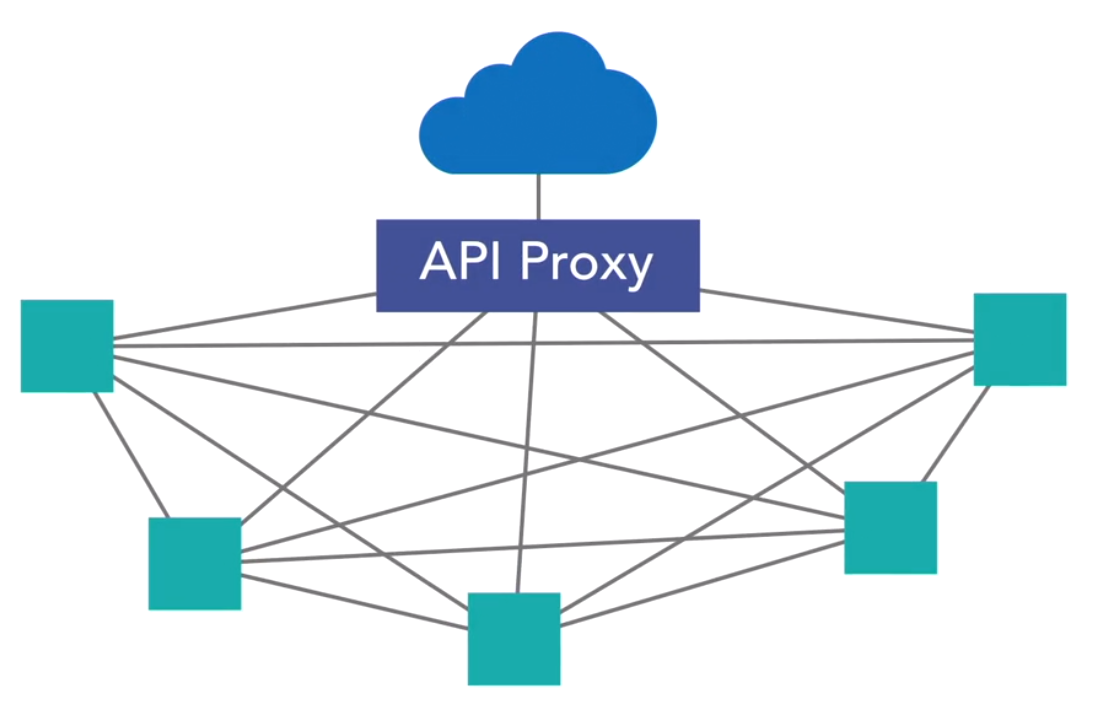
HighLevel Project Architecture

Microservice Architecture



**Microservices Core Concepts:**

In software engineering, a microservice architecture is a variant of the service-oriented architecture structural style. It is an architectural pattern that arranges an application as a collection of loosely coupled, fine-grained services, communicating through lightweight protocols

**9 Key attributes of Microservices as outlines in Martin Fowler's seminal article:**

## **Componentization via services**

* + Microservices are structured as independent components or services, each responsible for a specific business function. This modular approach enables flexibility, scalability, and ease of maintenance, as each service can be developed, deployed, and scaled independently.
  + Try as much as possible to make your component as “services”.
  + The modularity of your system should be done using out of process components, also known as services, rather than in process components implemented as libraries.
  + The ease of deployment & maintenance of a system where its components are out of process services is very valuable.

## **Organized around business capabilities**

* + Microservices are aligned with business capabilities or domains, ensuring that each service corresponds to a specific business function. This organizational principle promotes agility and focuses development efforts on delivering value to the business.
  + This forces the team, to think long & hard about the boundaries of this service.
  + We cannot say, let's start with small API, and see where it goes. We can't change the API at will. Since other services will become incompatible.
  + With well boundaries, we make system more modular, the services more independent of other services, and ease of maintenance

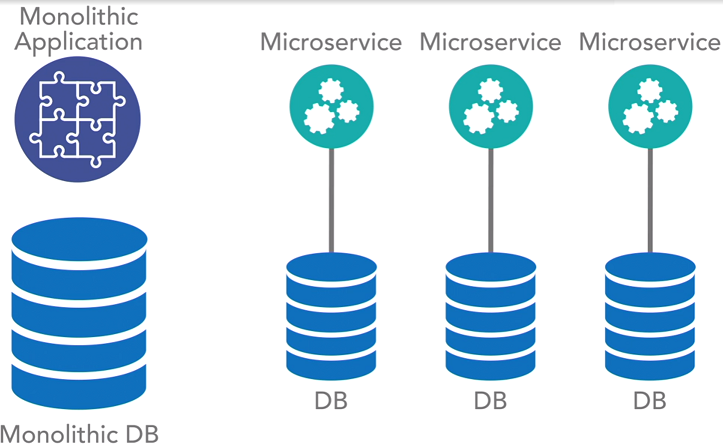
## Organize services based on the domain. Not on size.

* It isn't much about the size of the service. But it's about the operation of the services which really matters. Domain operation focused
* A microservice handles one set of related functions with little or no cross-domain operations (Domain-driven design).
* In a microservices architecture, a service operates on a well-defined domain. Operations aren’t defined on data objects or business objects. Instead, its the domain as a whole.

## **Decentralized governance**

* + Decision-making authority and control are distributed across autonomous teams, empowering them to make decisions independently within the boundaries of their respective services. This decentralized approach fosters agility, innovation, and accountability, enabling teams to respond effectively to local requirements and business needs.
  + It lets the teams use the technology best- suited for the services needs.
  + No need of Service handling documents with slow & incompatible relational database
  + No more must we use slow & legacy technology, when what we really need is a light & fast one.
  + Team, can use the best tool for the task. And makes team satisfied, and as a results, the other services and the end users too.

## **Decentralized data management (When possible)**

* + Each microservice has its own database or data store, allowing teams to choose the most suitable solution for their service's requirements. This decentralized approach minimizes dependencies and supports autonomy, enabling teams to innovate and iterate independently.
  + Using a separate database for each service
  + It makes the service more autonomous & less dependent on other external mechanisms
  + 

## **Infrastructure automation**

* + Microservices rely on automation for deployment, scaling, monitoring, and recovery, enabling rapid and reliable delivery of software at scale. Continuous integration/continuous deployment (CI/CD) pipelines, containerization, and orchestration tools streamline operations and enhance efficiency.
  + Without this automation, would result in a system slow to test & deploy. Clients would get doubts about the effectiveness of it all.

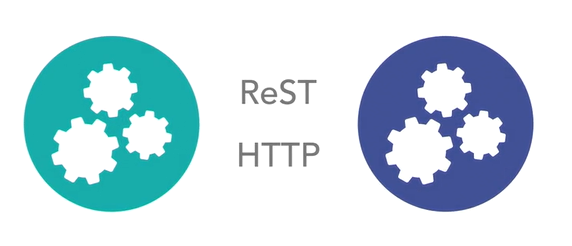
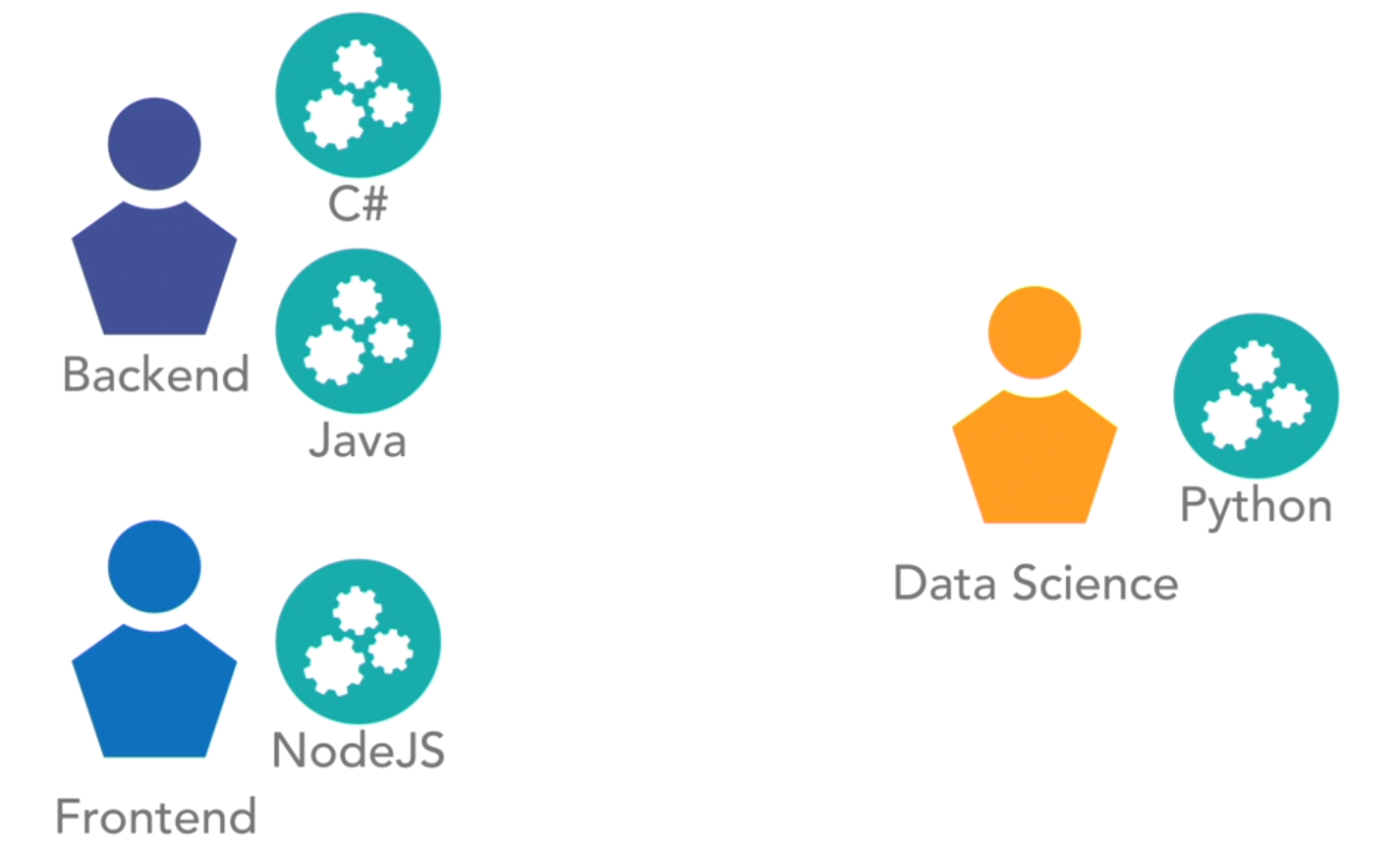
### Distribution & Scale:

* Communication patterns lead to a truly distributed model. Distribution beyond a single data center is not a requirement for microservices architecture, it's definitely one of the benefits of this model.
  + With distribution, you can deploy assets across the globe, & expand.
* Supports highly scalable systems.
  + Each application is independent of any other application.
  + As such when individual service comes under load, it can be individually scaled.
  + AWS infra diagram

**6. Smart Endpoints and Dumb Pipes:**

* + Microservices communicate via lightweight protocols, such as HTTP or messaging queues, with minimal reliance on complex middleware. This design philosophy emphasizes self-contained services that are easy to understand, test, and evolve.

### **How do the services communicate?**

* All communication between individual services in a microservices architecture is over HTTP using REST-based services(in most of scenarios).
* Allows for the use of any coding language or framework that supports RESTful services.
* 
* Teams can develop microservices in any technology, however, cross-communication should always be REST over HTTP. So they all communicate in a common way. Despite being developed in any manner. Teams work faster, as they are in the technology they are good at.
* In Pure Microservices, any service can talk to any other microservice.
  + Hierarchical Service architecture
    1. Here you define rules about which service types can or cannot consume other service types.
* **Protocol-Aware Heterogeneous Interoperability.** 
  + Services are bound to a protocol(HTTP) & execute communication over that protocol in a way that works in a mixed, or heterogeneous, environment.
  + 
* Issues:
  + Problems can arise from all of these network calls, as each service can call any other service.
  + Any service can call any other service. In purest form. There are no constraints on who can call what.

**7. Products, Not Projects:**

* + Microservices are treated as long-term products with ongoing ownership and responsibility. This approach encourages teams to prioritize quality, reliability, and maintainability, fostering a culture of continuous improvement and customer-centricity.

**8. Design for Failure:**

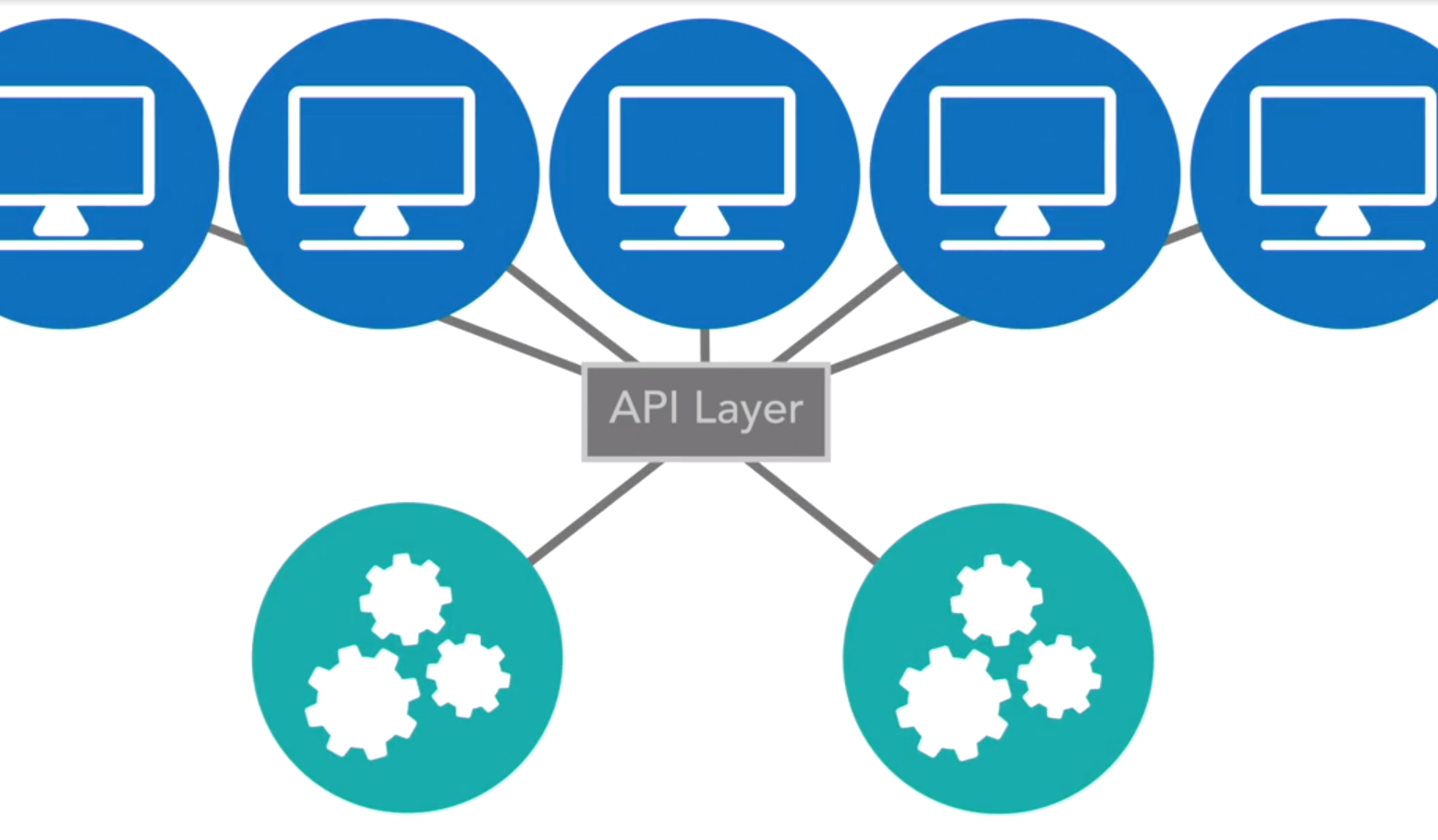
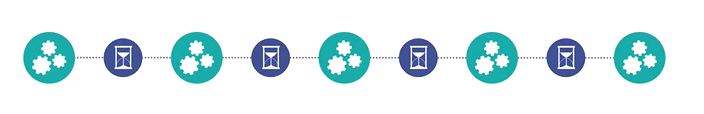
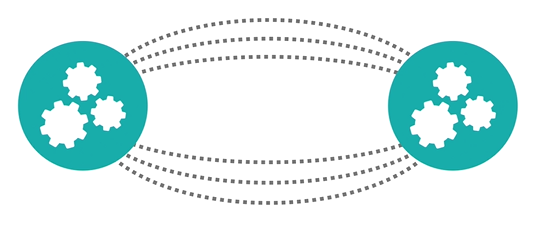
* + Microservices are designed to handle failures gracefully, with mechanisms such as redundancy, circuit breakers, and graceful degradation. This resilience-focused approach ensures that the system remains responsive and available, even in the face of unexpected failures or disruptions.

**9. Evolutionary Design:**

* + Microservices prioritize evolutionary design and incremental improvement over upfront planning and monolithic architectures. Services evolve iteratively based on feedback, changing requirements, and emerging business needs, enabling organizations to adapt quickly to market dynamics.

============================

**Another practice - The API layer**

* The API layer is a common component of a microservices architecture.
  + 
  + API layer should not transform or execute logic.
  + In a pure microservices architecture, an API layer is nothing more than an aggregated proxy of all of your service offerings.
  + The API layer is used, to shield the outside world or your clients from knowing the structure, organization, or even what exact service is exposing a specific operation.
  + The API layer provides a standardized proxy interface that will expose whatever service endpoints & API operations we configure it to expose.
  + The API layer is a pure proxy, with no transformation.
* **Concerns**
  + Every service invocation in a microservices architecture is over a remote network call. As such, there is connection setup, tear down, and wire latency on every single call. Since is less for 1 call, but as many hops come in place, this adds much more time. GridLock.
    1. 
    2. When traffic increases & services come under more load, the risk of latency in response time increases.
    3. At a certain point, this latency can lead to gridlock of the system as a whole. When calls are waiting for responses, delays can become unbearable, when this occurs, there could be a catastrophic failure.
  + Circular calls:
    1. 
  + The solution could be a circuit breaker. Cut down some of the features. .and continue a defined path.
  + Solution: Strong timeouts, Global distribution of all services offered, scaling of individual services under load & leveraging patterns like circuit breakers will help to solve problems when they do occur.