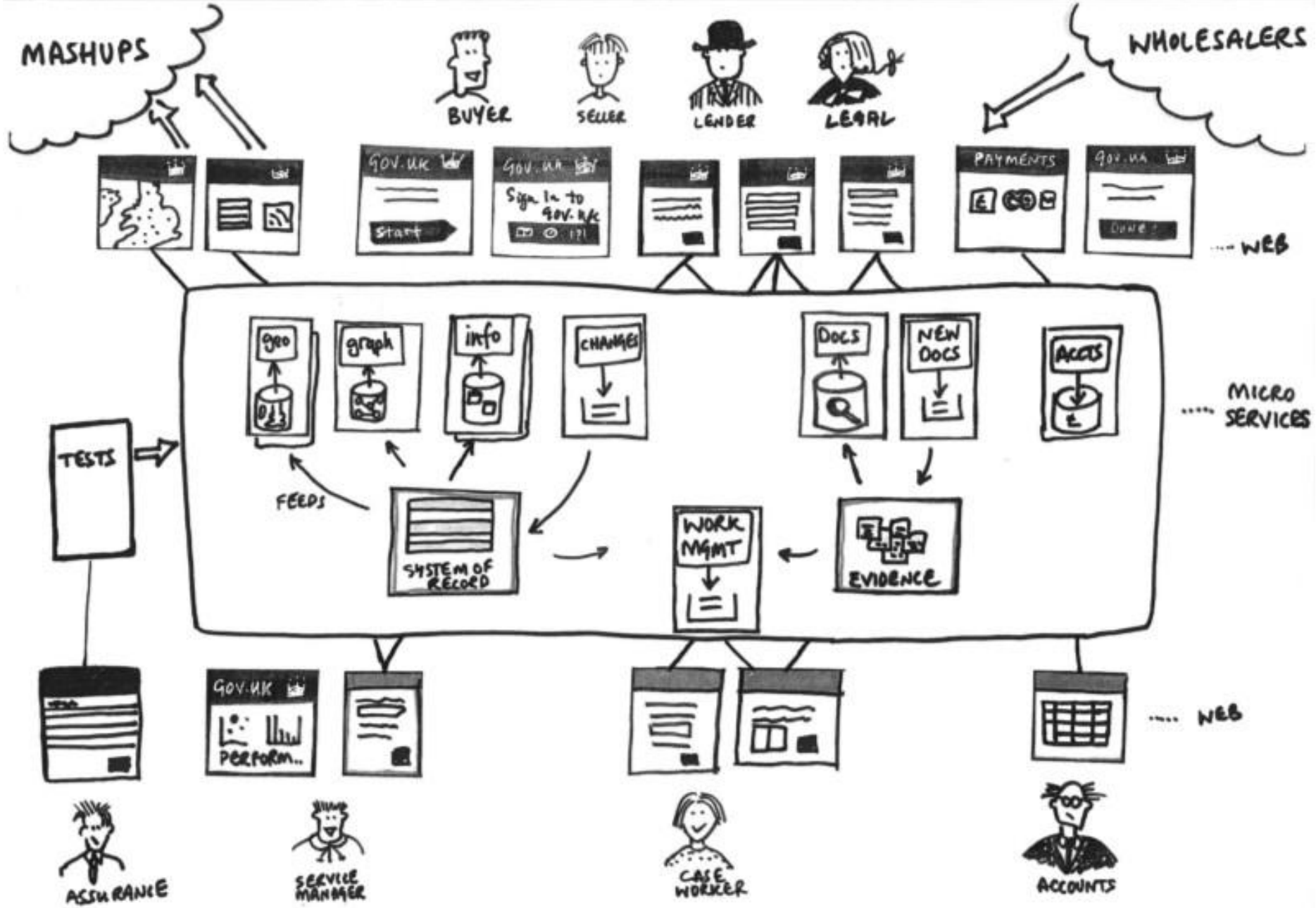
Introduction to Microservices


MASHUPS
BUYER
SELLER
LENDER
LEGAL
WHOLESALERS
GOV.UK
Start
GOV.UK
Sign in to GOV.UK
PAYMENTS
GOV.UK
Done
WEB
geo
graph
info
CHANGES
DOCS
NEW DOCS
ACCTS
MICRO SERVICES
TESTS
FEEDS
SYSTEM OF RECORD
WORK MGMT
EVIDENCE
GOV.UK
PERFORM..
WEB
ASSURANCE
SERVICE MANAGER
CASE WORKER
ACCOUNTS

Network architectures based on the micro service pattern have gained quite some popularity in recent years. Micro services have emerged as a solution to large, unwieldy monolithic applications.

This design attempts to solve the problems that emerge when your codebase grows beyond a certain size, and it becomes increasingly difficult to maintain.

Small scale services were borne out of the necessity to scale rapidly while still keeping your code manageable. Netflix, Amazon, and Spotify are some of the larger and more interesting players moving towards this type of pattern.

Why Microservices?

Let’s see the architecture that prevailed before microservices i.e. the **Monolithic Architecture.**

In layman terms, you can say that its similar to a big container wherein all the software components of an application are assembled together and tightly packaged.

Listed down are the challenges of Monolithic Architecture:

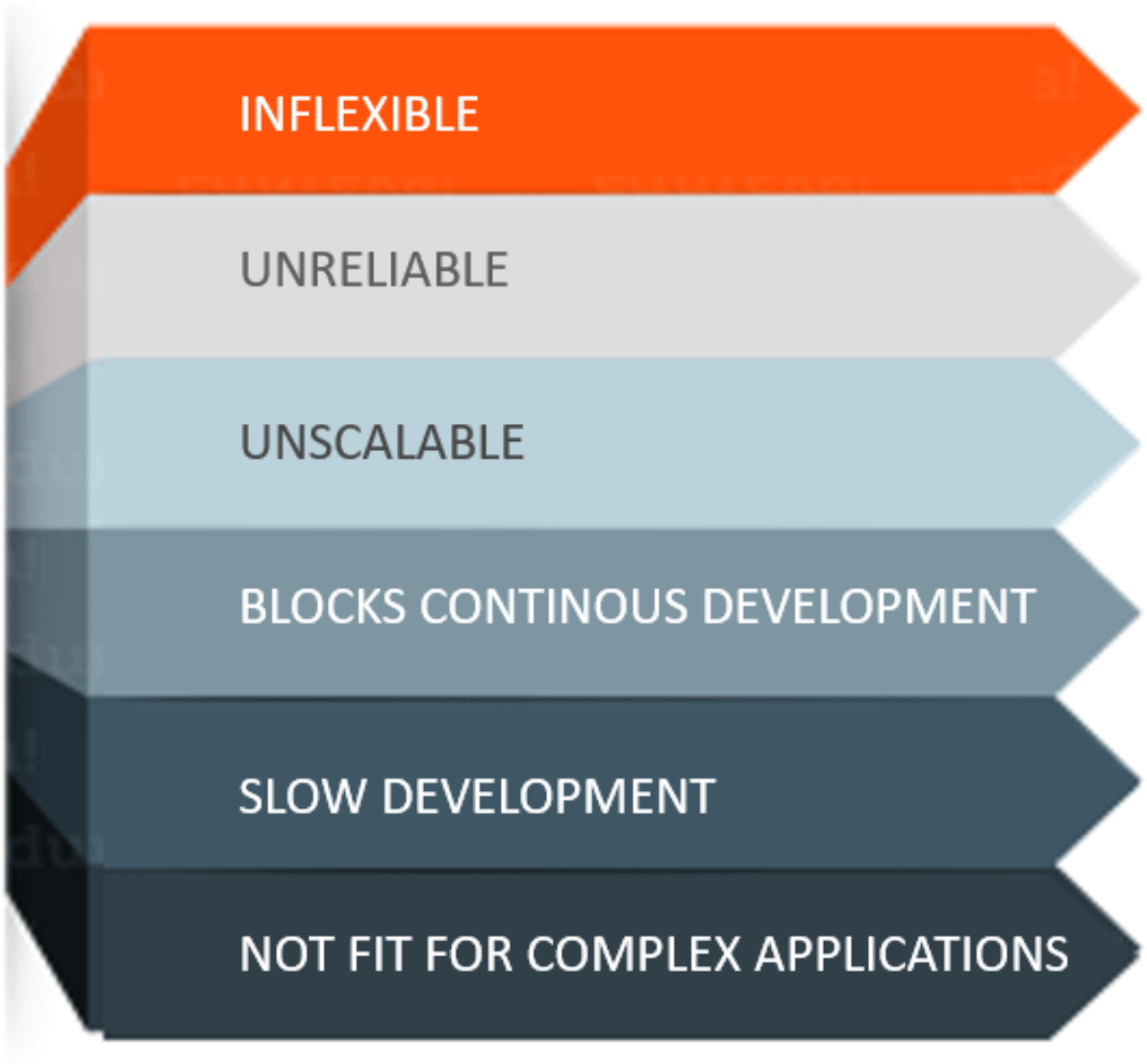
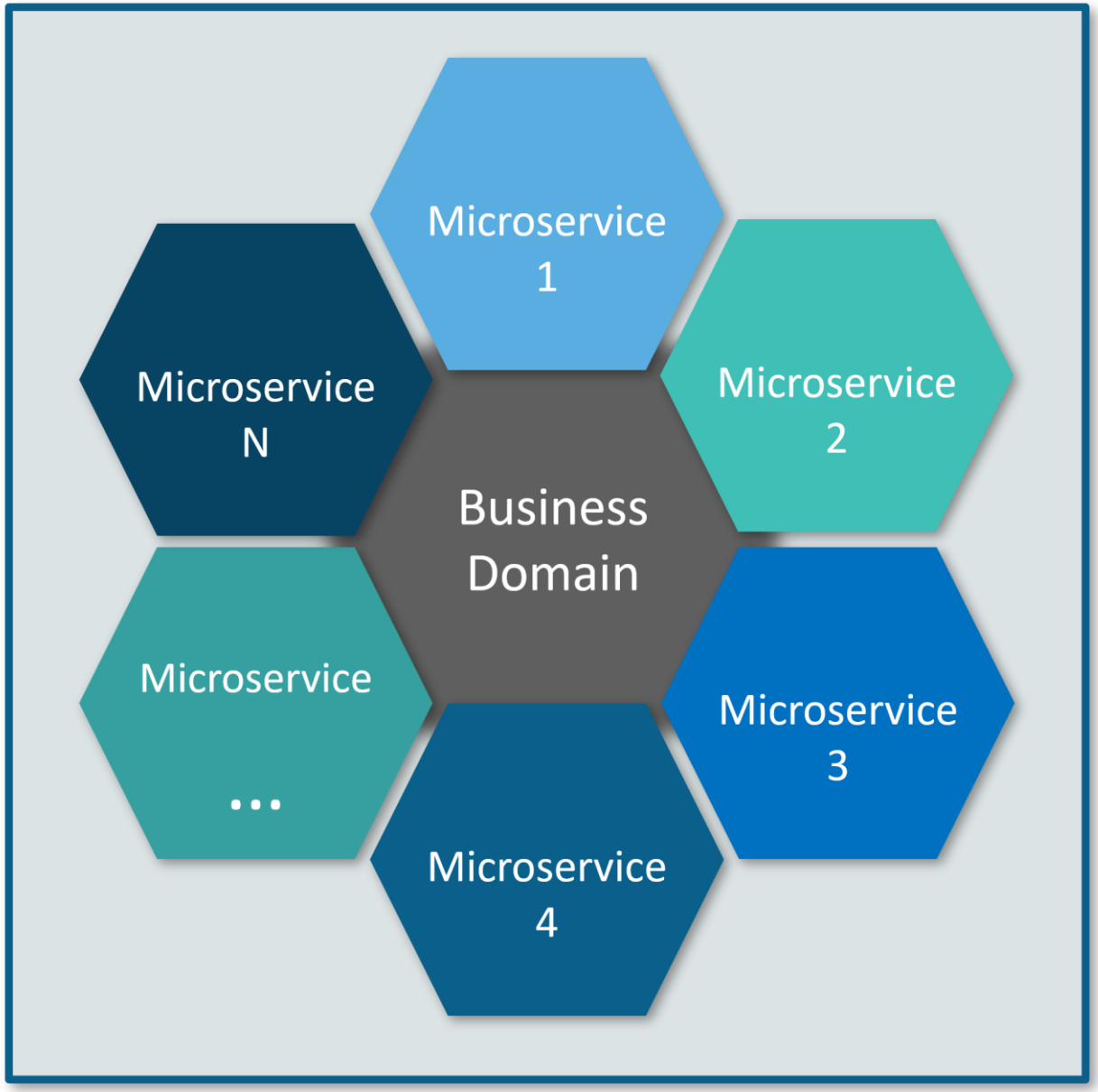


Figure 1: What Is Microservices – Challenges of Monolithic Architecture

- **Inflexible –** Monolithic applications cannot be built using different technologies
- **Unreliable –** Even if one feature of the system does not work, then the entire system does not work
- **Unscalable –** Applications cannot be scaled easily since each time the application needs to be updated, the complete system has to be rebuilt
- **Blocks Continous Development –** Many features of the applications cannot be built and deployed at the same time
- **Slow Development –** Development in monolithic applications take lot of time to be built since each and every feature has to be built one after the other
- **Not Fit For Complex Applications –** Features of complex applications have tightly coupled dependencies

What Is Microservices?

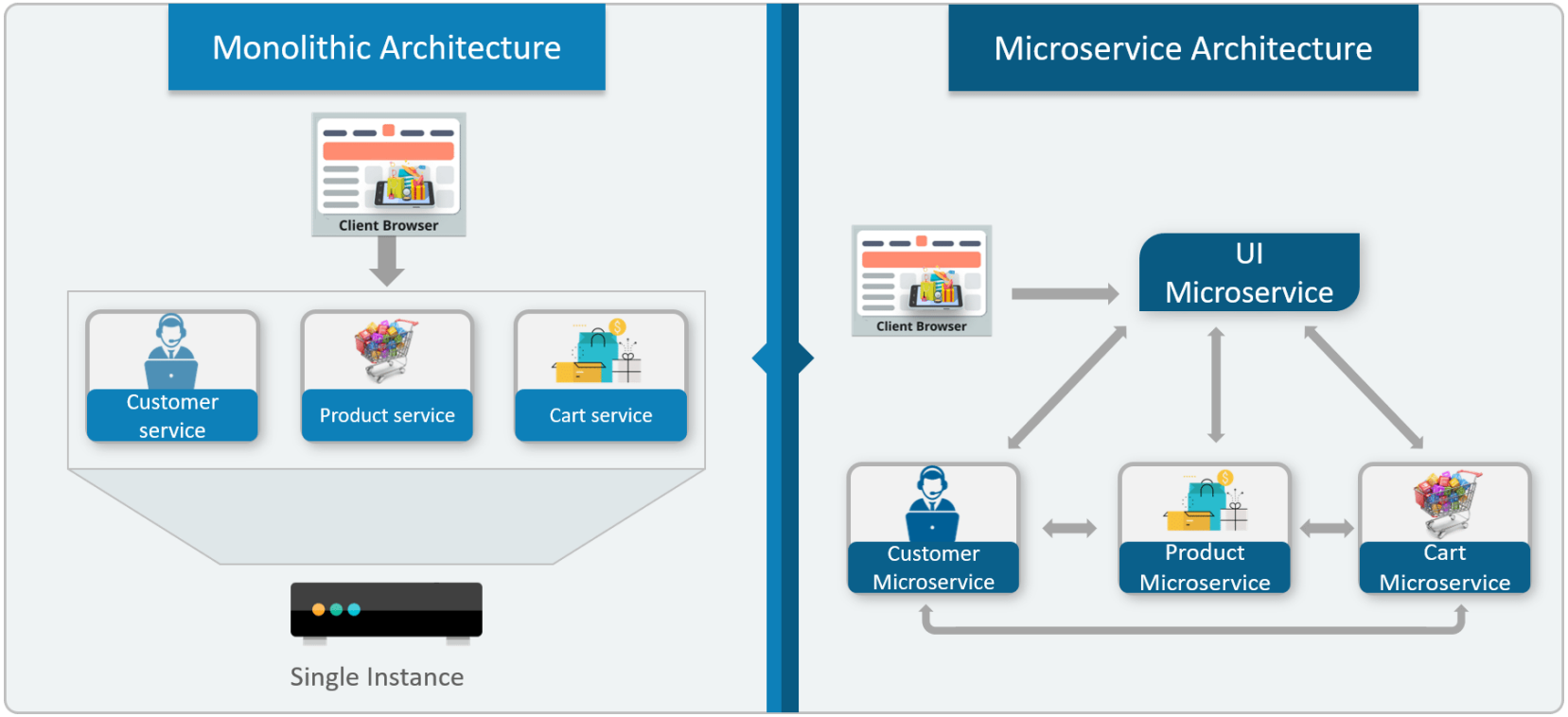
Microservices, aka ***Microservice Architecture***, is an architectural style that structures an application as a collection of small autonomous services, modeled around a **business domain.**



In Microservice Architecture, each service is **self-contained** and implements a **single business capability.**

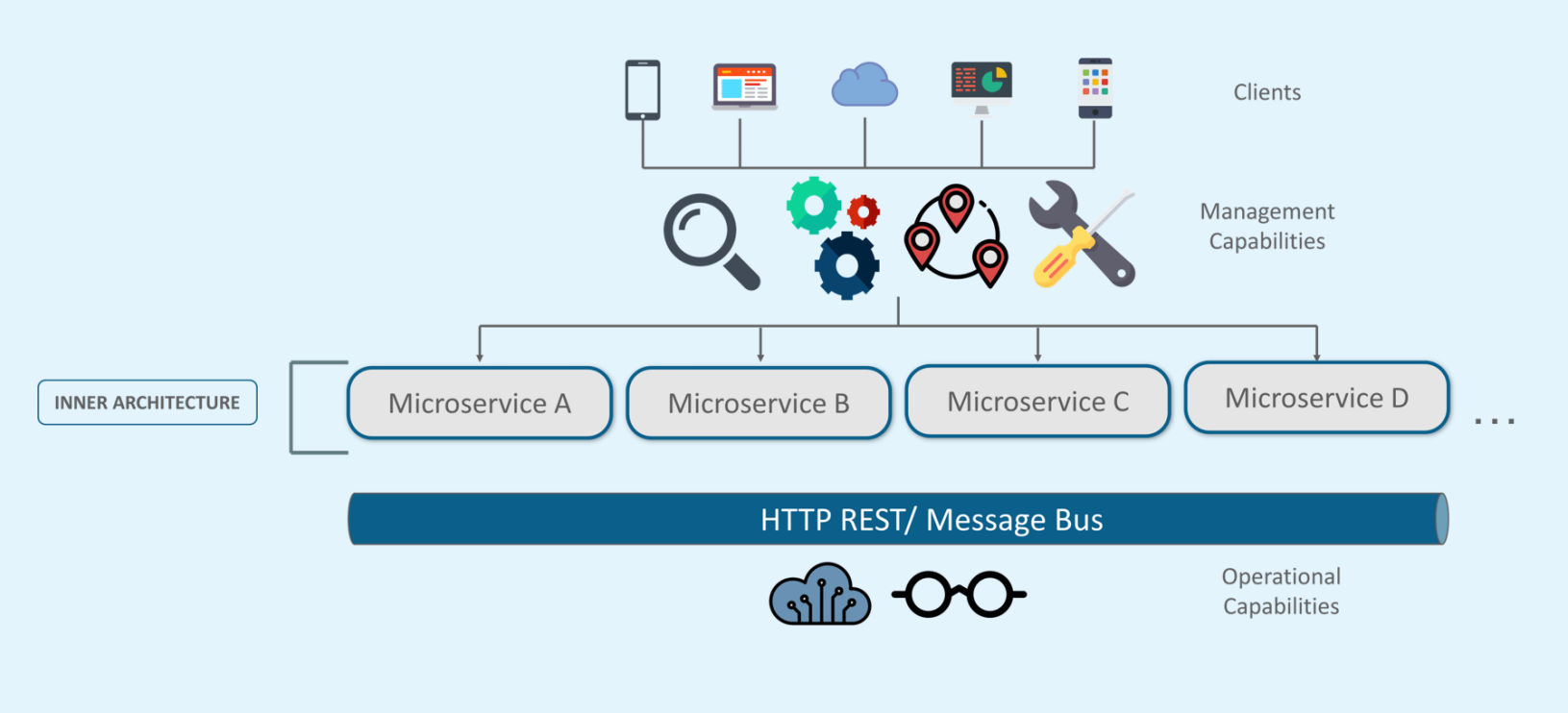
Differences Between Traditional Architecture and Microservices

Consider an E-commerce application as a use-case to understand the difference between both of them.



The main difference we observe in the above diagram is that all the features initially were under a single instance sharing a single database. But then, with microservices, each feature was allotted a different microservice, handling their own data, and performing different functionalities.

Microservice Architecture



Microservice Architecture

Different clients from different devices try to use different services like search, build, configure and other management capabilities

All the services are separated based on their domains and functionalities and are further allotted to individual microservices

These microservices have their own **load balancer** and **execution environment** to execute their functionalities & at the same time captures data in their own databases

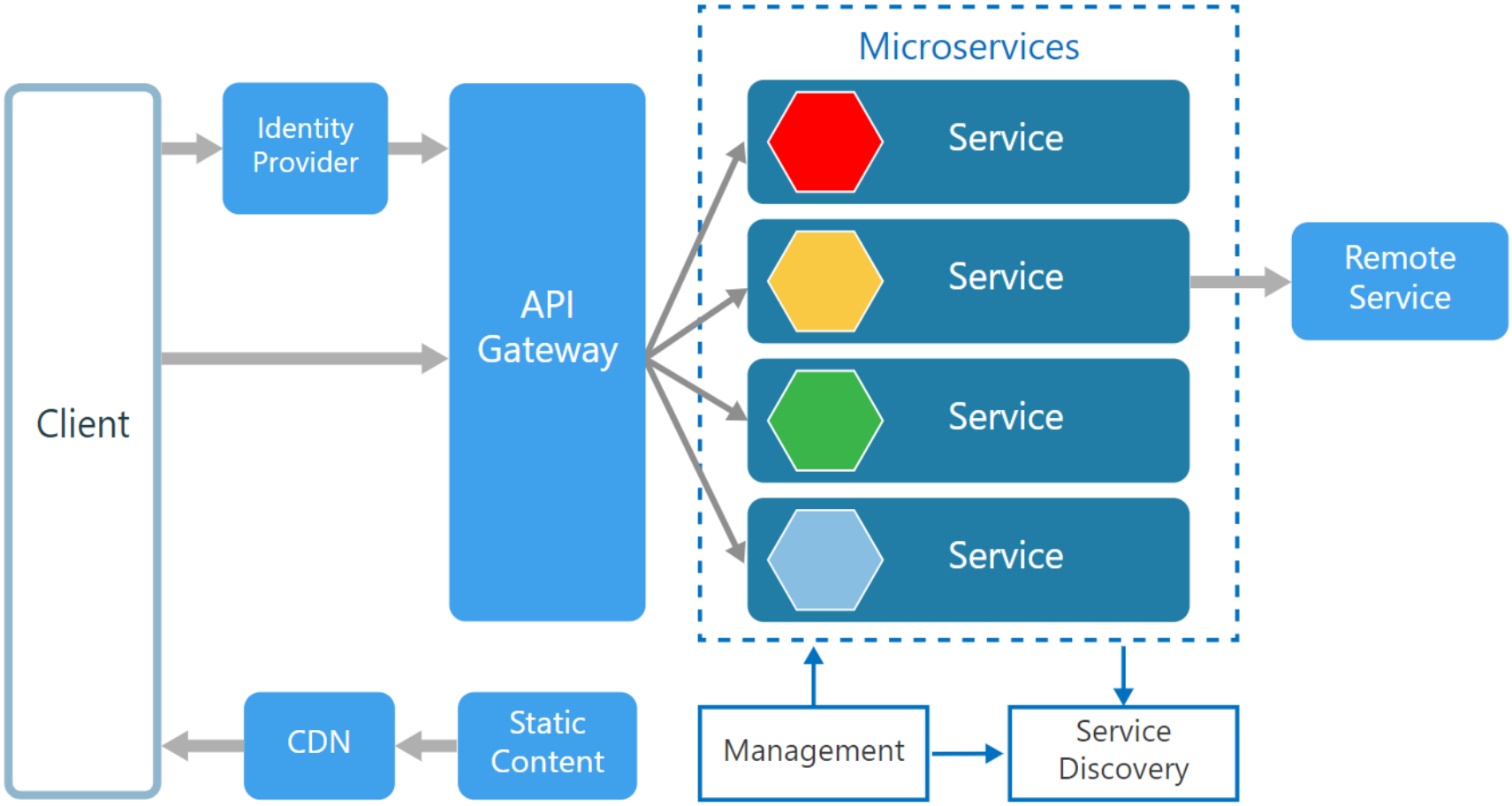
All the microservices communicate with each other through a stateless server which is either **REST** or **Message Bus**

Microservices know their path of communication with the help of **Service Discovery** and perform operational capabilities such as automation, monitoring

Then all the functionalities performed by microservices are communicated to clients via **API Gateway**

All the internal points are connected from the API Gateway. So, anybody who connects to the API Gateway automatically gets connected to the complete system

Microservices Architectures



- A microservices architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability. Here are some of the defining characteristics of microservices:
- In a microservices architecture, services are small, independent, and loosely coupled.
- A microservice is small enough that a single small team of developers can write and maintain it.
- Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
- Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.
- Services don't need to share the same technology stack, libraries, or frameworks.

Why build microservices?

Microservices can provide a number of useful benefits:

- **Agility.** Because microservices are deployed independently, it's easier to manage bug fixes and feature releases. You can update a service without redeploying the entire application, and roll back an update if something goes wrong. In many traditional applications, if a bug is found in one part of the application, it can block the entire release process; as a result, new features may be held up waiting for a bug fix to be integrated, tested, and published.

- **Small code, small teams.** A microservice should be small enough that a single feature team can build, test, and deploy it. Small code bases are easier to understand. In a large monolithic application, there is a tendency over time for code dependencies to become tangled, so that adding a new feature requires touching code in a lot of places. By not sharing code or data stores, a microservices architecture minimizes dependencies, and that makes it easier to add new features. Small team sizes also promote greater agility. The "two-pizza rule" says that a team should be small enough that two pizzas can feed the team. Obviously that's not an exact metric and depends on team appetites! But the point is that large groups tend be less productive, because communication is slower, management overhead goes up, and agility diminishes.

- **Mix of technologies**. Teams can pick the technology that best fits their service, using a mix of technology stacks as appropriate.
- **Resiliency**. If an individual microservice becomes unavailable, it won't disrupt the entire application, as long as any upstream microservices are designed to handle faults correctly (for example, by implementing circuit breaking).

- **Scalability**. A microservices architecture allows each microservice to be scaled independently of the others. That lets you scale out subsystems that require more resources, without scaling out the entire application. If you deploy services inside containers, you can also pack a higher density of microservices onto a single host, which allows for more efficient utilization of resources.
- **Data isolation**. It is much easier to perform schema updates, because only a single microservice is affected. In a monolithic application, schema updates can become very challenging, because different parts of the application may all touch the same data, making any alterations to the schema risky.

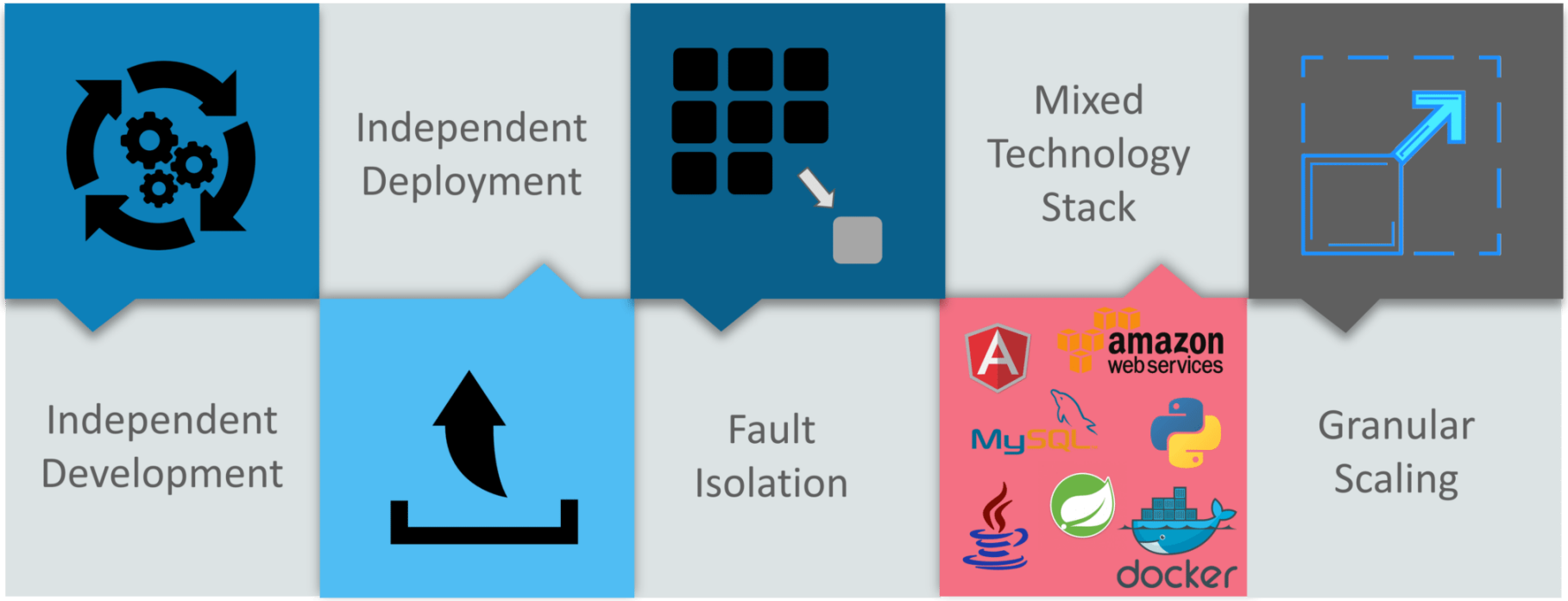
When should I build microservices?

- Consider a microservices architecture for:
- Large applications that require a high release velocity.
- Complex applications that need to be highly scalable.
- Applications with rich domains or many subdomains.
- An organization that consists of small development teams.

Microservices Features

- **Decoupling** – Services within a system are largely decoupled. So the application as a whole can be easily built, altered, and scaled
- **Componentization** – Microservices are treated as independent components that can be easily replaced and upgraded
- **Business Capabilities** – Microservices are very simple and focus on a single capability
- **Autonomy** – Developers and teams can work independently of each other, thus increasing speed
- **Continous Delivery** – Allows frequent releases of software, through systematic automation of software creation, testing, and approval
- **Responsibility** – Microservices do not focus on applications as projects. Instead, they treat applications as products for which they are responsible
- **Decentralized Governance** – The focus is on using the right tool for the right job. That means there is no standardized pattern or any technology pattern. Developers have the freedom to choose the best useful tools to solve their problems
- **Agility** – Microservices support agile development. Any new feature can be quickly developed and discarded again

Advantages Of Microservices

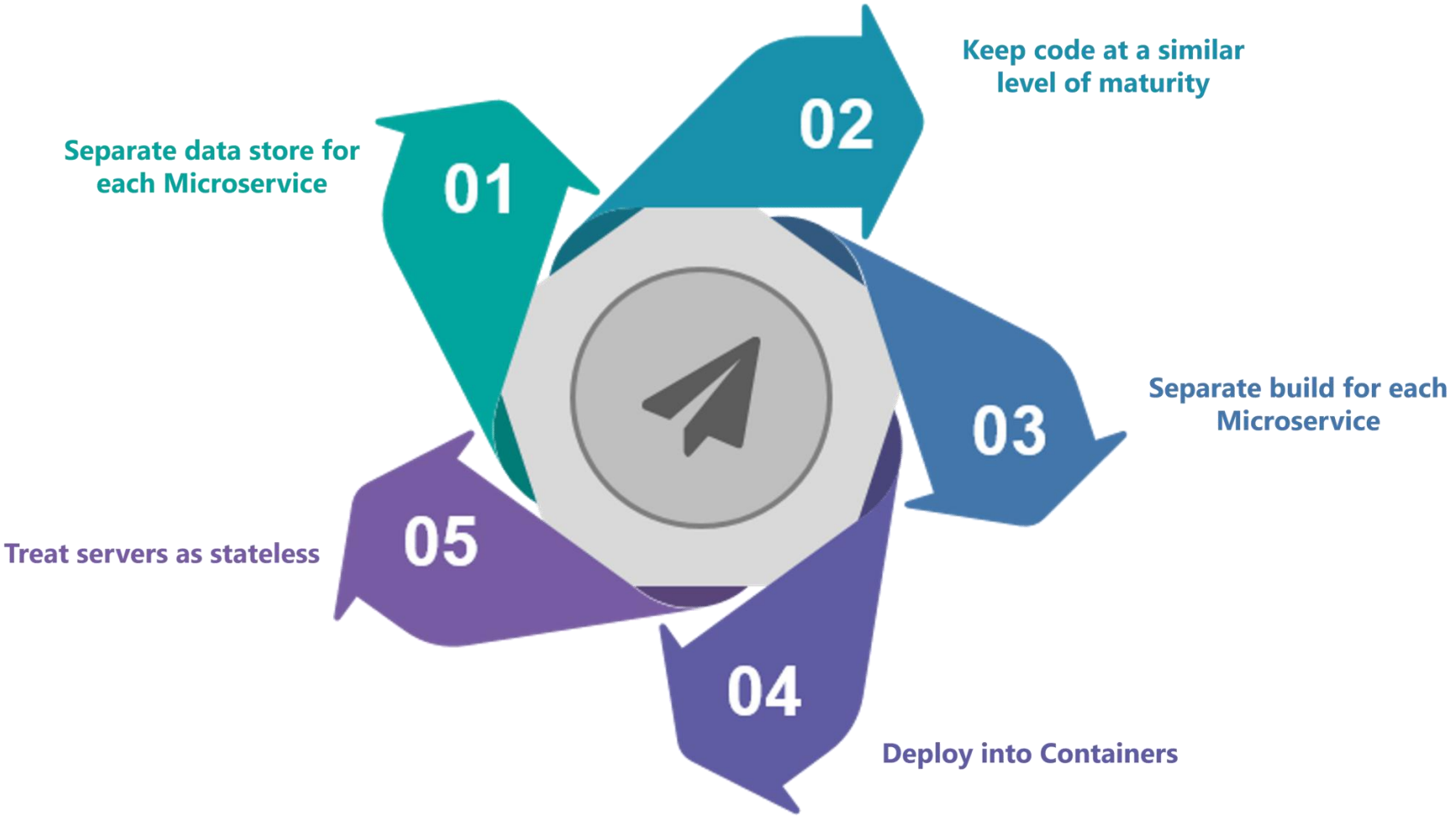


- **Independent Development** – All microservices can be easily developed based on their individual functionality
- **Independent Deployment** – Based on their services, they can be individually deployed in any application
- **Fault Isolation** – Even if one service of the application does not work, the system still continues to function
- **Mixed Technology Stack** – Different languages and technologies can be used to build different services of the same application
- **Granular Scaling** – Individual components can scale as per need, there is no need to scale all components together

Best Practices To Design Microservices

In today's world, complexity has managed to creep into products. Microservice architecture promises to keep teams scaling and function better.

The following are the best practices to design microservices:

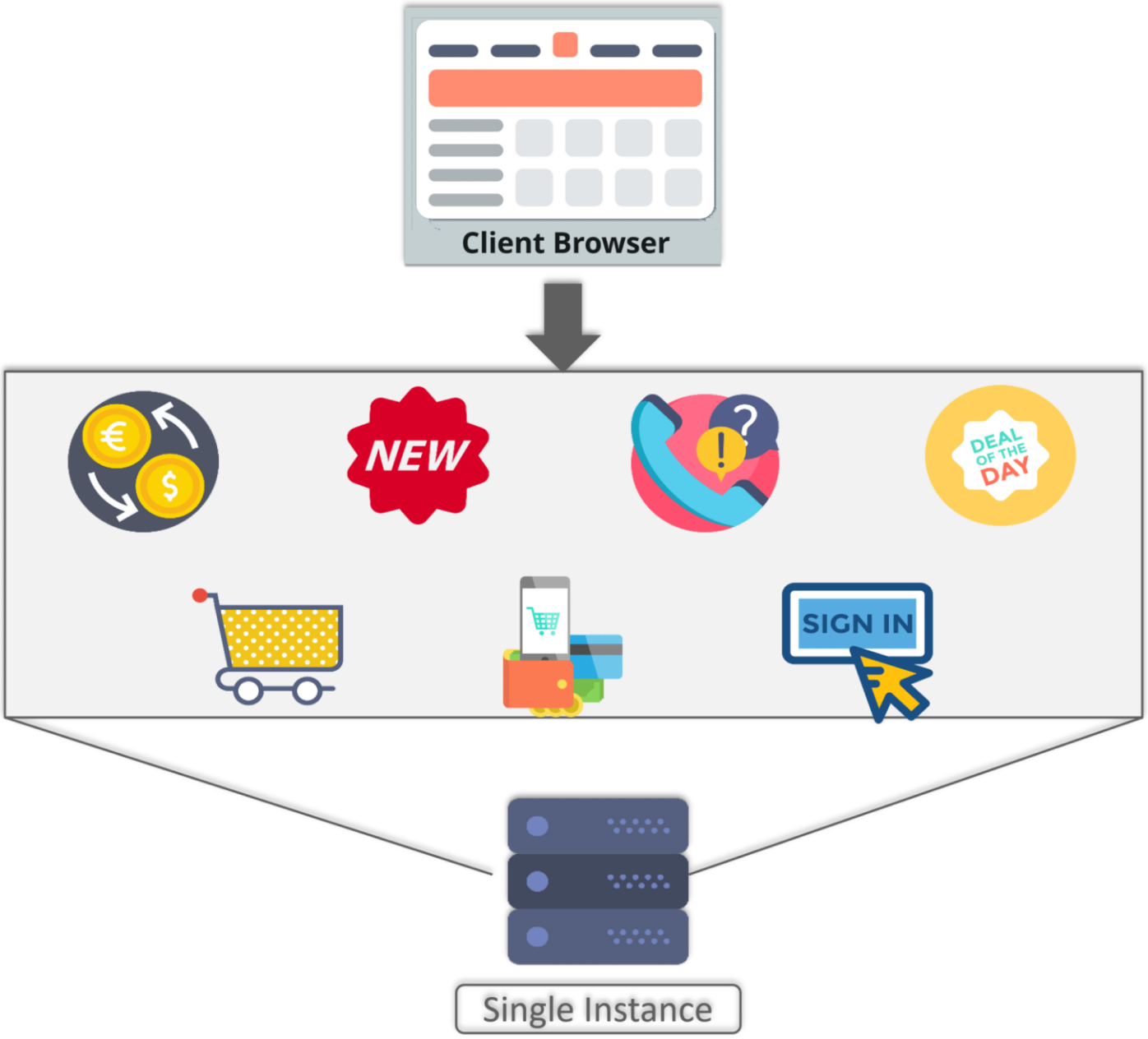


Use-Case: Shopping Cart Application

Let’s take a classic use case of a shopping cart application.

When you open a shopping cart application, all you see is just a website. But, behind the scenes, the shopping cart application has a service for accepting payments, a service for customer services and so on.

Assume that developers of this application have created it in a monolithic framework.Refer to the diagram below:


Client Browser
NEW
DEAL OF THE DAY
SIGN IN
Single Instance

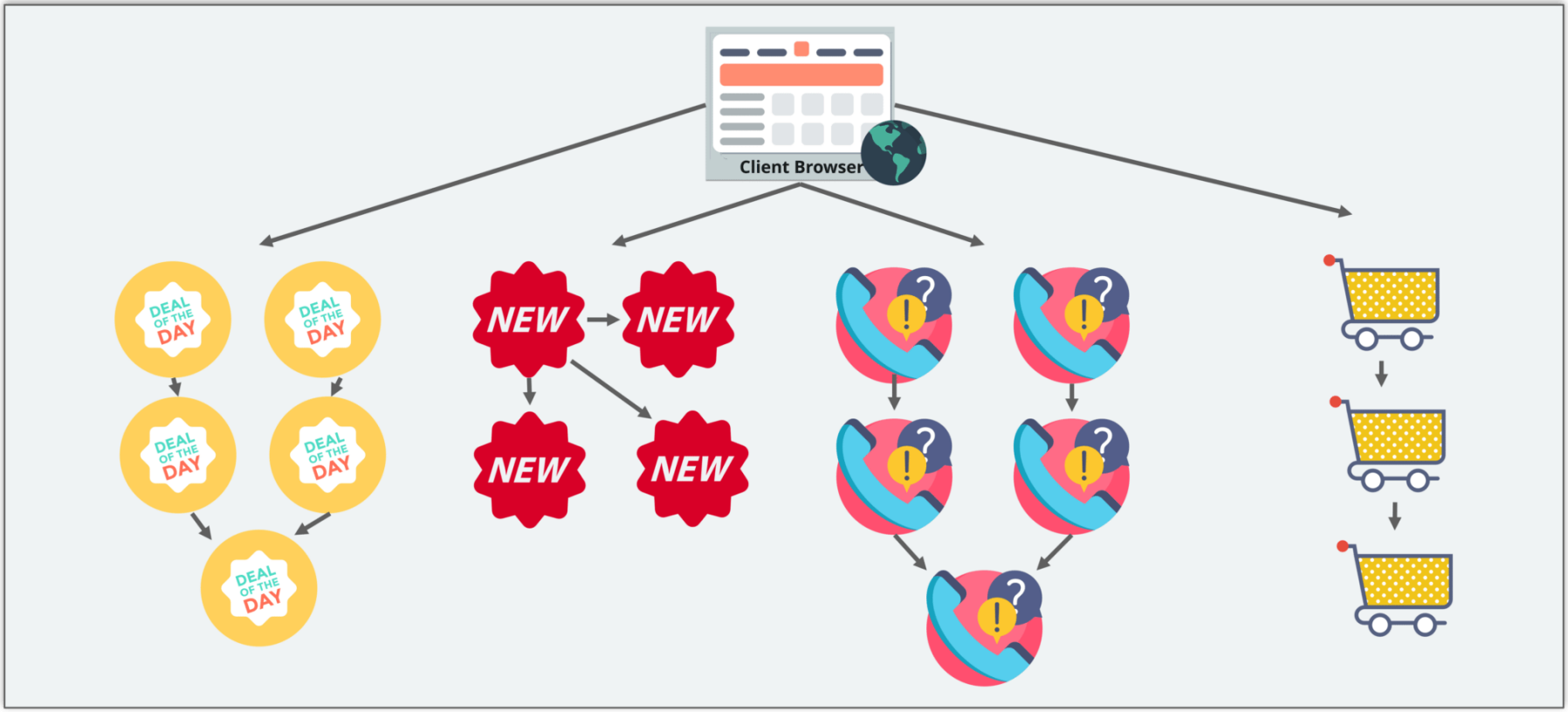
What Is Microservices – Monolithic Framework Of Shopping Cart Application

So, all the features are put together in a single code base and are under a single underlying database.

Now, let's suppose that there is a new brand coming up in the market and developers want to put all the details of the upcoming brand in this application.

Then, they not only have to rework on the service for new labels, but they also have to reframe the complete system and deploy it accordingly.

To avoid such challenges developers of this application decided to shift their application from a monolithic architecture to microservices.Refer to the diagram below to understand the microservices architecture of shopping cart application


Client Browser
DEAL OF THE DAY
DEAL OF THE DAY
DEAL OF THE DAY
DEAL OF THE DAY
DEAL OF THE DAY
NEW
NEW
NEW
NEW

Microservice Architecture Of Shopping Cart Application

This means that developers don’t create a web microservice, a logic microservice, or a database microservice. Instead, they create separate microservices for search, recommendations, customer services and so on.

This type of architecture for the application not only helps the developers to overcome all the challenges faced with the previous architecture but also helps the shopping cart application to be built, deployed, and scale up easily.

Companies using Microservices

There is a long list of companies using Microservices to build applications, these are just to name a few:

Challenges

The benefits of microservices don't come for free. Here are some of the challenges to consider before embarking on a microservices architecture.

Complexity. A microservices application has more moving parts than the equivalent monolithic application. Each service is simpler, but the entire system as a whole is more complex.

Development and testing. Writing a small service that relies on other dependent services requires a different approach than a writing a traditional monolithic or layered application. Existing tools are not always designed to work with service dependencies. Refactoring across service boundaries can be difficult. It is also challenging to test service dependencies, especially when the application is evolving quickly.

Lack of governance. The decentralized approach to building microservices has advantages, but it can also lead to problems. You may end up with so many different languages and frameworks that the application becomes hard to maintain. It may be useful to put some project-wide standards in place, without overly restricting teams' flexibility. This especially applies to cross-cutting functionality such as logging.

Network congestion and latency. The use of many small, granular services can result in more interservice communication. Also, if the chain of service dependencies gets too long (service A calls B, which calls C...), the additional latency can become a problem. You will need to design APIs carefully. Avoid overly chatty APIs, think about serialization formats, and look for places to use asynchronous communication patterns.

Data integrity. With each microservice responsible for its own data persistence. As a result, data consistency can be a challenge. Embrace eventual consistency where possible.

Management. To be successful with microservices requires a mature DevOps culture. Correlated logging across services can be challenging. Typically, logging must correlate multiple service calls for a single user operation.

Versioning. Updates to a service must not break services that depend on it. Multiple services could be updated at any given time, so without careful design, you might have problems with backward or forward compatibility.

Skillset. Microservices are highly distributed systems. Carefully evaluate whether the team has the skills and experience to be successful.

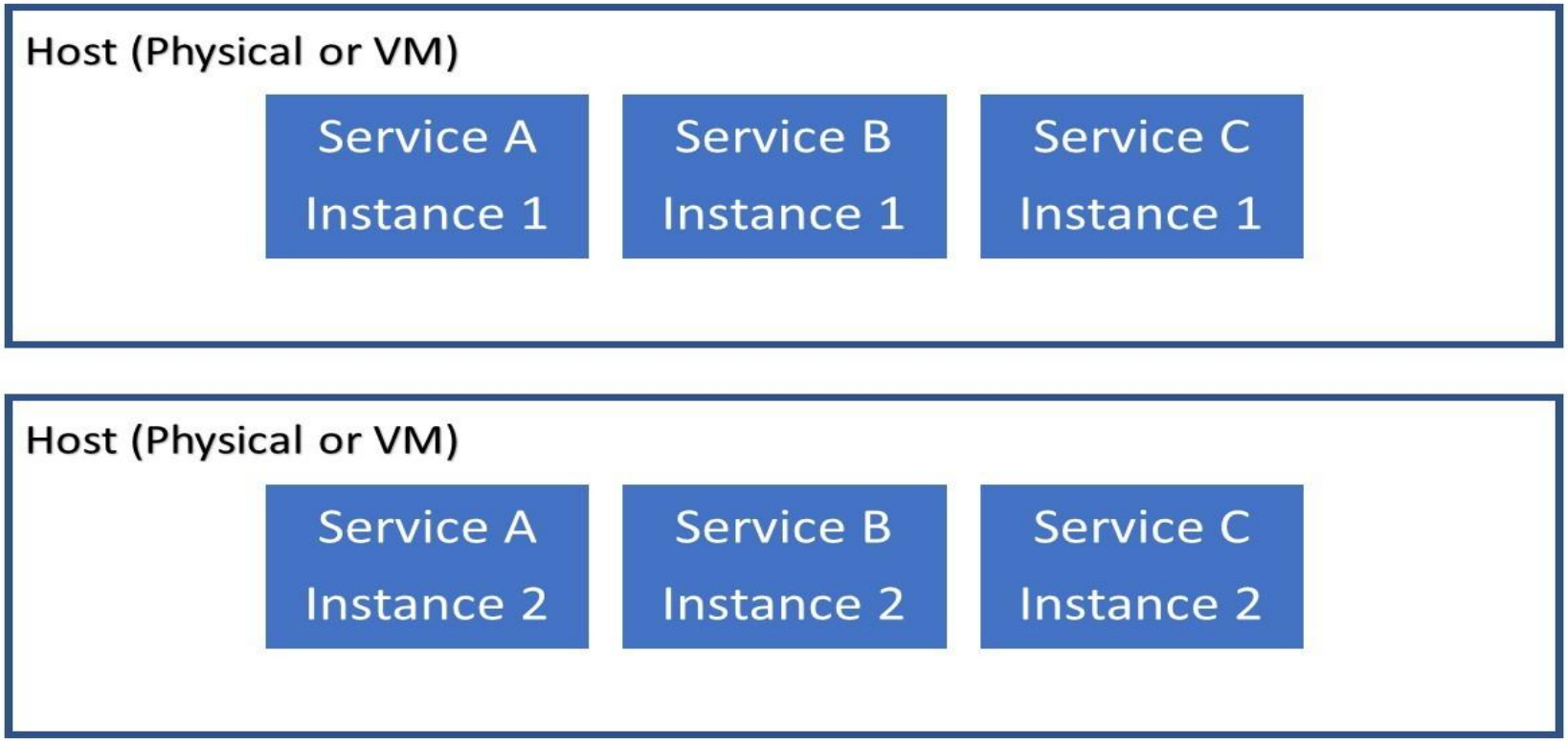
Microservices Deployment Patterns

Once you have created your Microservices you need to deploy them. There are various forces or issues you need to think about while deploying a Microservice :

- Services may be written using a variety of different languages, they can be using different frameworks or different versions of frameworks. For example, different versions of Java or Spring frameworks
- Each service runtime consists of multiple service instances. For example, in an application, a piece of code called catalog service may have to run multiple copies during runtime for handling the load and high availability as we need as much remaining instances to take over and handle the load of the requests in case of failure of one instance.

- The whole process of building and deploying the services need to be fast. In case some make a change, it needs to be automatically tested and deployed fast
- Services need to be deployed and scaled independently, which is one of the primary motivations behind microservices.
- Service instances need to be isolated from one another. In case one service is misbehaving ideally, we don't want it to impact any other services
- We should be able to constrain the resources a given service uses. We should be able to define how much CPU, Memory or bandwidth a particular service can use to avoid a single service eating up all the available resource
- The whole process of deploying changes to the production needs to be reliable. That should be an easy, stress-free process that shouldn't ever fail
- Deployment should be cost-effective
- There are various different patterns for deploying your services

Multiple Service Instances Per Host



It is a fairly traditional approach where you have a machine which could be physical or virtual and you run multiple service instances on this host. Each service instance could be a process like JVM or Tomcat instance, or each process may be comprised of multiple service instances. For example, each service instance could be a var file and you run multiple var files on one tomcat

Some of the benefits of using this pattern are

- Efficient resource utilization: You have one machine with multiple services and the whole resource is shared between these different services
- Fast deployment: In order to deploy a new version of your service you simply have to copy that to the machine and restart the process

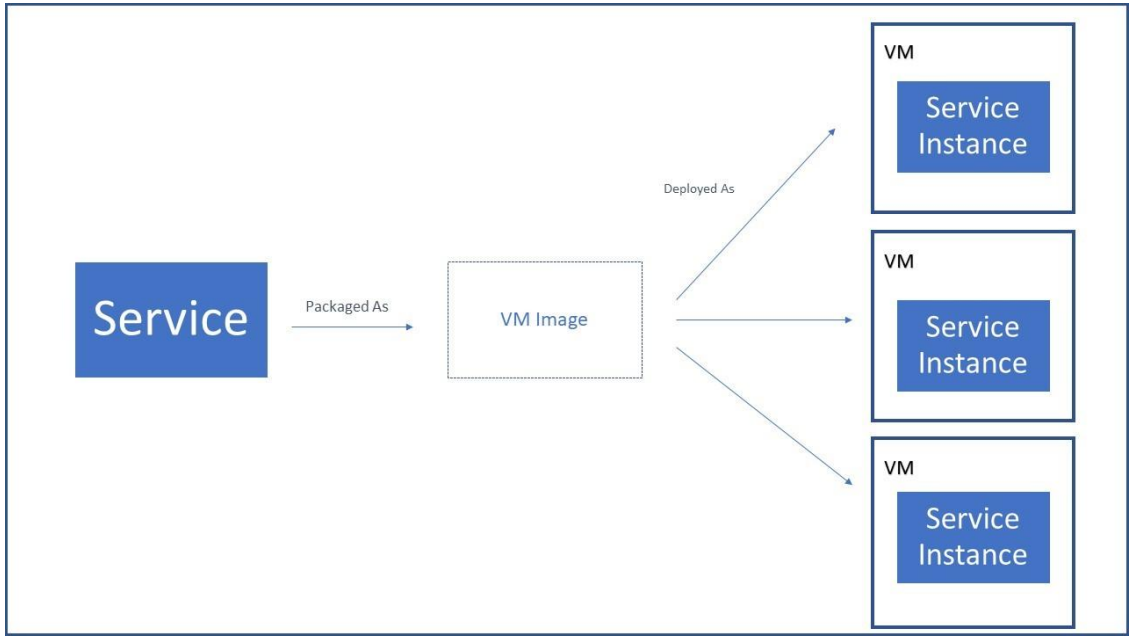
Some of the drawbacks of using this pattern are :

- You get little to no isolation between various service instances
- Poor visibility on how the services are behaving like memory and CPU utilization
- Difficult to constrain the resources a particular service can use
- Risk of dependency version conflicts
- Poor encapsulation of implementation technology. Whoever deploying the service has to have the detailed knowledge of the technology stack and mechanism used by each service

Service Instance Per Host

To have a greater isolation and manageability we can host a single service instance in a single host at the cost of resource utilization. This pattern can be achieved by two different ways: Service Instance per Virtual Machine and Service Instance per Container

Service Instance Per VM



Here you package each service as a virtual machine image and deploying each instance as a virtual machine instance. One of the example is Netflix and their video service is consisting of more than 600 services with many instance of each service

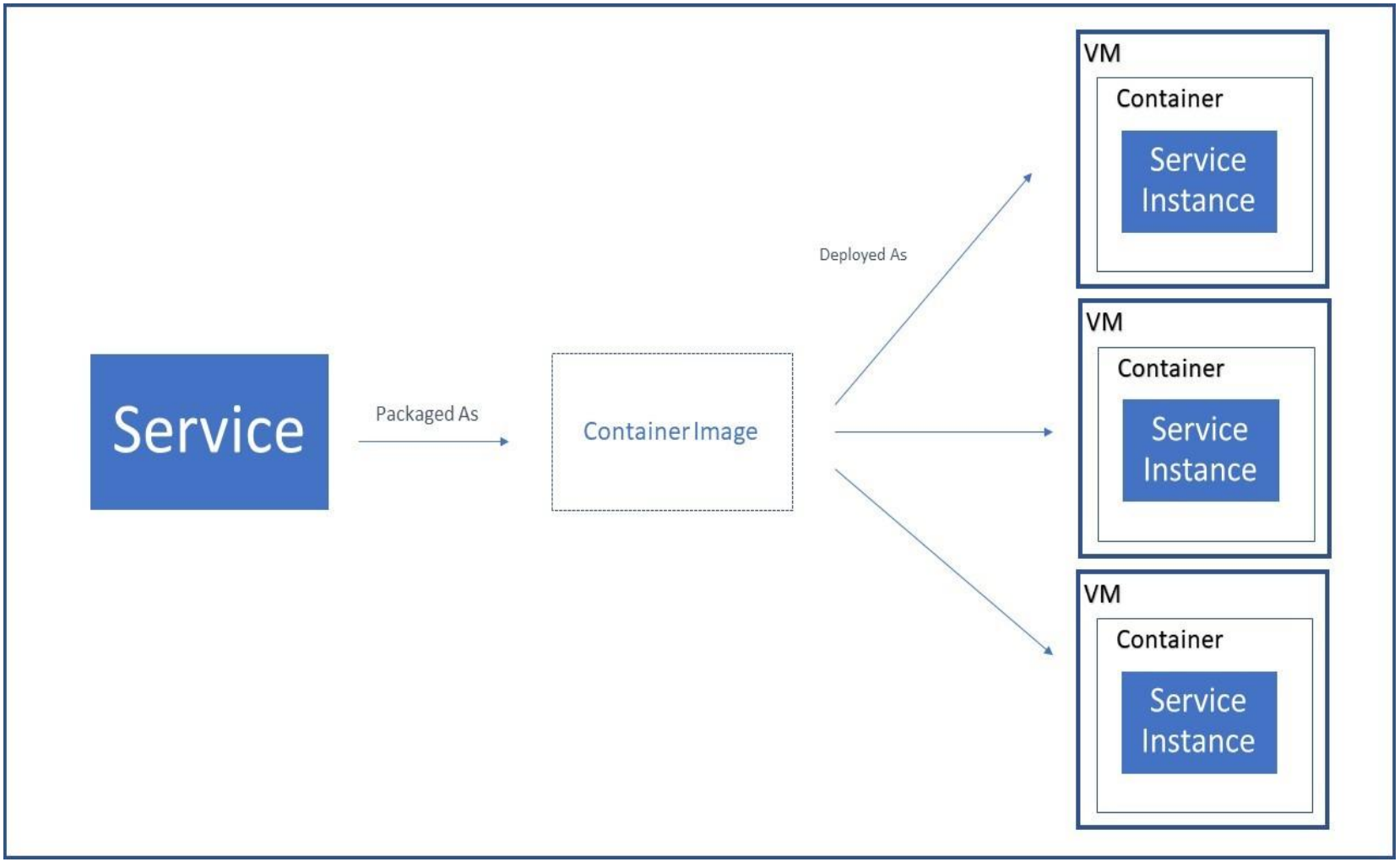
Some of the benefits of packaging services as virtual machines are :

- You get great isolation as the virtual machines are highly isolated with each other
- You get greater manageability as virtual machine environment has great tools for management
- Virtual machine encapsulates the implementation technology for the microservice. Once you package the service, the API for starting and stopping the service is your virtual machine interface. The deployment team need not to be knowledgeable about the implementation technology
- With this pattern you can leverage cloud infrastructure for auto-scaling and load balancing

Some of the drawbacks of using this pattern are :

- With virtual machines, we get less efficient resource utilization and thus increases the cost
- The process of building a virtual machine image is relatively slow because of the volume of the data needed to be moved around the network and thus slows the deployment process
- In some cases, it will take some time for the virtual machines to boot up

Service Instance Per Container



Here in this pattern the service is packaged as a container image and is deployed as running container where you can have multiple containers running on the same virtual machine.

Some of the benefits of packaging services as container are :

Greater isolation as each container is highly isolated from each other at OS level

Greater manageability with containers

Container encapsulates the implementation technology

As containers are lightweight and sharing the same resources, it has efficient resource utilization

Fast deployments

Some of the drawbacks of packaging services as container are :

As the container technology is new, it is not that much mature compared to virtual machines

Containers are not as secured as VMs

Serverless Deployment

The serverless deployment hides the underlying infrastructure and it will take your service's code just run it. It is charged on usage on how many requests it processed and based on how much resources utilized for processing each request. To use this pattern, you need to package your code and select the desired performance level. Various public cloud providers are offering this service where they use containers or virtual machines to isolate the services which are hidden from you. In this system, you are not authorized to manage any low-level infrastructure such as servers, operating system, virtual machines or containers. AWS Lambda, Google Cloud Functions, Azure Functions are few serverless deployment environments. We can call a serverless function directly using a web service request or in response to an event or via an API gateway or can run periodically according to cron-like schedule.

Some of the benefits of serverless deployments are :

You can focus on your code and application and need not to be worried about the underlying infrastructure

You need not worry about scaling as it will automatically scale in case of load

You pay for what you use and not for the resources provisioned for you

Some of the drawbacks of serverless deployments are :

Many constraints and limitations like limited languages support, suits better only for stateless services

Can only respond to request from a limited set of input sources

Only suited for applications that can start quickly

Chance of high latency in case of a sudden spike in load

Service Deployment Platform

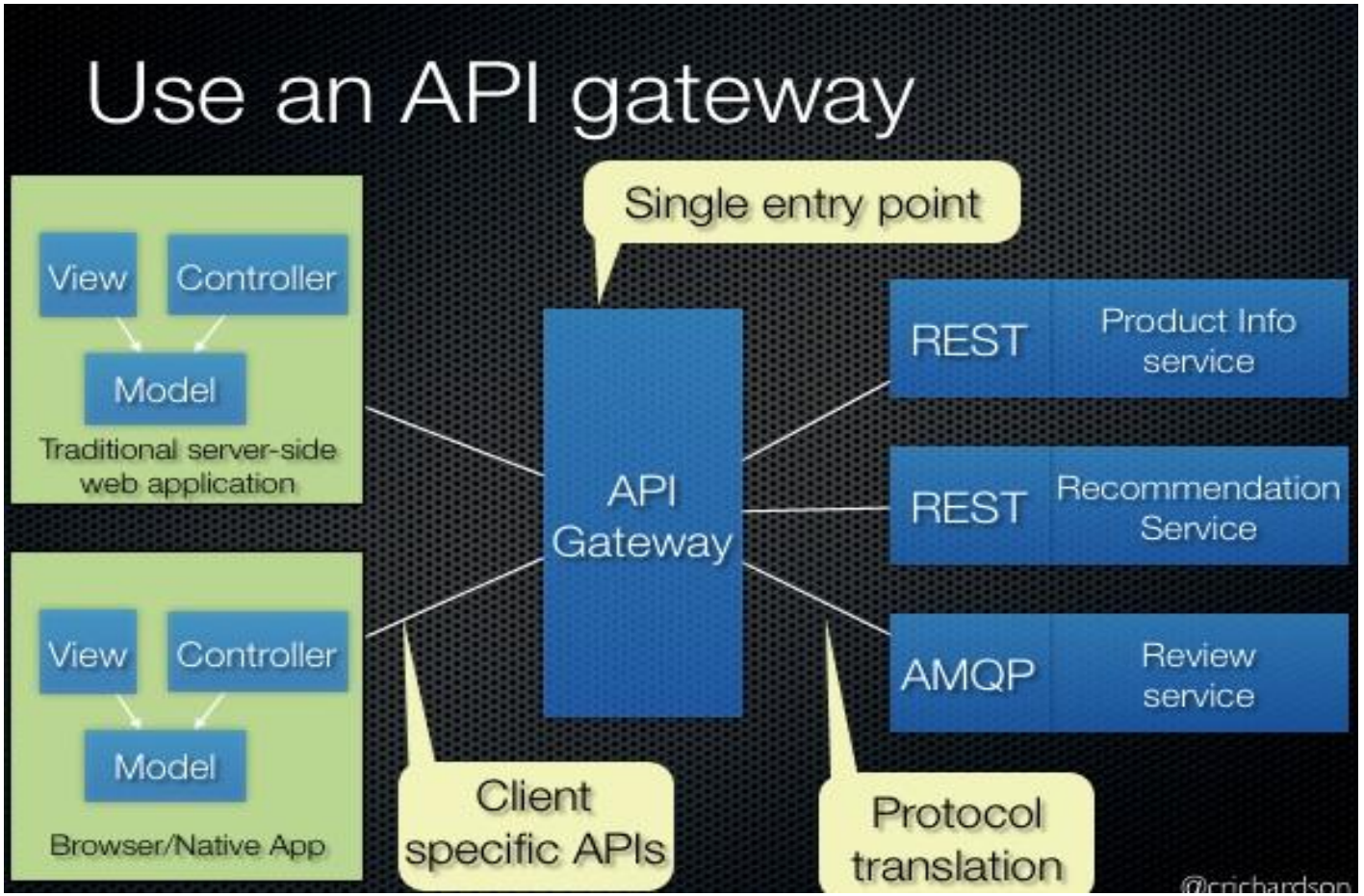
This pattern uses a deployment platform, which is automated infrastructure for application deployment. It provides a service abstraction, which is a named, set of highly available (e.g. load balanced) service instances. Some of the examples of this type are Docker swarm mode, Kubernetes, AWS Lambda, Cloud Foundry, and AWS Elastic Beanstalk

How do the clients of a Microservices-based application access the individual services

- The granularity of APIs provided by microservices is often different than what a client needs. Microservices typically provide fine-grained APIs, which means that clients need to interact with multiple services.
- Different clients need different data.
- Network performance is different for different types of clients.
- The number of service instances and their locations (host+port) changes dynamically
- Partitioning into services can change over time and should be hidden from clients
- Services might use a diverse set of protocols, some of which might not be web friendly

Solution

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.

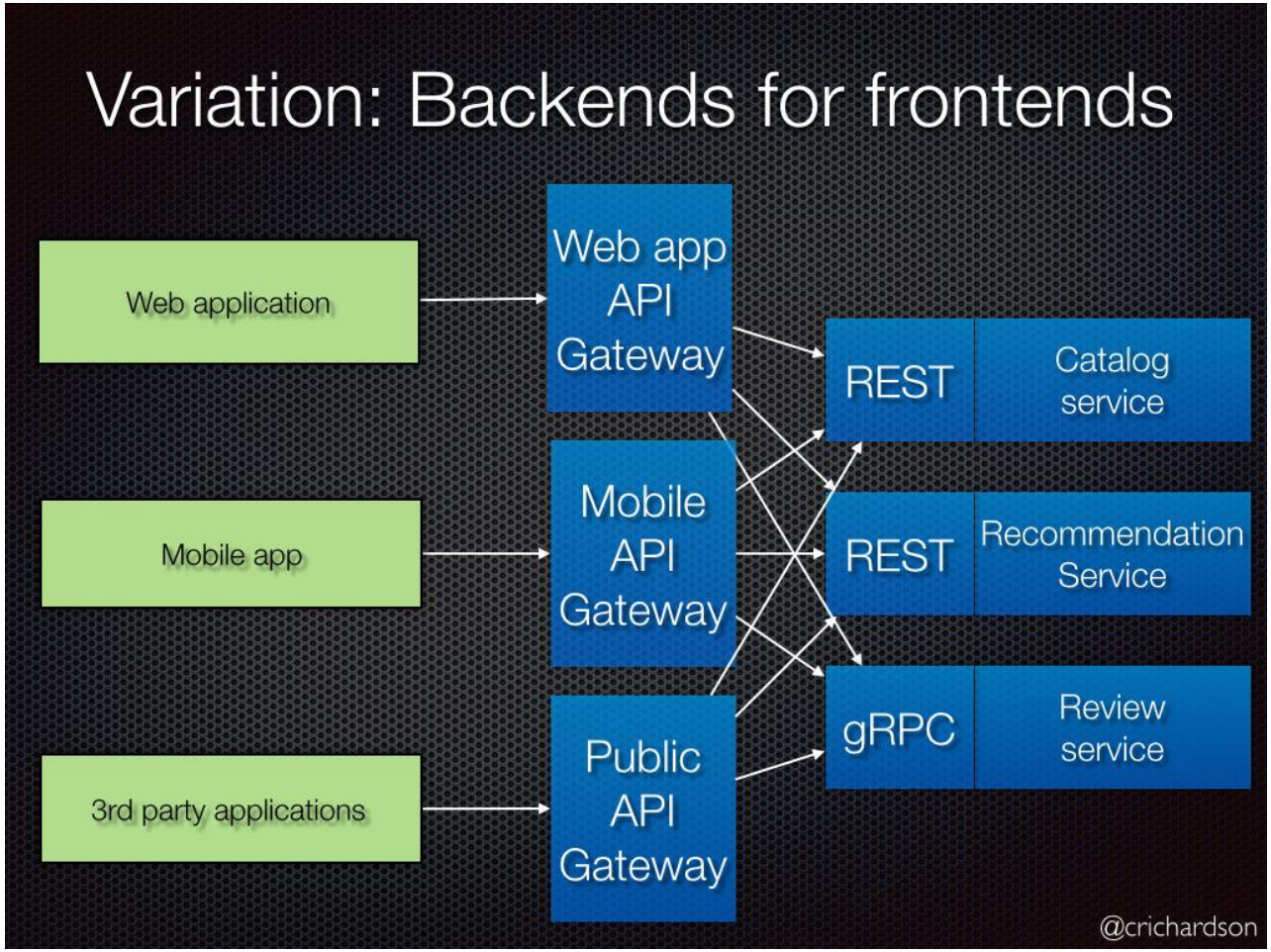


Rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client. For example, the Netflix API gateway runs client-specific adapter code that provides each client with an API that's best suited to its requirements.

The API gateway might also implement security, e.g. verify that the client is authorized to perform the request

Variation: 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 above example, there are three kinds of clients: web application, mobile application, and external 3rd party application. There are three different API gateways. Each one is provides an API for its client.

Using an API gateway has the following 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

The API gateway pattern has some drawbacks:

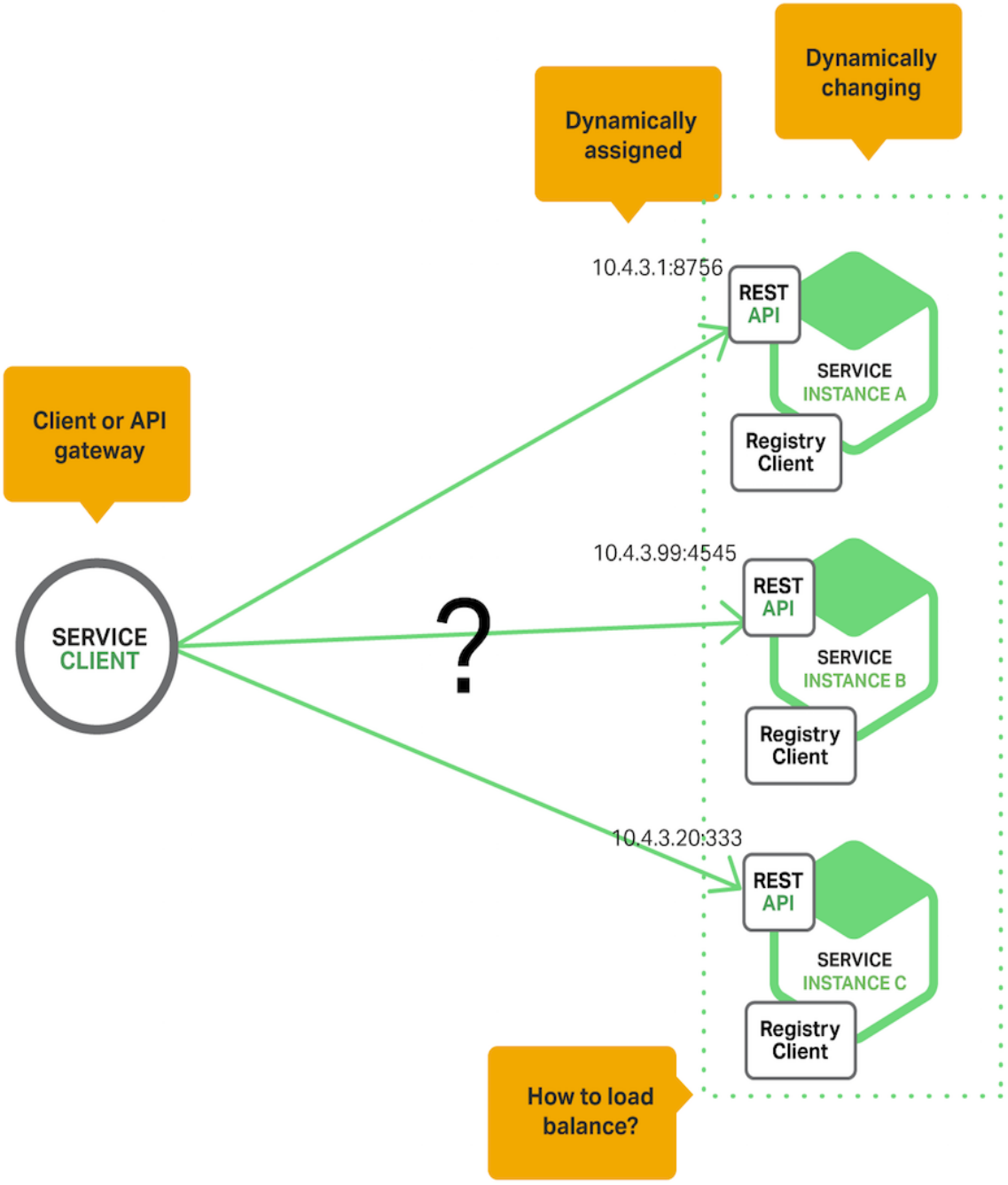
- Increased complexity - the API gateway is yet another moving part that must be developed, deployed and managed
- Increased response time due to the additional network hop through the API gateway - however, for most applications the cost of an extra roundtrip is insignificant

Service Discovery in a Microservices Architecture

Why Use Service Discovery?

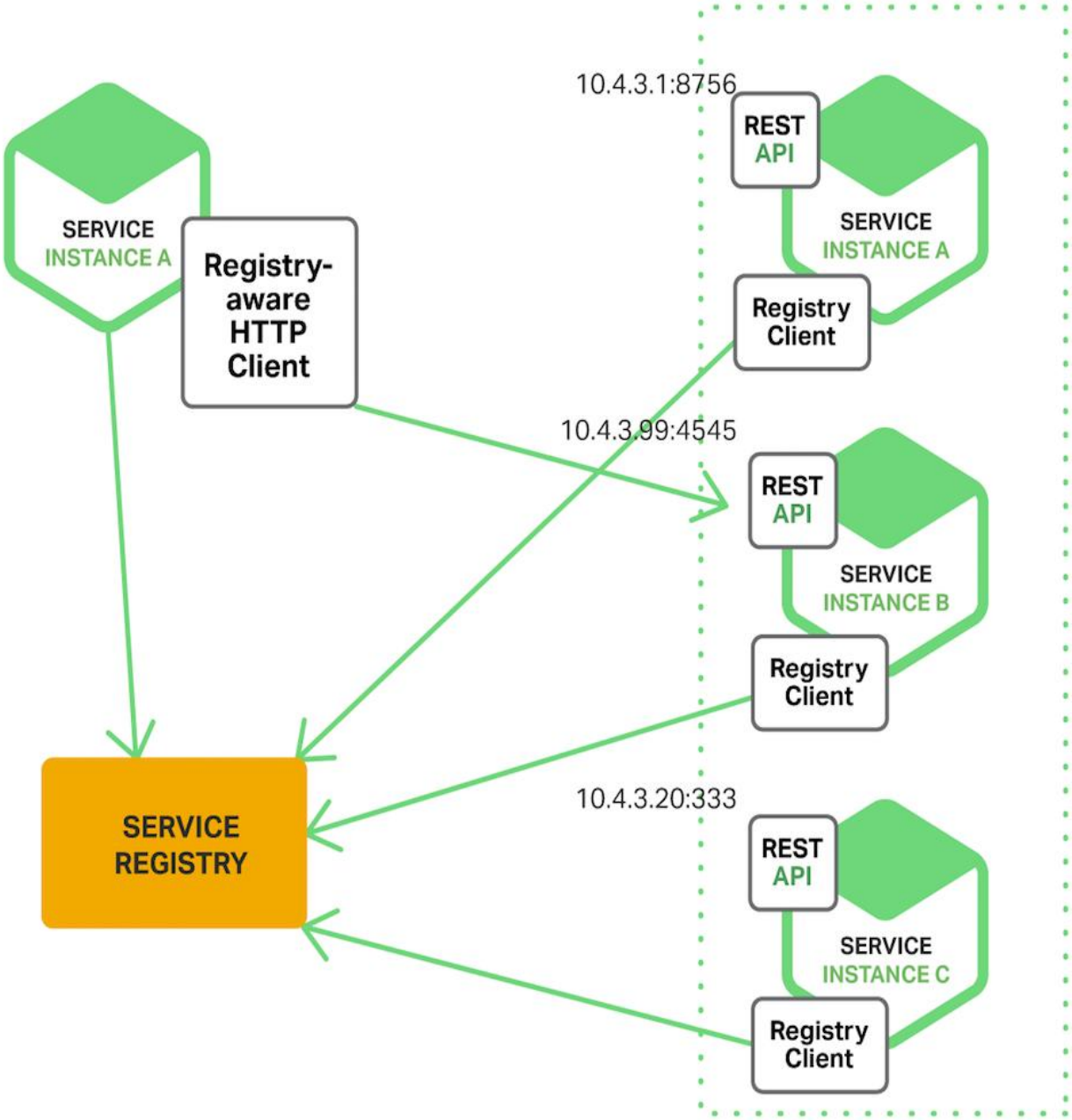
Let’s imagine that you are writing some code that invokes a service that has a REST API or Thrift API. In order to make a request, your code needs to know the network location (IP address and port) of a service instance. In a traditional application running on physical hardware, the network locations of service instances are relatively static. For example, your code can read the network locations from a configuration file that is occasionally updated.

In a modern, cloud-based microservices application, however, this is a much more difficult problem to solve


Dynamically changing
Dynamically assigned
10.4.3.1:8756
REST API
SERVICE INSTANCE A
Registry Client
Client or API gateway
SERVICE CLIENT
?
10.4.3.99:4545
REST API
SERVICE INSTANCE B
Registry Client
10.4.3.20:333
REST API
SERVICE INSTANCE C
Registry Client
How to load balance?

The Client-Side Discovery Pattern

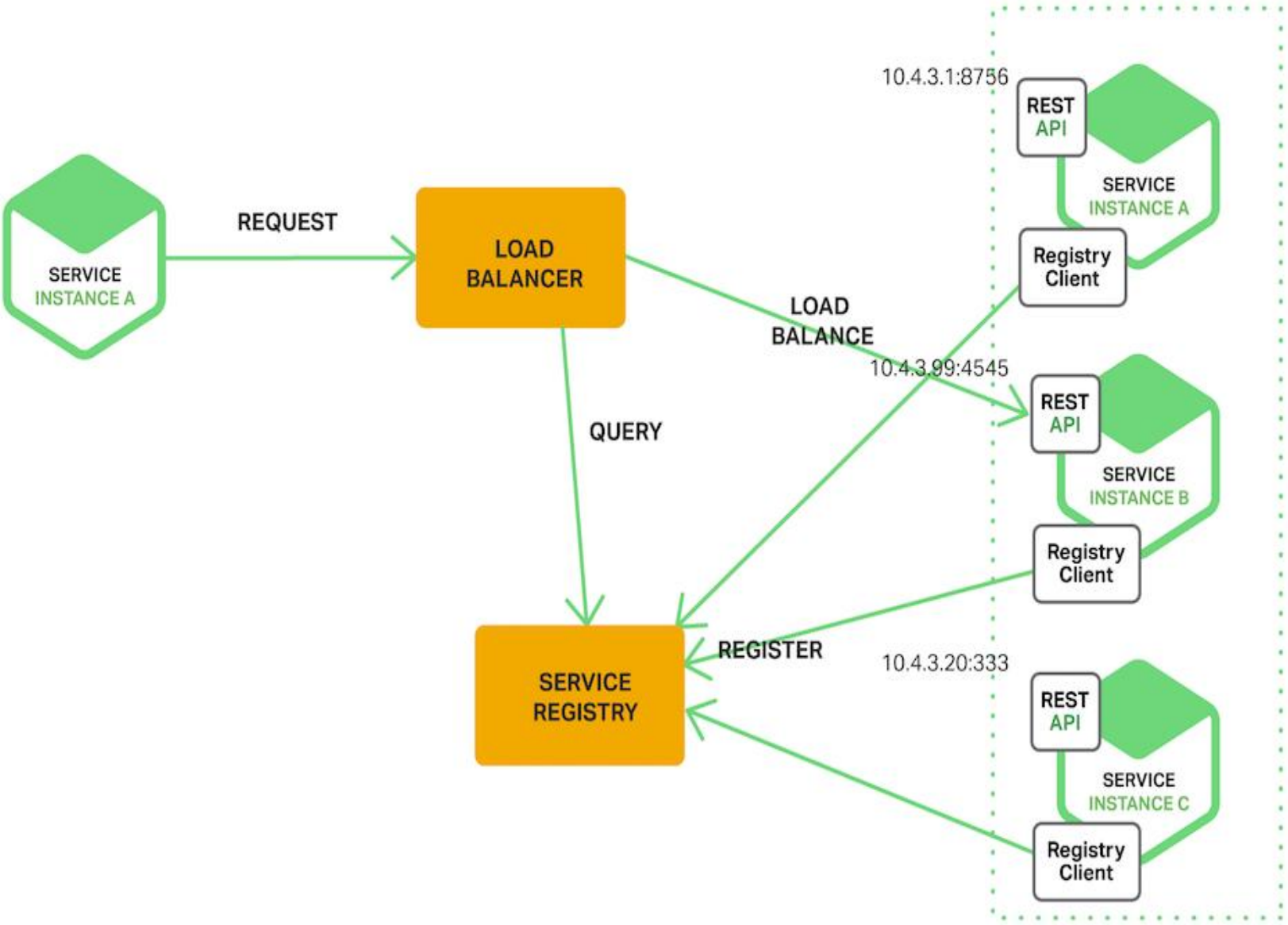
When using client-side discovery, the client is responsible for determining the network locations of available service instances and load balancing requests across them. The client queries a service registry, which is a database of available service instances. The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.


10.4.3.1:8756
REST API
SERVICE INSTANCE A
Registry Client
SERVICE INSTANCE A
Registry-aware HTTP Client
10.4.3.99:4545
REST API
SERVICE INSTANCE B
Registry Client
SERVICE REGISTRY
10.4.3.20:333
REST API
SERVICE INSTANCE C
Registry Client

The network location of a service instance is registered with the service registry when it starts up. It is removed from the service registry when the instance terminates. The service instance's registration is typically refreshed periodically using a heartbeat mechanism.

The Server-Side Discovery Pattern

The other approach to service discovery is the server-side discovery pattern. The following diagram shows the structure of this pattern.


SERVICE
INSTANCE A
REQUEST
LOAD
BALANCER
LOAD
BALANCE
QUERY
SERVICE
REGISTRY
REGISTER
10.4.3.1:8756
REST
API
SERVICE
INSTANCE A
Registry
Client
10.4.3.99:4545
REST
API
SERVICE
INSTANCE B
Registry
Client
10.4.3.20:333
REST
API
SERVICE
INSTANCE C
Registry
Client

The client makes a request to a service via a load balancer. The load balancer queries the service registry and routes each request to an available service instance. As with client-side discovery, service instances are registered and deregistered with the service registry.

The AWS Elastic Load Balancer (ELB) is an example of a server-side discovery router. An ELB is commonly used to load balance external traffic from the Internet. However, you can also use an ELB to load balance traffic that is internal to a virtual private cloud (VPC).

The server-side discovery pattern has several benefits and drawbacks. One great benefit of this pattern is that details of discovery are abstracted away from the client. Clients simply make requests to the load balancer. This eliminates the need to implement discovery logic for each programming language and framework used by your service clients. Also, as mentioned above, some deployment environments provide this functionality for free. **This pattern also has some drawbacks, however**. Unless the load balancer is provided by the deployment environment, it is yet another highly available system component that you need to set up and manage.