virtual Machine & Containers



Containers - Lowered Cost & Efficiency

- VMs partition execution environments among microservices by virtualizing hardware as well as operating systems. Running every microservice through a separate VM requires the replication of all the OS and Hardware as well.
- It is possible to execute multiple microservices in a single VM, but that would reduce the single biggest advantage of breaking down a monolithic application into small easily executable microservices. The issue of conflicting libraries and application components will remain unsolved.
- Containers offer isolation at the OS level. A single operating system can support multiple containers, each running within its own, separate execution environment. By running multiple components on a single operating system you reduce overhead costs

Microservices - Containers - Flexible Storage

- **VMs** have **multiple options for storage**. They can either have a **local or a network-based storage** type.
- There will always be a physical disk space allocated to every VM which is isolated from the VM itself. All the times, **the operating system, program files and data of a VM occupy storage space on the dedicated storage disk—stateful storage**.
- **Containers**, offer the choice of being **stateful** or **stateless**. **Storage space is created or occupied when the container is created and discarded along with its deletion**.
- A **sandbox layer** is created when a container image is edited and that stores all the data. This layer is active only as part of the container. **Each service of your microservices app can have its own storage, that can be managed differently from the storage of other services providing more flexibility and control**.

Microservices - Containers - Better Isolation

- **Containers** offer **independent execution** environments to microservices while enabling cohabitation on a single operating system.
- The databases, as well as the environments of microservices, are completely independent, despite being deployed on a single OS. If one tries to run multiple microservices on a single VM, there can be overlapping environments which may end up clogging up the server.
- Using containers, with smart workload allocation, app developers can efficiently utilize server resources, ensuring resources is not left unused (example, some microservices require a lot of processing power, while others may generate a lot of network).

Microservices - Containers - Reduced Size

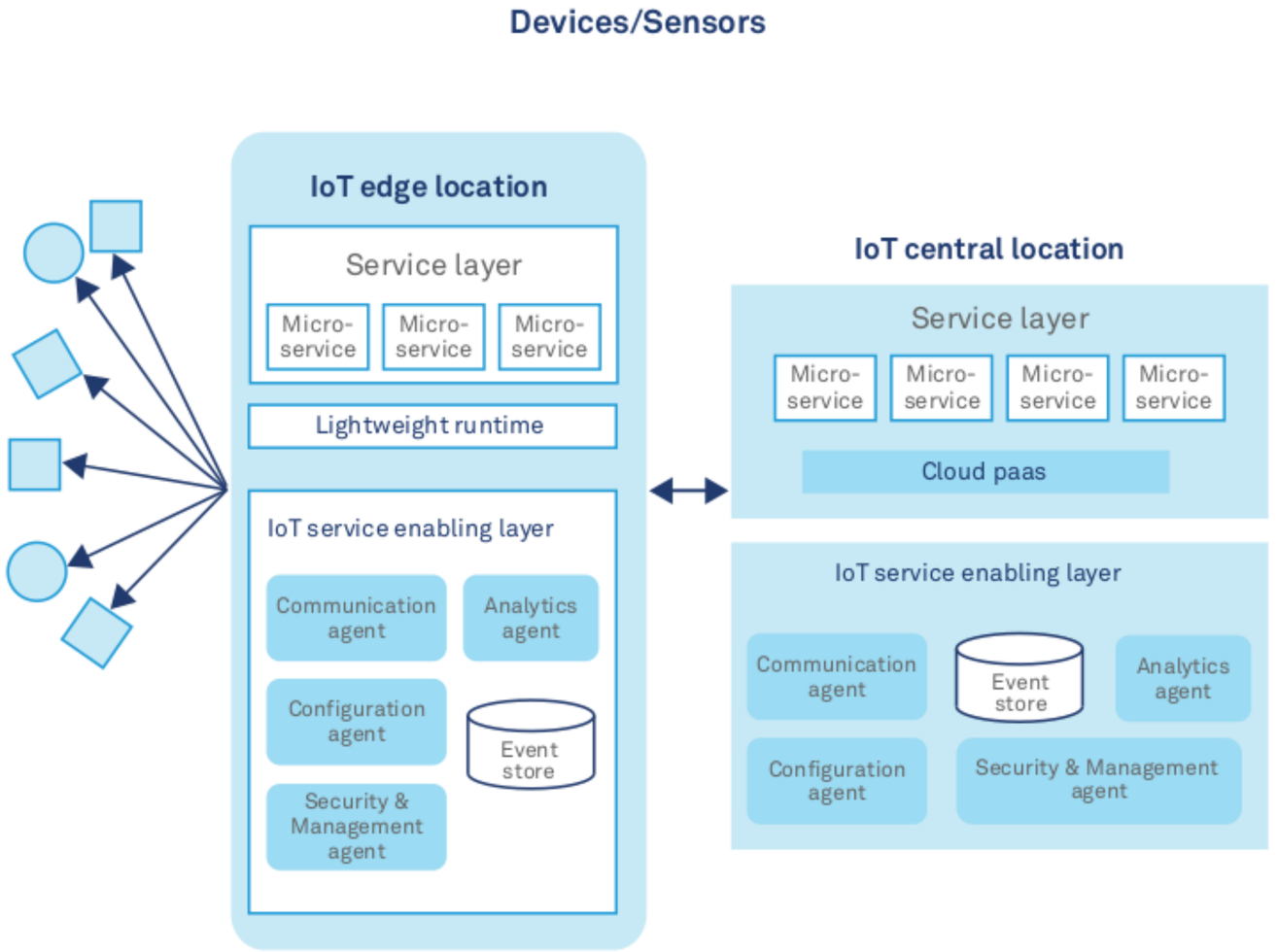
- **VMs, as we know, take up a lot more storage space than containers**
- A simple container is typically as small as 10MB
- A VM occupies at least a few GBs of storage space considering Operating System and Storage
- The same physical server can hold many more containers than VMs
- Of course if we are modeling a complex application with server microservices (where each functionality is a separate microservice) the isolation will require a large number of VMs or containers

Microservices - Containers - Faster Execution

- VMs are created by a hypervisor that needs a lot of configurational decisions at the start—the guest OS for running the application, amount of storage space needed, network preferences, and many such settings.
- On the other hand, containers are created faster than VMs due to the absence of a hypervisor. Container images are stored in a repository from where they can be pulled as required through a few quick commands
- Thus, the startup time for containers (e.g., Docker) ranges from a few milliseconds to a couple of seconds which makes it much swifter than a VM

IoT, Edge Computing & Microservices

- Due to large-scale IoT systems and connected devices, enterprises face difficulty in developing, deploying, integrating and scaling the applications
- Microservices and containerization enable efficient and faster development by breaking down IoT functionalities into small, modular and independent units that work in isolation without affecting the overall performance of the IoT ecosystem



IoT, Edge Computing & Microservices

- Services at the **edge** and in the central location (**cloud**) should **work together** to deliver complete business functionality.
- Services requiring **large computing resources** – big data, analytics, machine learning etc. – should be **delegated to the cloud**, not only because of the computing load but also to leverage cloud services.
- Services requiring scalability should be delegated to the cloud where possible
- Services with **high (local) criticality and quick response time** are better deployed **at the edge** because critical services should not depend on connectivity to the central location being available at all times
- Following **microservices architecture will make it easier to migrate services from the edge to the cloud, and vice versa**.
- As more computing power becomes available at the edge, it may make sense to move some services to the edge. Conversely, improved connectivity may allow some services to migrate to the cloud to take advantage of cloud features.

IoT, Edge Computing & Microservices

- **Interoperability** is one of the main fundamental feature talking about microservice at the Edge of the network
- **Microservices** architecture fully **enables heterogeneous technologies to co-exist**
- Services **communicate via standard interfaces** only and assume no knowledge of inner working of other services, therefore **they can work with one another even if they are implemented using different technology stacks**
- IoT solutions are evolving fast; devices, protocols, data formats and other technical details are still evolving. Therefore, I**oT services that communicate via standard interfaces will accelerate interoperability and help gain customer**

IoT, Edge Computing & Microservices

- Microservices architecture allows **faster releases** because of small service size and independence of deployment, supported by appropriate release philosophy, methods and tooling (Agile/DevOps, CI/DC etc.)
- This is another aspect that should appeal to IoT solution providers: **quick releases can be done to adapt to application behavior and augment/change functionalities of demand**
- An IoT system will have many physical parts deployed in possibly tough environment; any single component failing should not bring down a whole system
- **Decomposition of responsibility** into small size, and independent deployment - characteristics of microservices architecture - can create overall system **resilience**.

IoT, Edge Computing & Microservices

- Microservices architecture uses fine grained, single-responsibility services to build more complex business functionality
- New system capability can be built combining existing services in new ways (**Composability & Re-use**) - REST based interfaces commonly used in microservices architecture facilitates this. The IoT ecosystem can create faster value if basic services can be leveraged by multiple modules and actors to deliver higher value services.
- The **integration challenge** of IoT is varied and complex. Applying microservices architecture – encapsulating data and logic in a “black-box” (i.e. service) which is only accessible through a stateless interface - can simplify the integration challenge.
- A state-less interface is easier to integrate with because a service consumer will not need to track state transitions while communicating with the service provider.

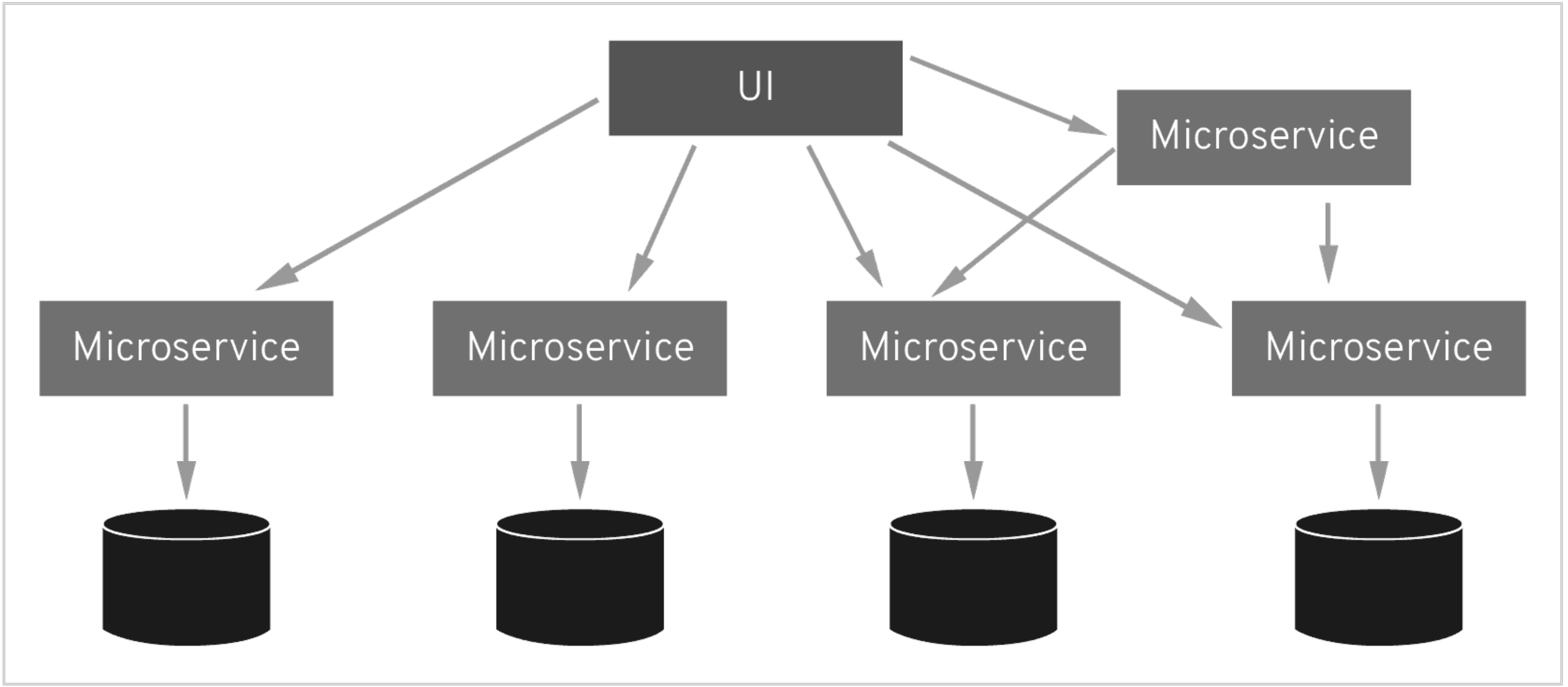
IoT, Edge Computing & Microservices

- Microservices architecture has become closely associated with certain architecture component/patterns such as
  - SQL and NoSQL databases
  - Event bus and Asynchronous communication
  - API management platform
  - Event and Data Correlation
  - Service choreography and Orchestration
- All these aspects are crucial also for IoT deployments both on the Edge and in Cloud.
- A large ecosystem of tools/methods have come up to support the development, deployment and operation of microservices – CI/CD tools, container management tools, and log aggregation and monitoring etc. These can be easily leveraged to build and operate IoT services.

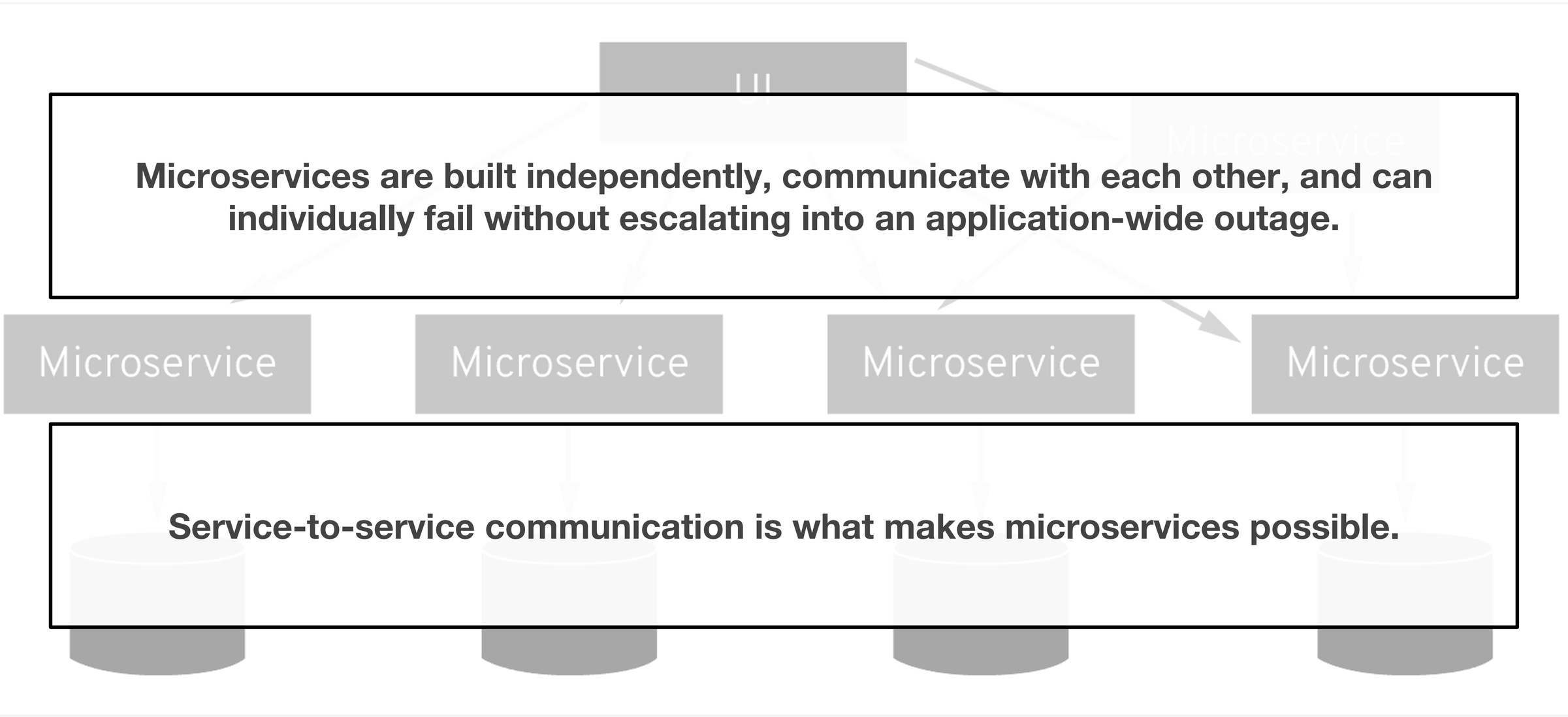
IoT, Edge, Microservices - Connected Future

- Microservices architecture encapsulates best practices for designing innovative, scalable, and resilient applications. It focuses on tangible results while remaining responsive to changing requirements. It is also an approach more suited to experimentation as it encourages a method of incremental learning-by-doing, and limits the impact of failure – at design time and at run time.
- Not all challenges of IoT and Edge Computing can be addressed with microservices architecture. Open challenges will be:
  - X,Y,Z, scalability at the edge
  - higher risk of failure of IoT devices due to exposure to physical environment
  - integration at the physical layer
  - Etc ….
- But it is absolutely true that applying microservices architecture to IoT and Edge Computing will build momentum towards a **service-based future for IoT and Edge Computing**

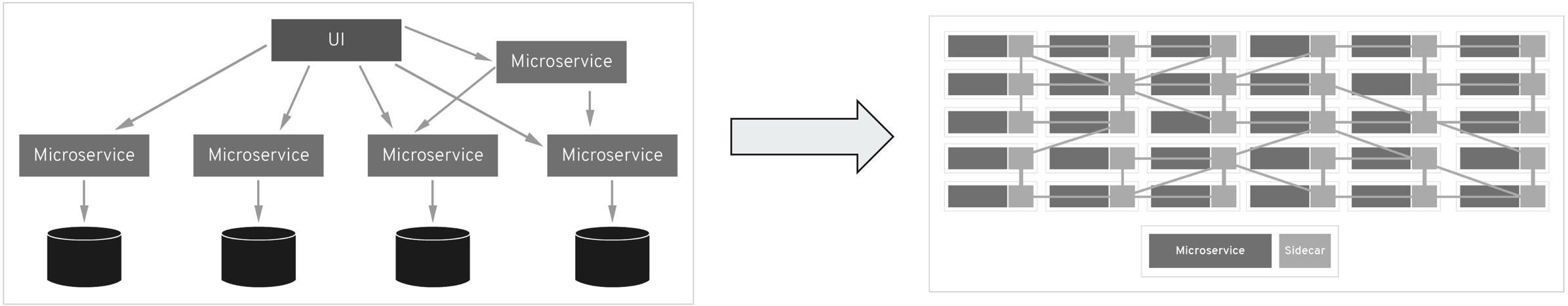
From Microservices to Service Mesh



From Microservices to Service Mesh



From Microservices to Service Mesh

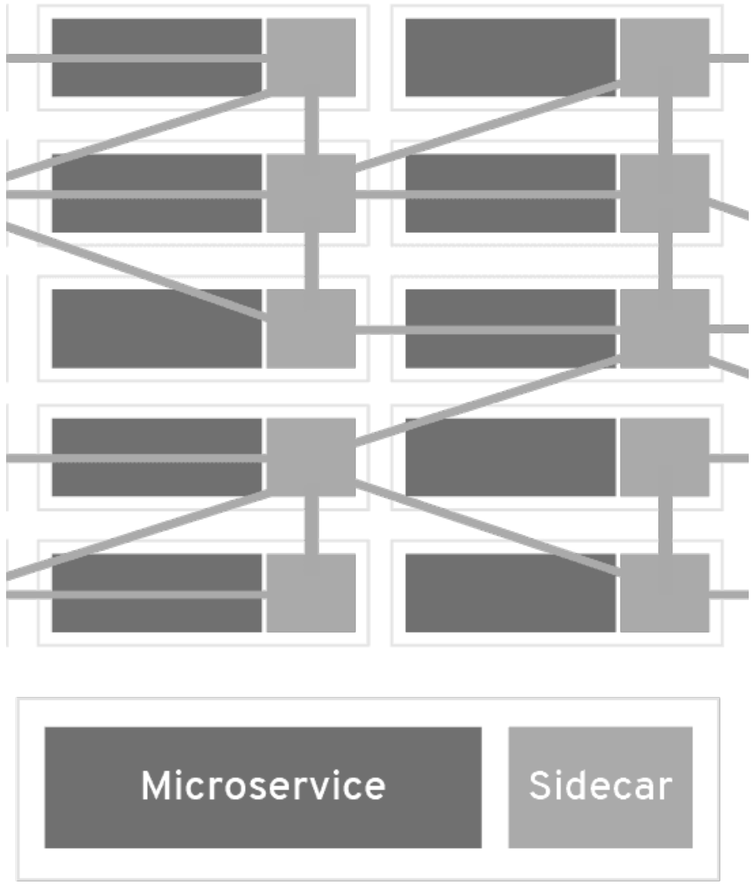


The logic governing communication can be coded into each service without a service mesh layer—but as communication gets more complex, a Service Mesh becomes more valuable.

For apps built in a microservices architecture, a Service mesh is a way to comprise a large number of discrete services into a functional application.

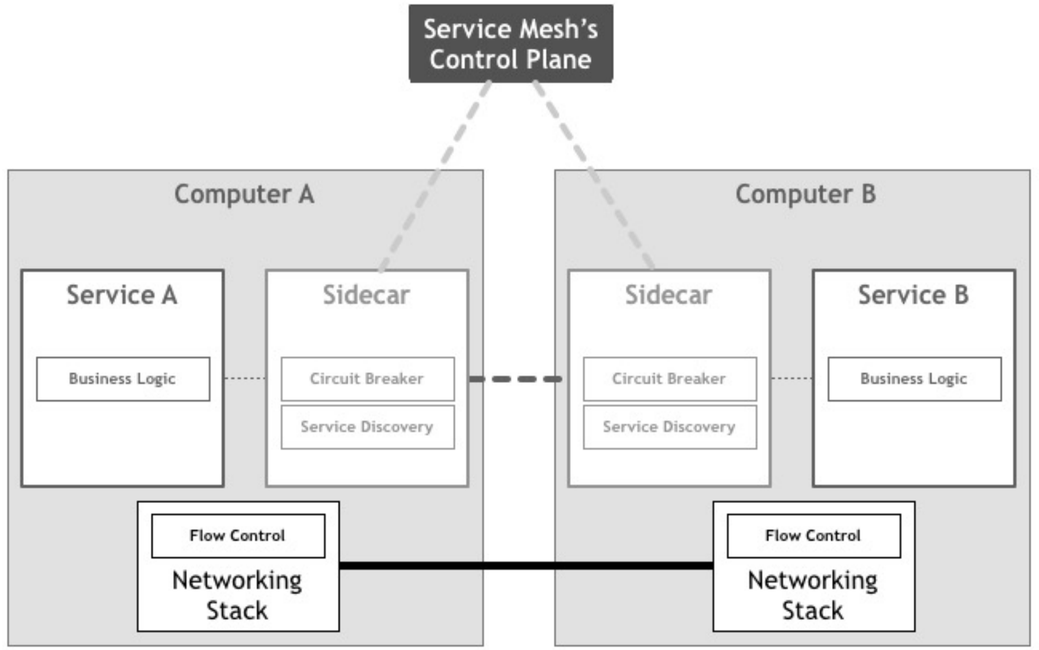
Service Mesh

- A service mesh is a configurable, low-latency infrastructure layer designed to handle a high volume of network-based interprocess communication among application infrastructure services using application programming interfaces (APIs)
- A service mesh ensures that communication among containerized and often ephemeral application infrastructure services is fast, reliable, and secure.
- The mesh provides critical capabilities including service discovery, load balancing, encryption, observability, traceability, authentication and authorization, and support for the circuit breaker pattern.



Service Mesh

- The service mesh is usually implemented by providing a proxy instance, called a **sidecar** for each service instance
- Sidecars handle
  - interservice communications
  - Monitoring
  - Security
  - and in general anything that can be abstracted away from the individual services
- This way ...
  - developers can focus and handle development, support, and maintenance for the application code in the services
  - operations teams can maintain the service mesh and run the app.



Service Mesh - Container Orchestration

Service mesh comes with its own terminology for component services and functions:

- **Container orchestration framework**
  - As more and more containers are added to an application's infrastructure, a separate tool for monitoring and managing the set of containers (denoted as container orchestration framework) becomes essential
  - **Kubernetes** seems to have cornered this market, with even its main competitors, Docker Swarm and Mesosphere DC/OS, offering integration with Kubernetes as an alternative.

Service Mesh - Service & Instances

- **Services and instances (Kubernetes pods)**
  - An instance is a single running copy of a microservice
  - Sometimes the instance is a single container
  - In Kubernetes, an instance is made up of a small group of interdependent containers (called a pod)
  - Clients rarely access an instance or pod directly. They usually access a service, which is a set of identical instances or pods (replicas) that is scalable and fault-tolerant.

Service Mesh - Sidecar Proxy

- **Sidecar proxy**
  - A sidecar proxy runs alongside a single instance or pod
  - The purpose of the sidecar proxy is to route, or proxy, traffic to and from the container it runs alongside
  - The sidecar communicates with other sidecar proxies and is managed by the orchestration framework
  - Many service mesh implementations use a sidecar proxy to intercept and manage all incoming and outgoing traffic to the instance or pod.

Service Mesh - Service Discovery

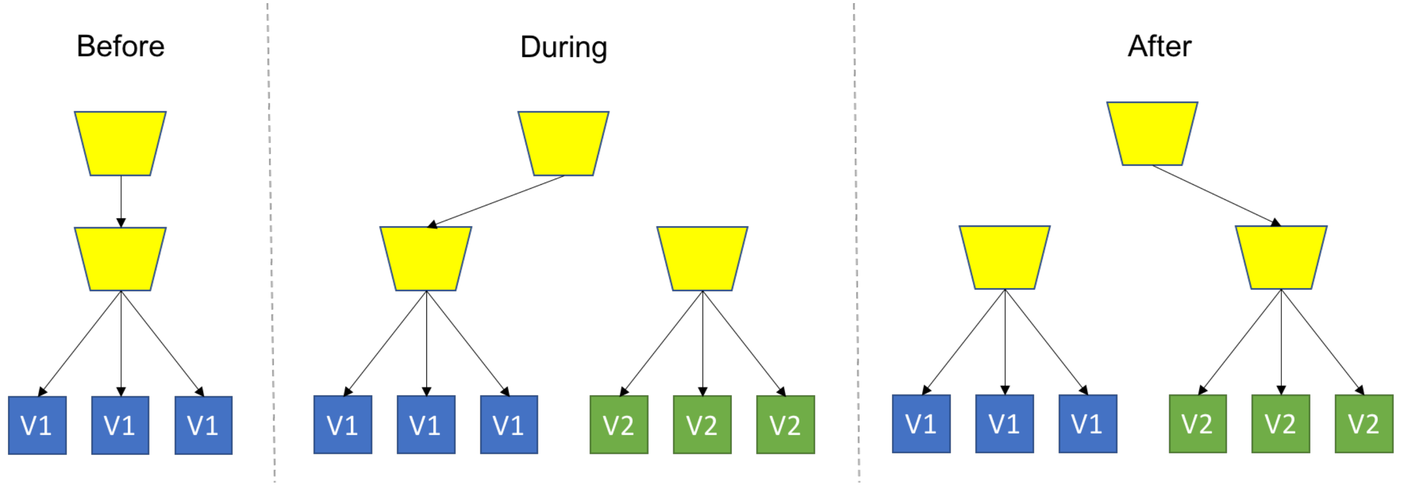
- **Microservice Service Discovery**
  - When an instance needs to interact with a different service, it needs to find/discover a healthy, available instance of the other service
  - Typically, the instance performs a DNS lookup for this purpose
  - The container orchestration framework keeps a list of instances that are ready to receive requests and provides the interface for DNS queries

Service Mesh - Load Balancing

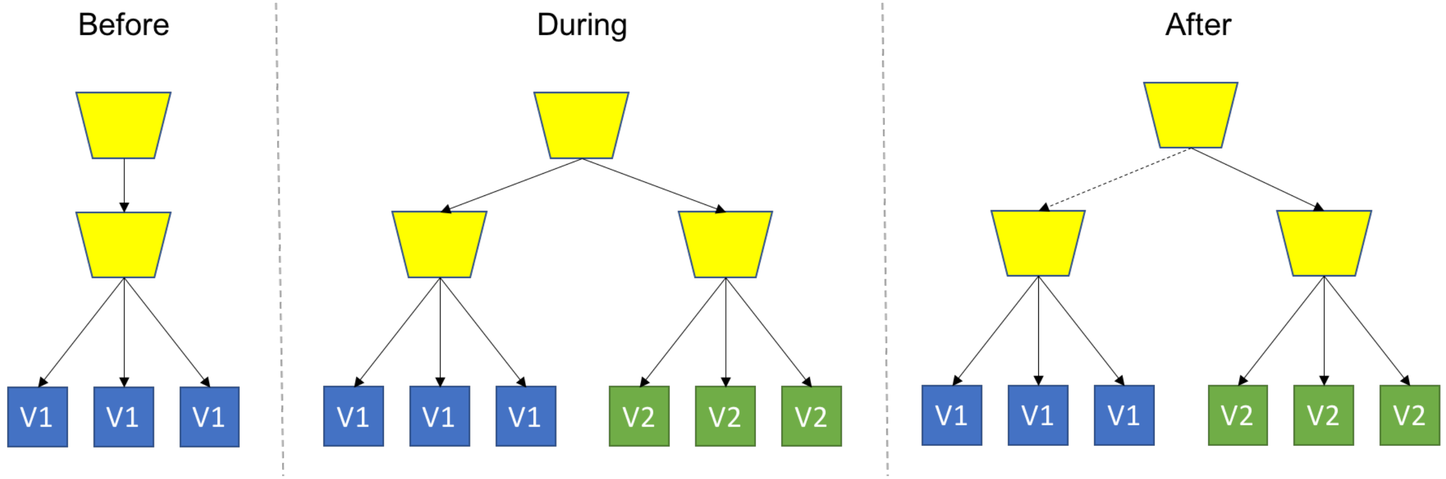
- **Load Balancing**
  - Most orchestration frameworks already provide Layer 4 (transport layer) load balancing
  - A service mesh implements more sophisticated Layer 7 (application layer) load balancing, with richer algorithms and more powerful traffic management
  - Load-balancing parameters can be modified via API, making it possible to orchestrate blue-green or canary deployments.

Service Mesh - Load Balancing

Blue/Green



Canary



Service Mesh - Encryption

- **Encryption**
  - The service mesh can encrypt and decrypt requests and responses, removing that responsability from each of the services
  - The service mesh can also improve performance by prioritizing the reuse of existing, persistent connections, which reduces the need for the computationally expensive creation of new ones
  - The most common implementation for encrypting traffic is mutual TLS (mTLS), where a Public Key Infrastructure (PKI) generates and distributes certificates and keys for use by the sidecar proxies

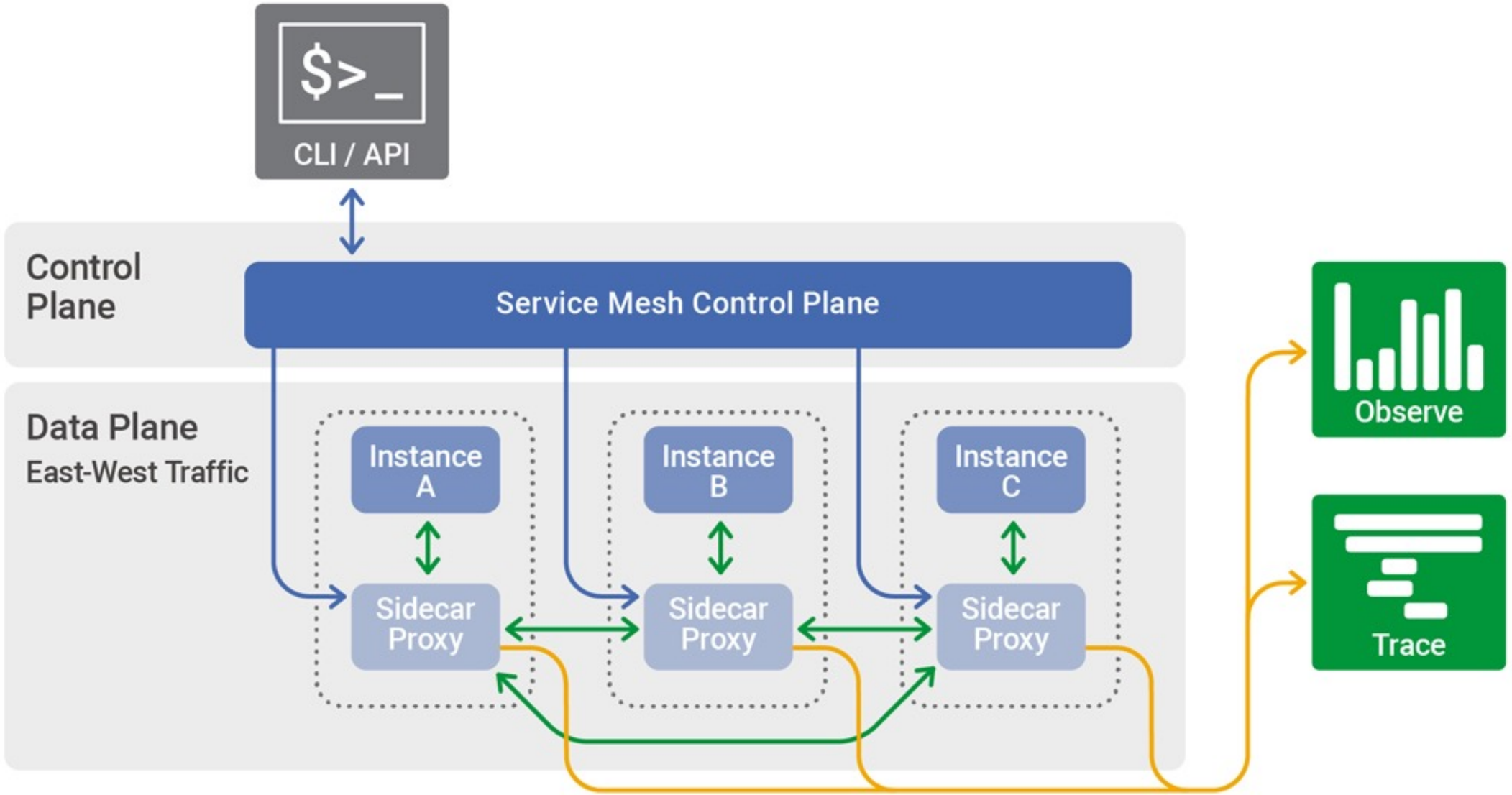
Service Mesh - AA & Circuit Breaker Pattern

- **Authentication and authorization**
  - The service mesh can authorize and authenticate requests made from both outside and within the app, sending only validated requests to instances
- **Circuit breaker pattern**
  - The service mesh can support the circuit breaker pattern, which isolates unhealthy instances, then gradually brings them back into the healthy instance pool if warranted.

Service Mesh - Data Plane & Control Plane

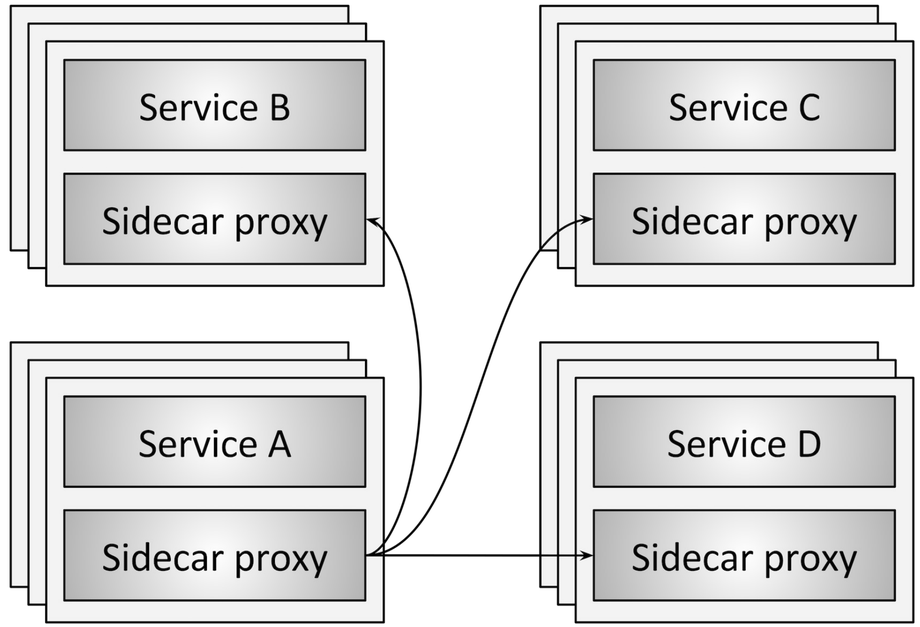
- The **Data Plane (DP)** represents the part of a service mesh application that manages the network traffic between instances
- Generating and deploying the configuration that controls the data plane's behavior is done using a separate control plane
- The **Control Plane (CP)** in a service mesh distributes configuration across the sidecar proxies in the data plane
- The CP typically includes, or is designed to connect to, an API, a command-line interface, and a graphical user interface for managing the app.

Service Mesh



Service Mesh - Data Plane

- In a service mesh, the sidecar proxy performs the following tasks
  - Service Discovery
  - Health Checking
  - Routing
  - Load Balancing
  - Authentication and Authorization
  - Observability
- We can say that … the sidecar proxy is the data plane

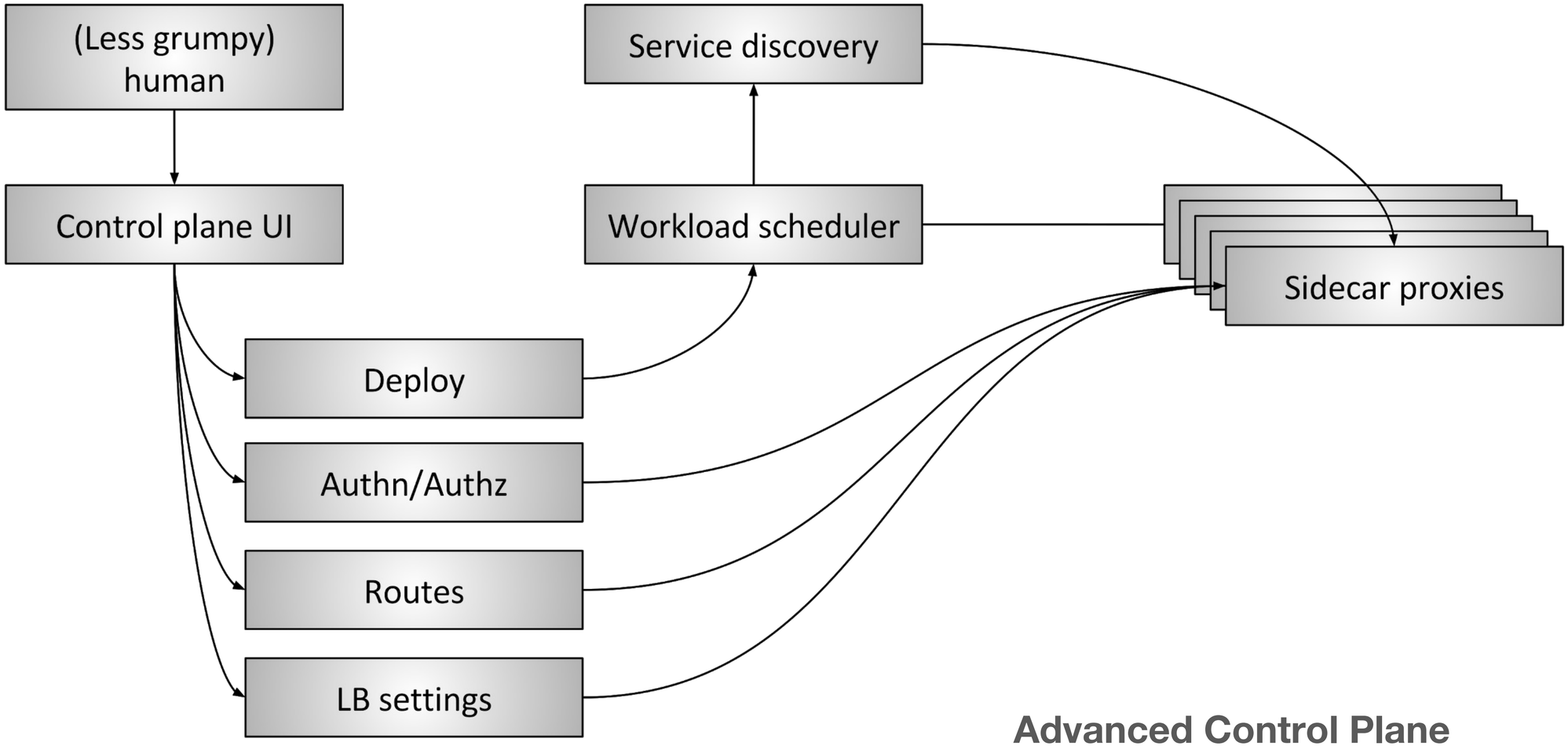


Service Mesh - Control Plane

- Provides policy and configuration for all of the running data planes in the mesh
- Does not touch any packets/requests in the system.
- The control plane turns all of the data planes into a distributed system.

Human Control Plane

Service Mesh - Control Plane



Advanced Control Plane

Service Mesh - Current Projects

- Istio, backed by Google, IBM, and Lyft, is currently the best-known service mesh architecture
- Kubernetes, which was originally designed by Google, is currently the only container orchestration framework supported by Istio.
- Vendors (e.g., RedHat) are seeking to build commercial, supported versions of Service Mesh solution on top of community software
- Data planes: Linkerd, NGINX, HAProxy, Envoy, Traefik
- Control planes: Istio, Nelson, SmartStack

https://blog.openshift.com/red-hat-openshift-service-mesh-is-now-available-what-you-should-know/

References

1. Monolithic vs Microservices - https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59
2. Pattern: Microservice Architecture - https://microservices.io/patterns/microservices.html
3. Monolithic Architecture - https://microservices.io/patterns/monolithic.html
4. Don’t start with Monolith - https://martinfowler.com/articles/dont-start-monolith.html
5. API Gateway Pattern - https://www.linkedin.com/pulse/api-gateway-pattern-ronen-hamias/
6. Monolithic vs Microservice and all in between - https://medium.com/swlh/monolithic-vs-micro-services-and-all-in-between-7d496408ad02
7. Best Architecture for an MVP: Monolith, SOA, Microservices, or Serverless? - https://rubygarage.org/blog/monolith-soa-microservices-serverless
8. Microservices Introduction (Monolithic vs. Microservice Architecture) - https://dzone.com/articles/microservices-1-introduction-monolithic-vs-microse
9. How to break a Monolith into Microservices - https://martinfowler.com/articles/break-monolith-into-microservices.html

References

10. Accelerate IoT Applications With Microservices: A Use Case - https://medium.com/the-why-and-how/accelerate-iot-applications-with-microservices-a-use-case-1dd925aad0ae
11. Why IoT Development Needs Microservices and Containerization - https://www.einfochips.com/blog/why-iot-development-needs-microservices-and-containerization/
12. Future of Microservices & IoT - https://hackernoon.com/future-of-microservices-iot-2efc0ca84eb6
13. Bringing the power of Microservices to IoT - https://www.wipro.com/applications/bringing-the-power-of-microservices-to-iot/
14. Microservice Architectures: What They Are and Why You Should Use Them - https://blog.newrelic.com/technology/microservices-what-they-are-why-to-use-them/
15. The Scale Cube - https://microservices.io/articles/scalecube.html
16. The Art of Scalability - http://theartofscalability.com/
17. Architected Cubed - https://www.benefitfocus.com/blogs/design-engineering/architecture-cubed
18. Microservice: Architecture - https://medium.com/@cinish/microservices-architecture-5da90504f92a
19. Orchestrating Microservices: A Guide for Architects - https://dzone.com/articles/orchestrating-microservices-a-guide-for-architects

References

20. Microservice Orchestration - https://medium.com/jexia/microservice-orchestration-9ee71160882f
21. Service Mesh for Microservices - https://medium.com/microservices-in-practice/service-mesh-for-microservices-2953109a3c9a
22. What is a Service Mesh ? - https://www.nginx.com/blog/what-is-a-service-mesh/
23. API Gateway / Backends for Frontends - https://microservices.io/patterns/apigateway.html
24. Embracing the Differences : Inside the Netflix API Redesign - https://medium.com/netflix-techblog/embracing-the-differences-inside-the-netflix-api-redesign-15fd8b3dc49d
25. Optimizing the Netflix API - https://medium.com/netflix-techblog/optimizing-the-netflix-api-5c9ac715cf19
26. Microservices vs. Monolith Architecture - https://dev.to/alex_barashkov/microservices-vs-monolith-architecture-4l1m
27. Monolith First - https://martinfowler.com/bliki/MonolithFirst.html
28. Ecliode - ioFog - https://iofog.org/
29. VM vs Containers for Microservice: https://hackernoon.com/vms-vs-containers-for-microservices-a6f559970704

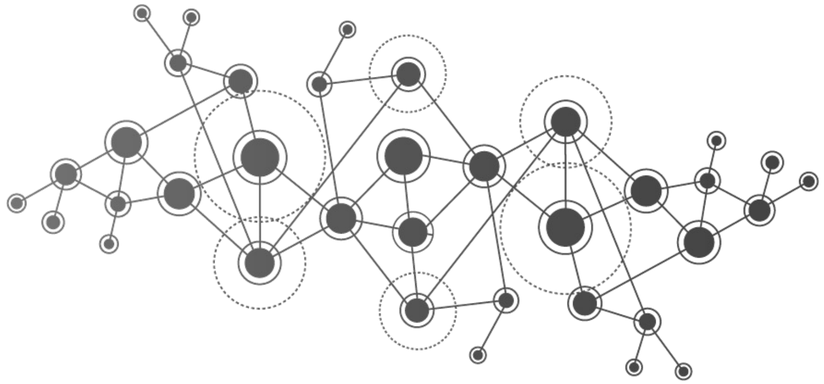
References

30. Distributed Application Architecture for Edge-Based Service Delivery - https://thenewstack.io/distributed-application-architecture-for-edge-based-service-delivery/
31. Next generation of edge computing: AI, microservices and container orchestration - https://medium.com/@antowan/next-generation-of-edge-computing-ai-microservices-and-container-orchestration-1c22fc2be71b
32. IoT edge development and deployment with containers through OpenShift - https://developers.redhat.com/blog/2019/01/31/iot-edge-development-and-deployment-with-containers-through-openshift-part-1/
33. What's a Service Mesh ? - https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh
34. Service Mesh for developer workflow, a series - https://medium.com/solo-io/service-mesh-for-the-developer-workflow-a-series-ef279c49ab56
35. Red Hat OpenShift Service Mesh is now available: What you should know - https://blog.openshift.com/red-hat-openshift-service-mesh-is-now-available-what-you-should-know/
36. What is Kubernetes - https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

References

30. Distributed Application Architecture for Edge-Based Service Delivery - https://thenewstack.io/distributed-application-architecture-for-edge-based-service-delivery/
31. Next generation of edge computing: AI, microservices and container orchestration - https://medium.com/@antowan/next-generation-of-edge-computing-ai-microservices-and-container-orchestration-1c22fc2be71b
32. IoT edge development and deployment with containers through OpenShift - https://developers.redhat.com/blog/2019/01/31/iot-edge-development-and-deployment-with-containers-through-openshift-part-1/
33. What's a Service Mesh ? - https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh
34. Service Mesh for developer workflow, a series - https://medium.com/solo-io/service-mesh-for-the-developer-workflow-a-series-ef279c49ab56
35. Red Hat OpenShift Service Mesh is now available: What you should know - https://blog.openshift.com/red-hat-openshift-service-mesh-is-now-available-what-you-should-know/
36. What is Kubernetes - https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
37. Service mesh data plane vs. control plane - https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA



Intelligent Internet of Things

Monolithic vs Microservice oriented Architectures for the IoT

Prof. Marco Picone

A.A 2023/2024