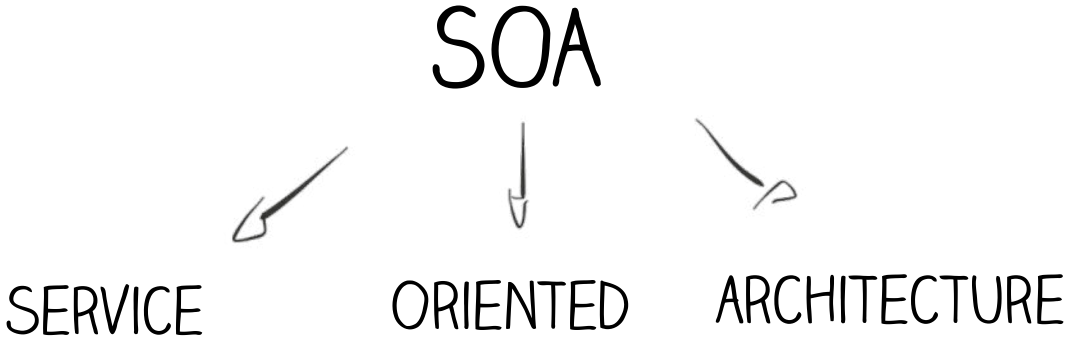
INTRODUCTION TO MICROSERVICES



SOA

SERVICE

ORIENTED

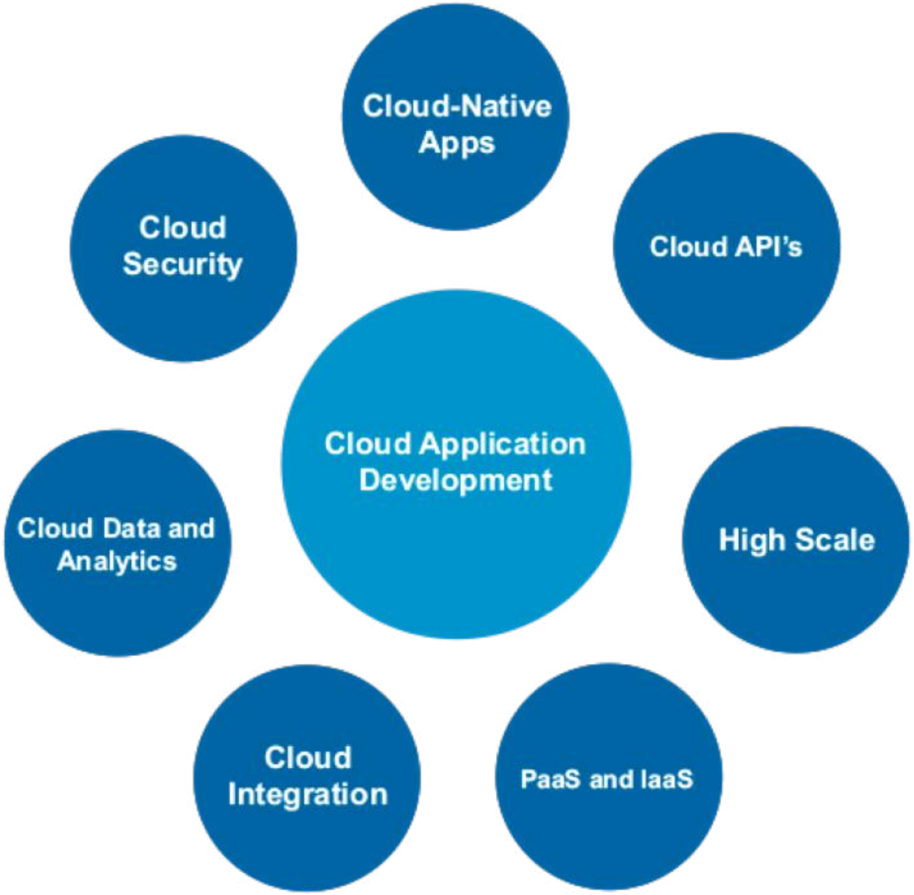
ARCHITECTURE

- MODERNIZED VERSION OF SOA

NEW WORLD:

- SPEED OF DELIVERY
- SCALABILITY
- INNOVATION / EXPERIMENTATION
- CLOUD / DEVOPS

Cloud Application Development



Cloud-Native Apps

Cloud Security

Cloud API's

Cloud Application Development

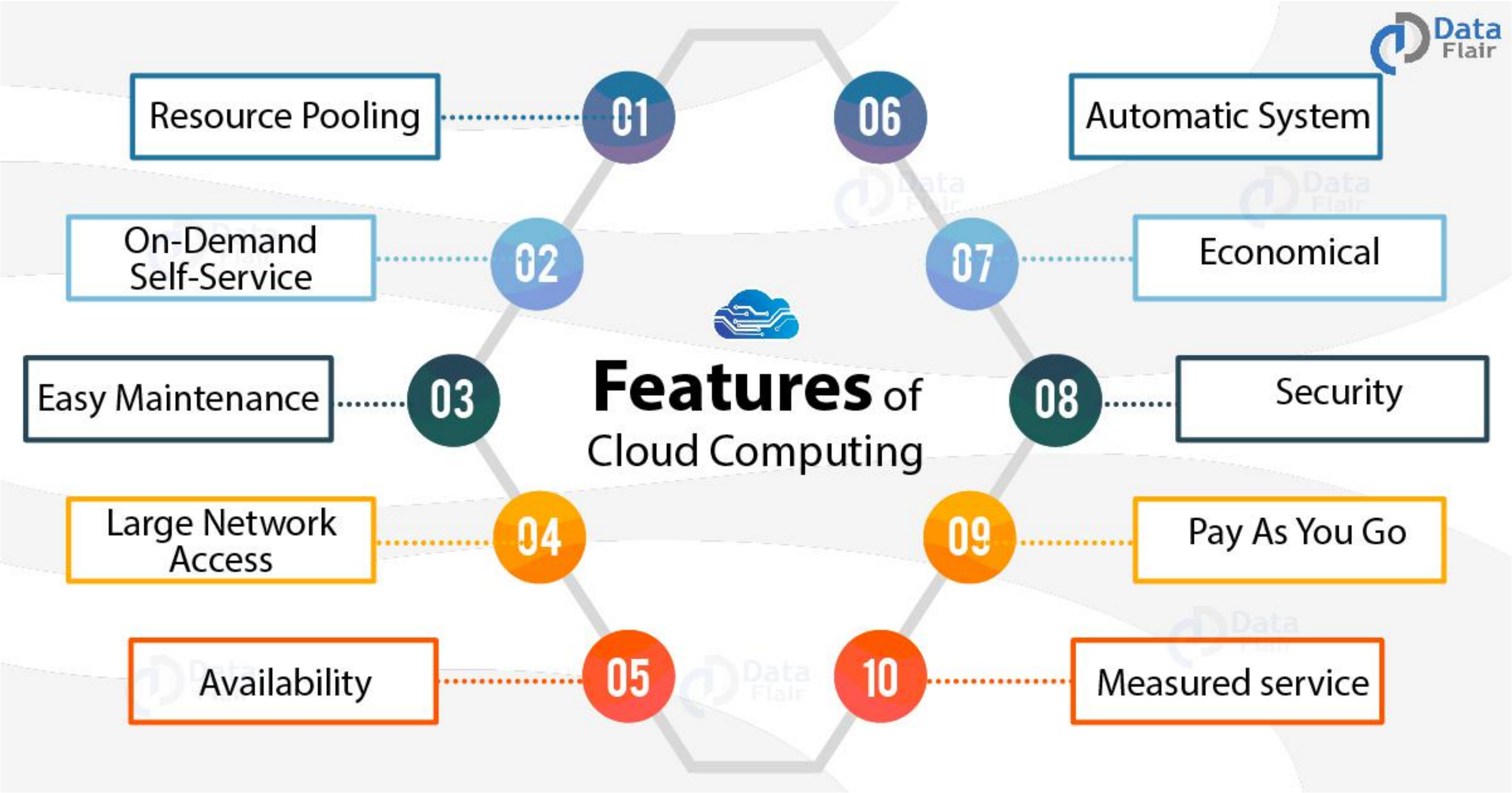
Cloud Data and Analytics

High Scale

Cloud Integration

PaaS and IaaS

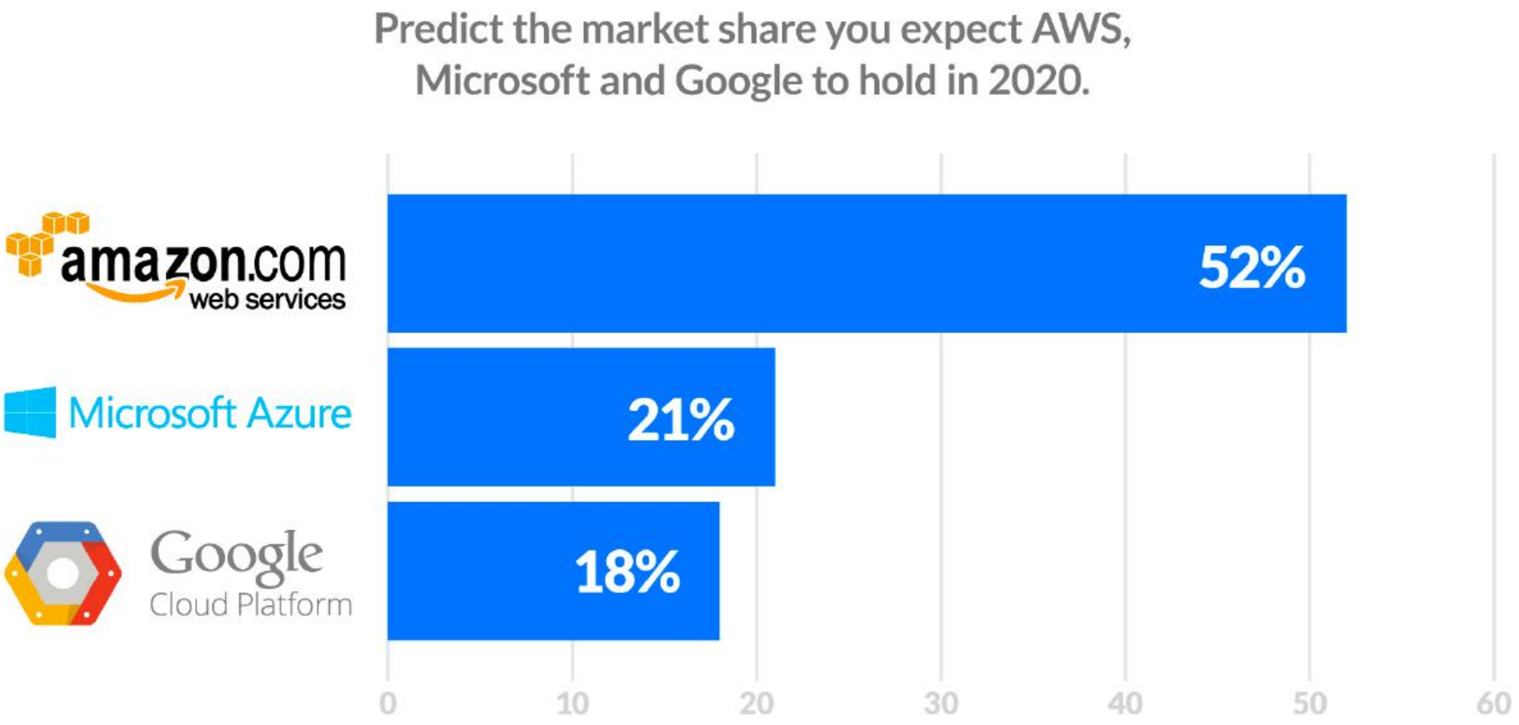
Cloud Characteristics



What is Cloud computing?

Cloud computing usecases

Cloud Computing Dynamics



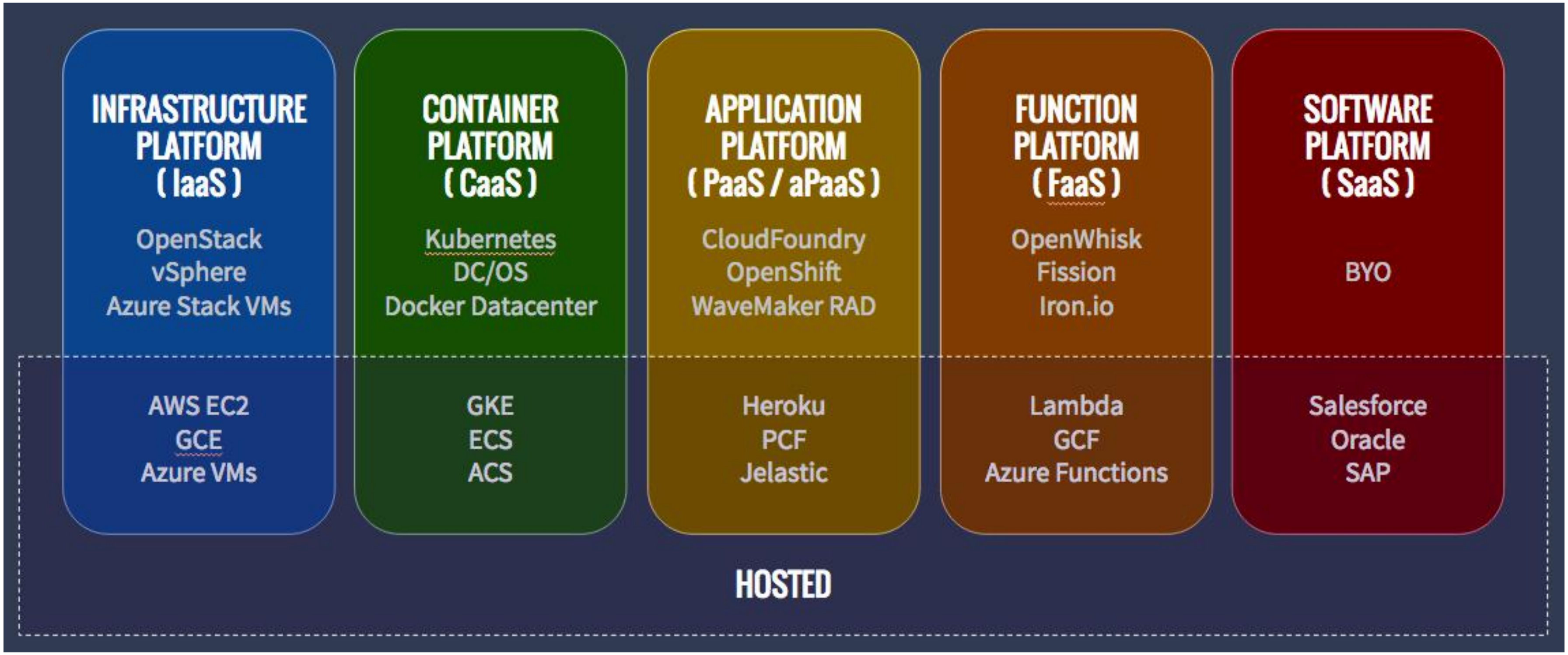
IBM Open Group - TOGAF

- *“The capability provided to the consumer is to use the provider’s applications running on a cloud infrastructure and accessible from various client devices through a thin client interface such as a Web browser (e.g., web-based email). The consumer does not manage or control the underlying cloud infrastructure, network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings”*
- *“Software-as-a-Service (SaaS) shares the distinction of being both a business model and an application delivery model. SaaS enables customers to utilize an application on a pay-as- you-go basis and eliminates the need to install and run the application on the customer’s own hardware”*

SaaS simplified..

- SaaS can be roughly defined as software deployed on cloud, provided from or through cloud and also being consumed by accessing these services through cloud. The cloud here refers to the Internet.
- SaaS through cloud has:
  - A new method to provide software, which has otherwise been provided in compact disk or memory stick.
  - A new business model for software providers on a ‘pay-per-use’ mode rather than one-time buying of license for each copy being used.
  - SaaS has begun to open a new market segment for software; otherwise is confined to use by large enterprises with heavy IT use

*aaS



INFRASTRUCTURE PLATFORM (IaaS)	CONTAINER PLATFORM (CaaS)	APPLICATION PLATFORM (PaaS / aPaaS)	FUNCTION PLATFORM (FaaS)	SOFTWARE PLATFORM (SaaS)
OpenStack vSphere Azure Stack VMs	Kubernetes DC/OS Docker Datacenter	CloudFoundry OpenShift WaveMaker RAD	OpenWhisk Fission Iron.io	BYO
AWS EC2 GCE Azure VMs	GKE ECS ACS	Heroku PCF Jelastic	Lambda GCF Azure Functions	Salesforce Oracle SAP

HOSTED

Details

- **Infrastructure as a Service (IaaS) :** It provides only a base infrastructure (Virtual machine, Software Define Network, Storage attached). End user have to configure and manage platform and environment, deploy applications on it.
  - **AWS (EC2), GCP (CE), Microsoft Azure (VM)**
- **Software as a Service (SaaS**) : It is sometimes called to as “on-demand software”. Typically accessed by users using a thin client via a web browser. In SaaS everything can be managed by vendors: applications, runtime, data, middleware, OSes, virtualization, servers, storage and networking, End users have to use it.
  - **GMAIL** is Best example of SaaS.

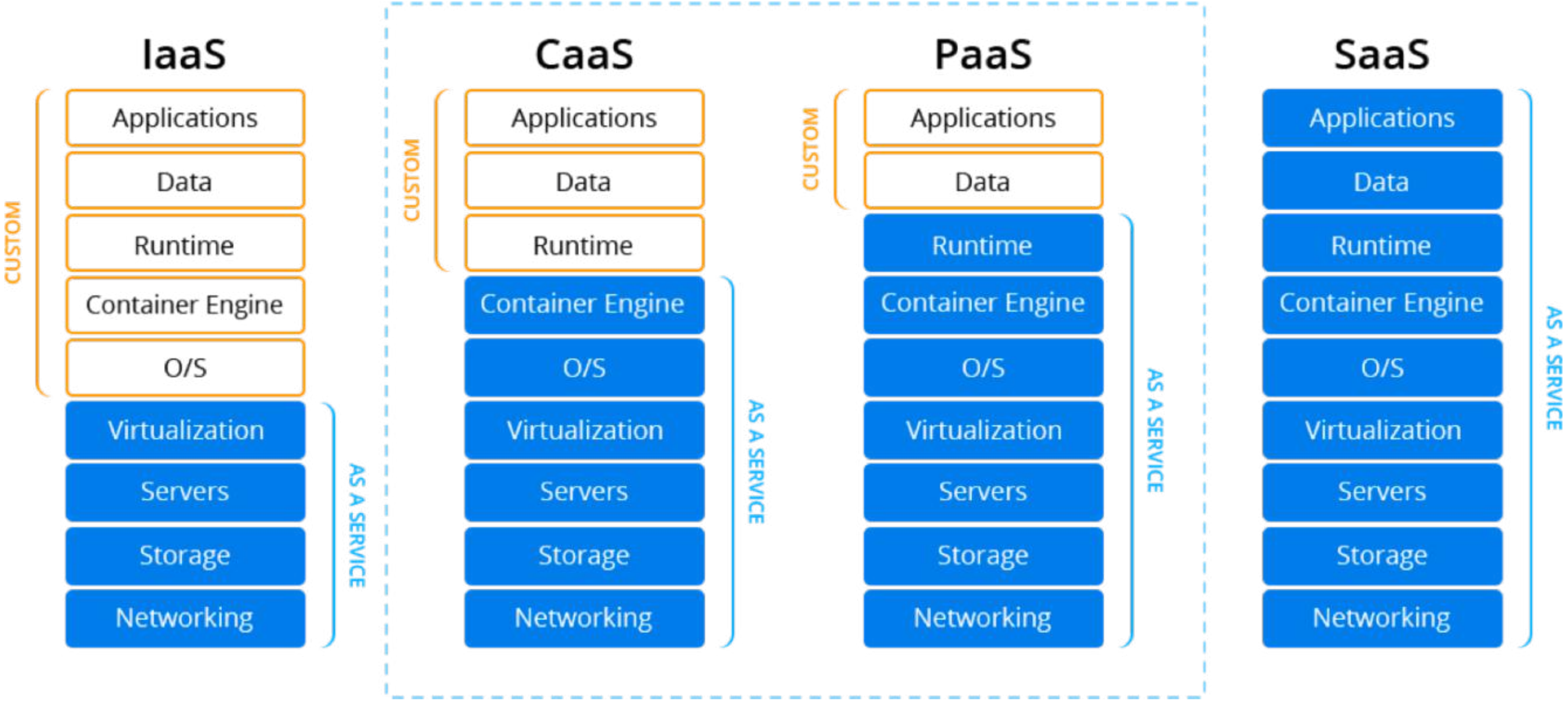
Details

- **Platform as a Service (PaaS):** It provides a platform allowing end user to develop, run, and manage applications without the complexity of building and maintaining the infrastructure.
  - **Google App Engine, CloudFoundry, Heroku, AWS (Beanstalk)**
- **Container as a Service (CaaS):** Is a form of container-based virtualization in which container engines, orchestration and the underlying compute resources are delivered to users as a service from a cloud provider.
  - **Google Container Engine(GKE), AWS (ECS), Azure (ACS)** and **Pivotal (PKS)**

Details

- **Function as a Service (FaaS):** It provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure.
    - **AWS (Lamda), Google Cloud Function** and **Azure Functions**

Let’s dig in



IaaS

CUSTOM: Applications, Data, Runtime, Container Engine, O/S

AS A SERVICE: Virtualization, Servers, Storage, Networking

CaaS

CUSTOM: Applications, Data, Runtime

AS A SERVICE: Container Engine, O/S, Virtualization, Servers, Storage, Networking

PaaS

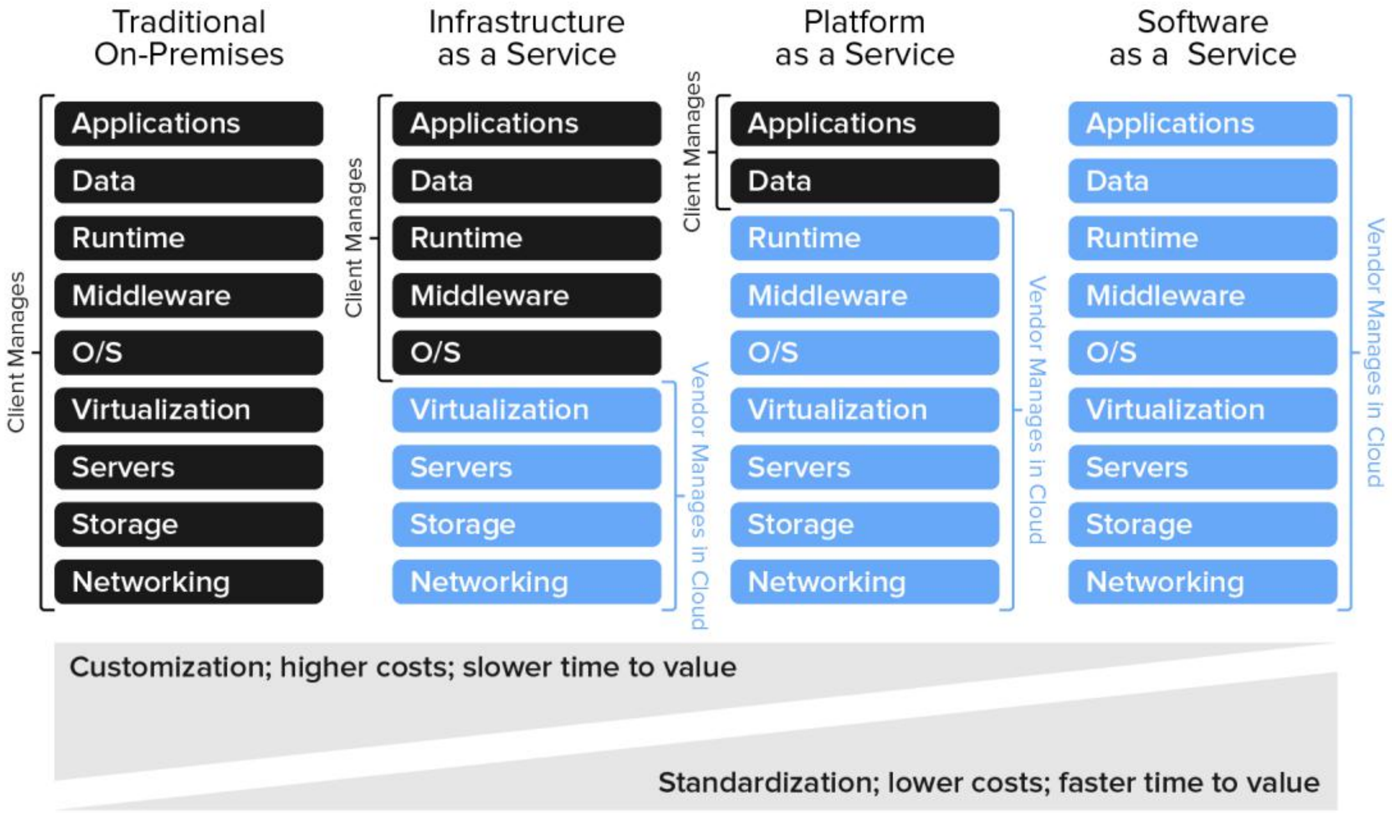
CUSTOM: Applications, Data

AS A SERVICE: Runtime, Container Engine, O/S, Virtualization, Servers, Storage, Networking

SaaS

AS A SERVICE: Applications, Data, Runtime, Container Engine, O/S, Virtualization, Servers, Storage, Networking

On-Premises to SaaS



Traditional On-Premises	Infrastructure as a Service	Platform as a Service	Software as a Service
Applications	Applications	Applications	Applications
Data	Data	Data	Data
Runtime	Runtime	Runtime	Runtime
Middleware	Middleware	Middleware	Middleware
O/S	O/S	O/S	O/S
Virtualization	Virtualization	Virtualization	Virtualization
Servers	Servers	Servers	Servers
Storage	Storage	Storage	Storage
Networking	Networking	Networking	Networking

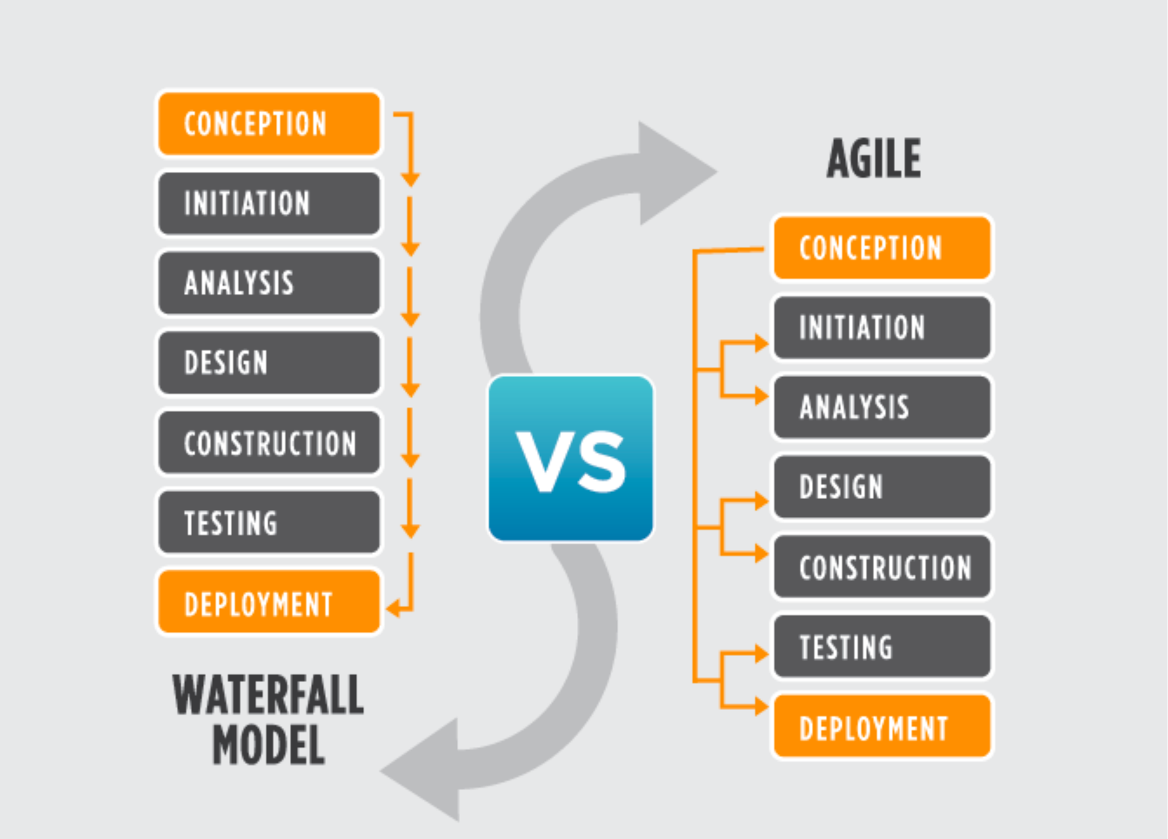
Customization; higher costs; slower time to value

Standardization; lower costs; faster time to value

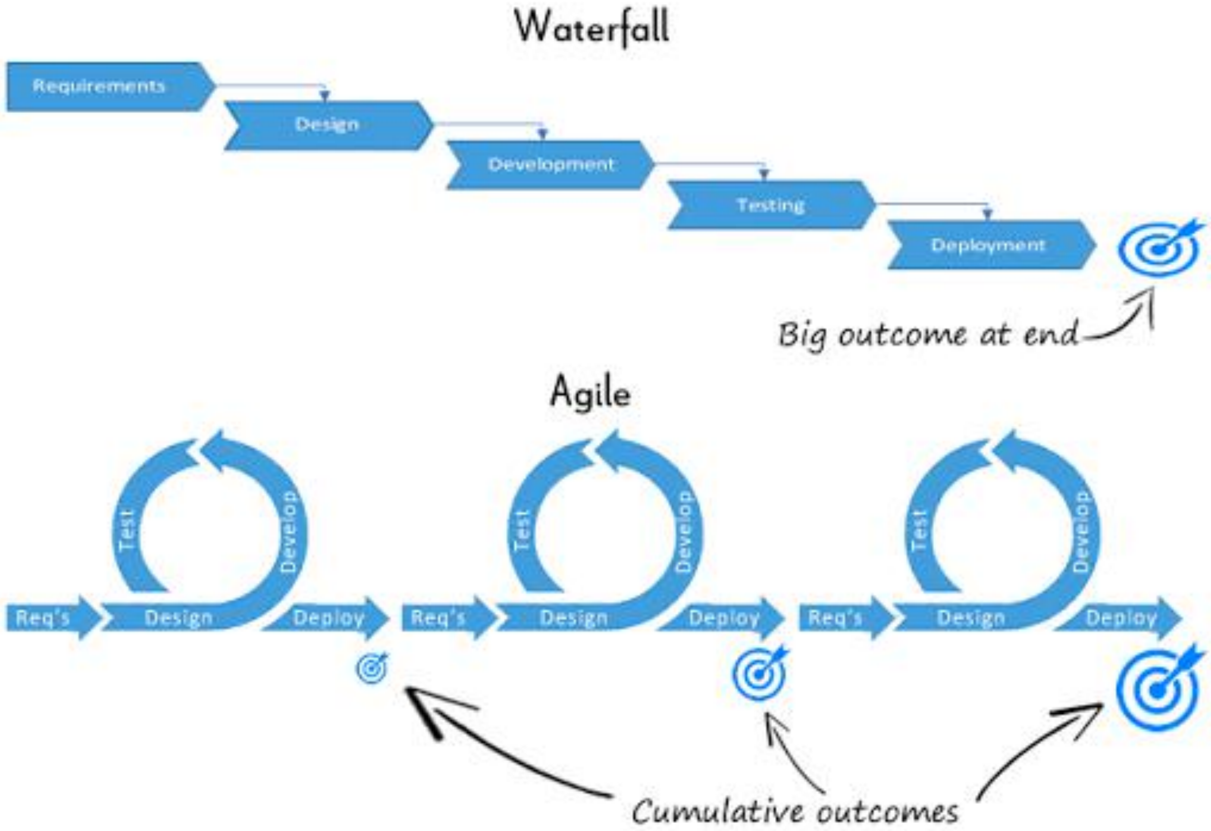
AGILE Methodology

AGILE methodology is a practice that promotes continuous iteration of development and testing throughout the software development lifecycle of the project. Both development and testing activities are concurrent unlike the Waterfall model.

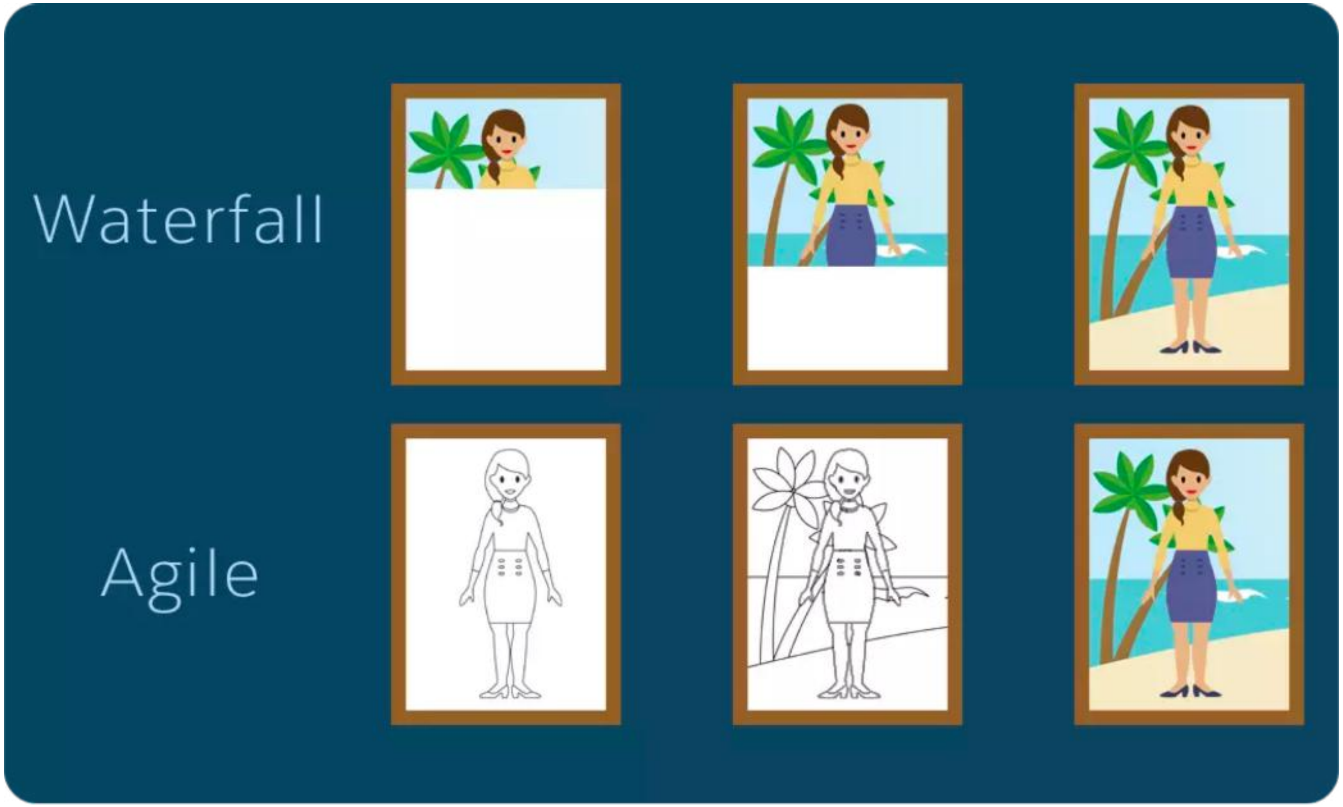
Waterfall vs Agile



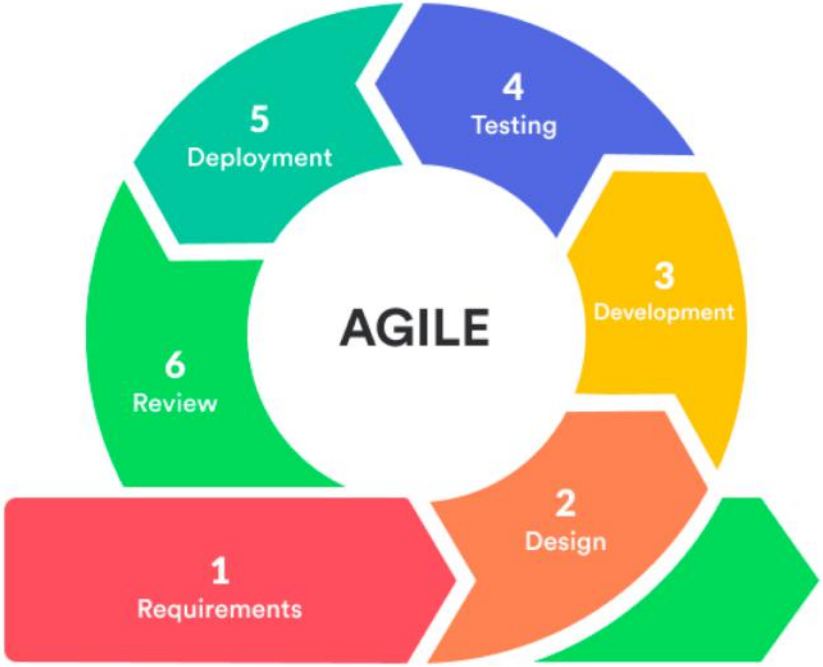
Waterfall vs Agile

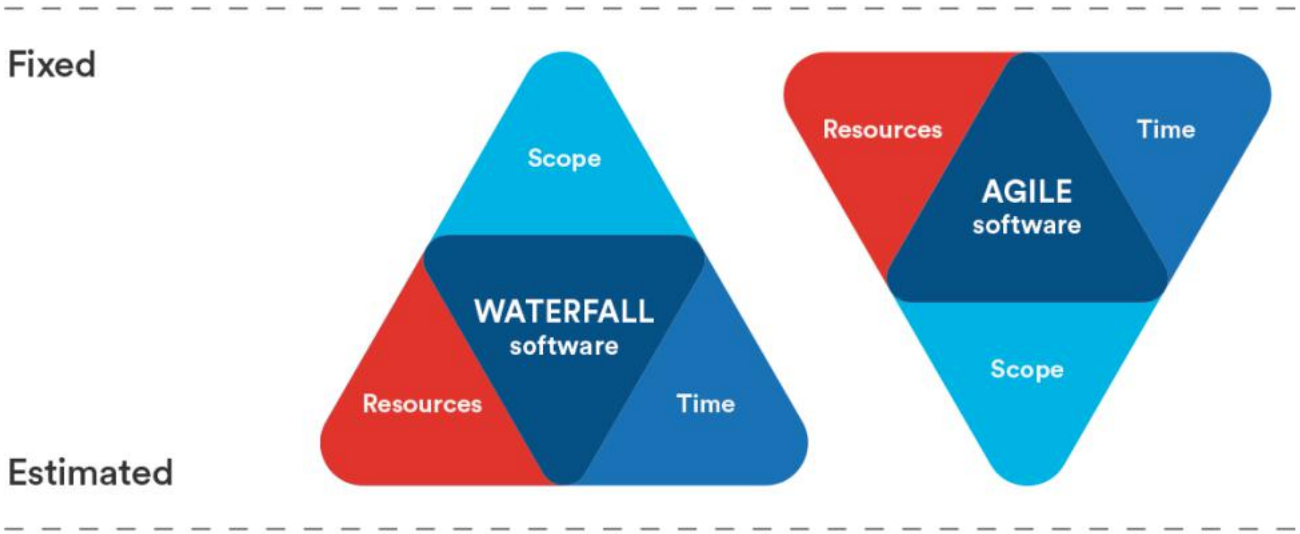


Better view



Agile


AGILE
1
Requirements
2
Design
3
Development
4
Testing
5
Deployment
6
Review



Fixed

Estimated

Scope

WATERFALL
software

Resources

Time

Resources

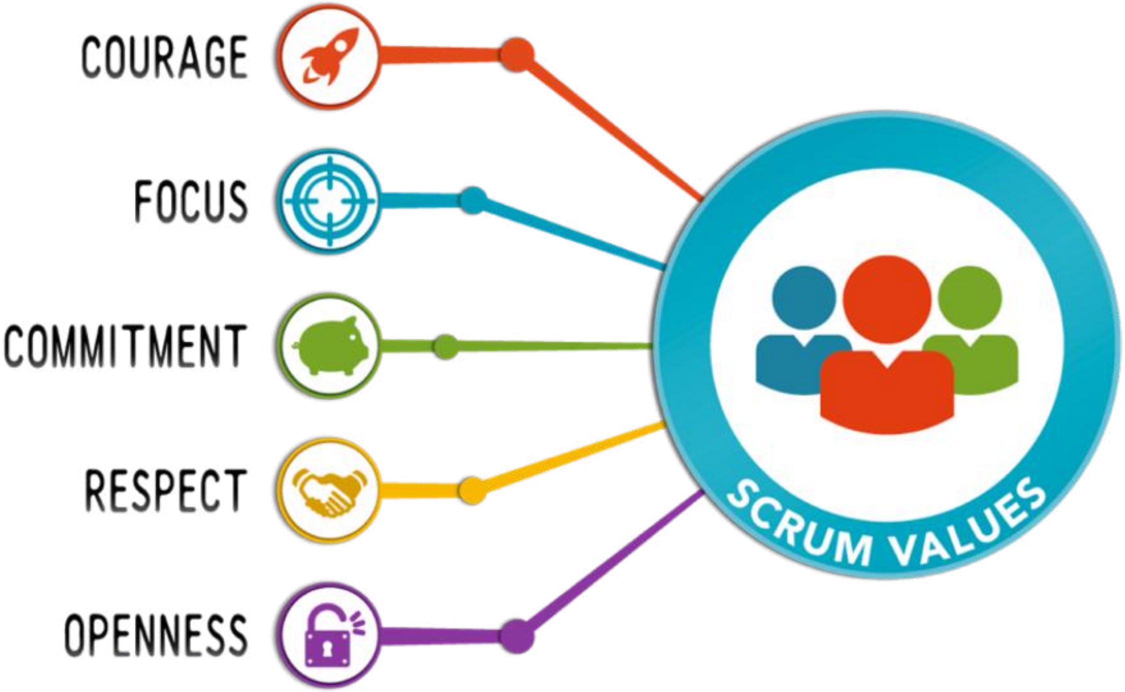
Time

AGILE
software

Scope

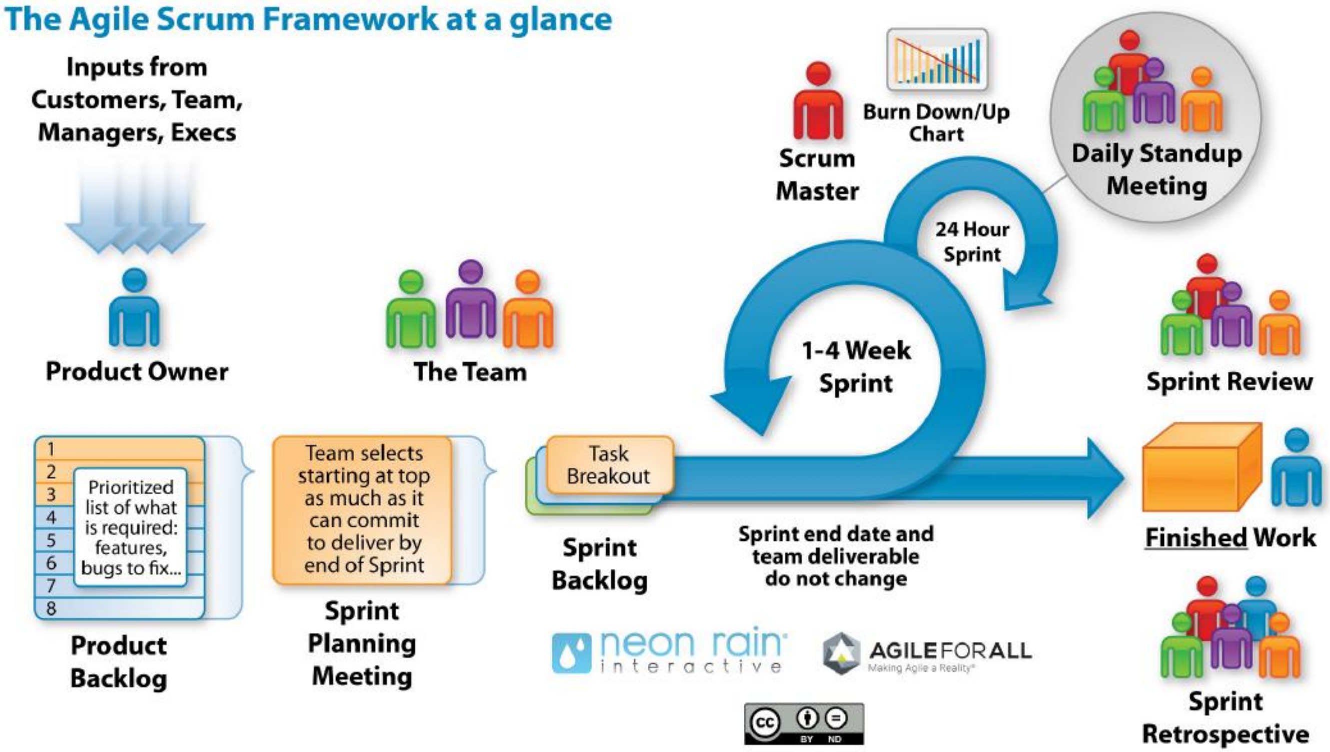
SCRUM

OPENNESS
COURAGE
RESPECT
FOCUS
COMMITMENT

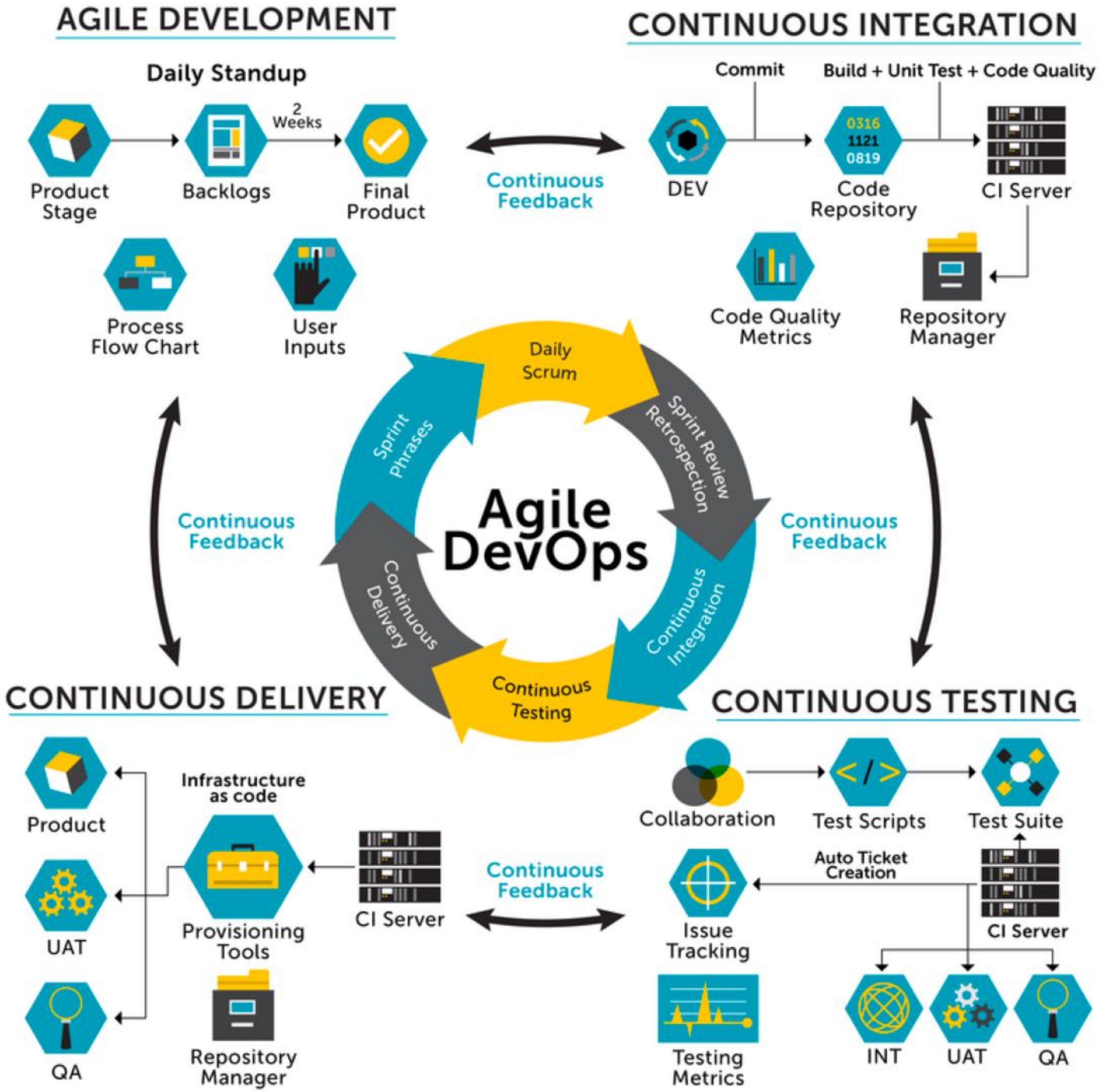

COURAGE
FOCUS
COMMITMENT
RESPECT
OPENNESS
SCRUM VALUES

Agile

AGILE Methodology



DEVOPS


AGILE DEVELOPMENT
Daily Standup
Product Stage
Backlogs
2 Weeks
Final Product
Process Flow Chart
User Inputs
Continuous Feedback
CONTINUOUS INTEGRATION
Commit
Build + Unit Test + Code Quality
DEV
0316
1121
0819
Code Repository
CI Server
Code Quality Metrics
Repository Manager
Continuous Feedback
Daily Scrum
Sprint Review Retrospection
Continuous Integration
Continuous Testing
Continuous Delivery
Sprint Phrases
Agile DevOps
Continuous Feedback
CONTINUOUS DELIVERY
Product
UAT
QA
Infrastructure as code
Provisioning Tools
Repository Manager
CI Server
Continuous Feedback
CONTINUOUS TESTING
Collaboration
Test Scripts
Test Suite
Auto Ticket Creation
Issue Tracking
CI Server
Testing Metrics
INT
UAT
QA

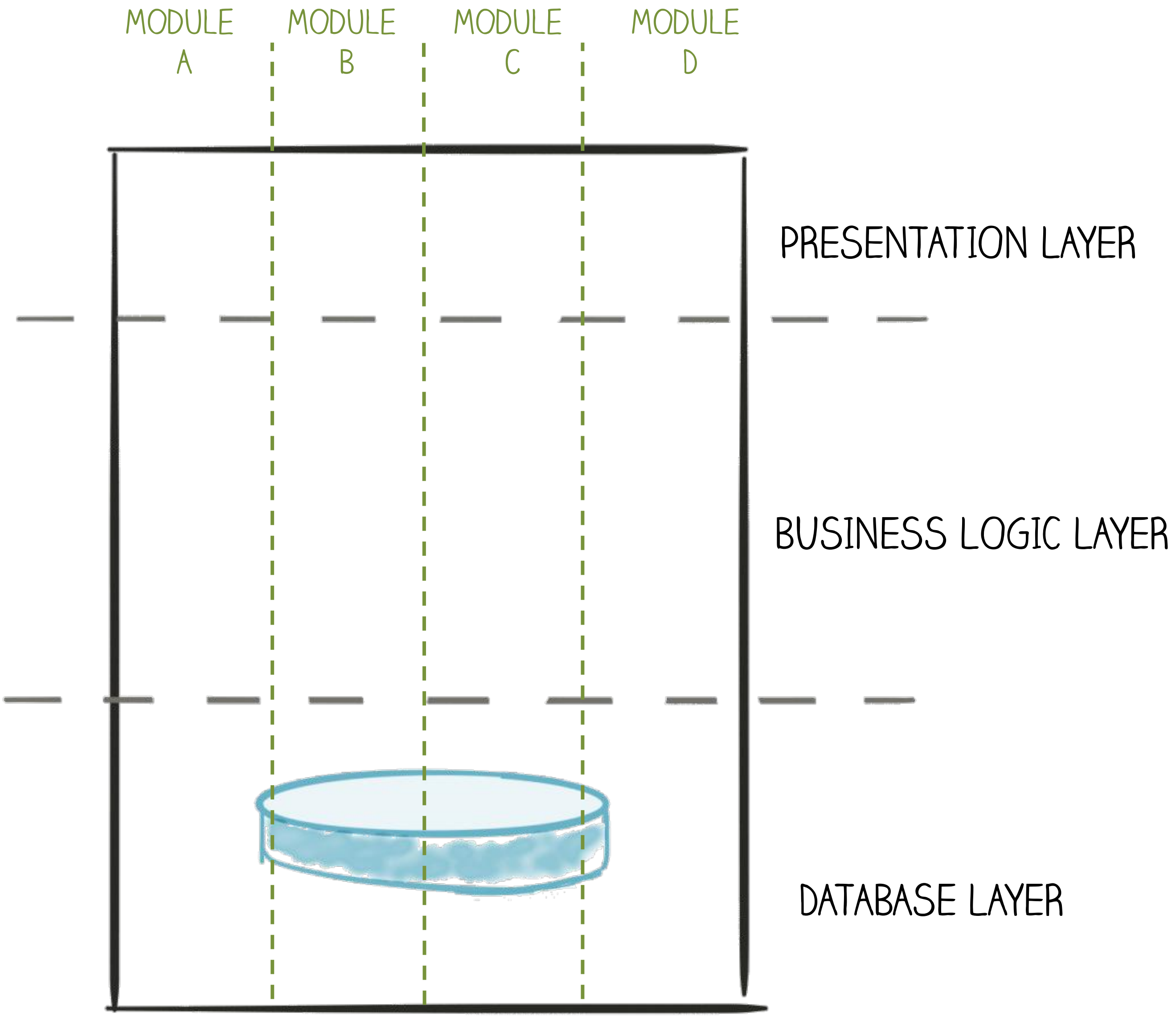
INTRODUCTION TO MICROSERVICES

VS

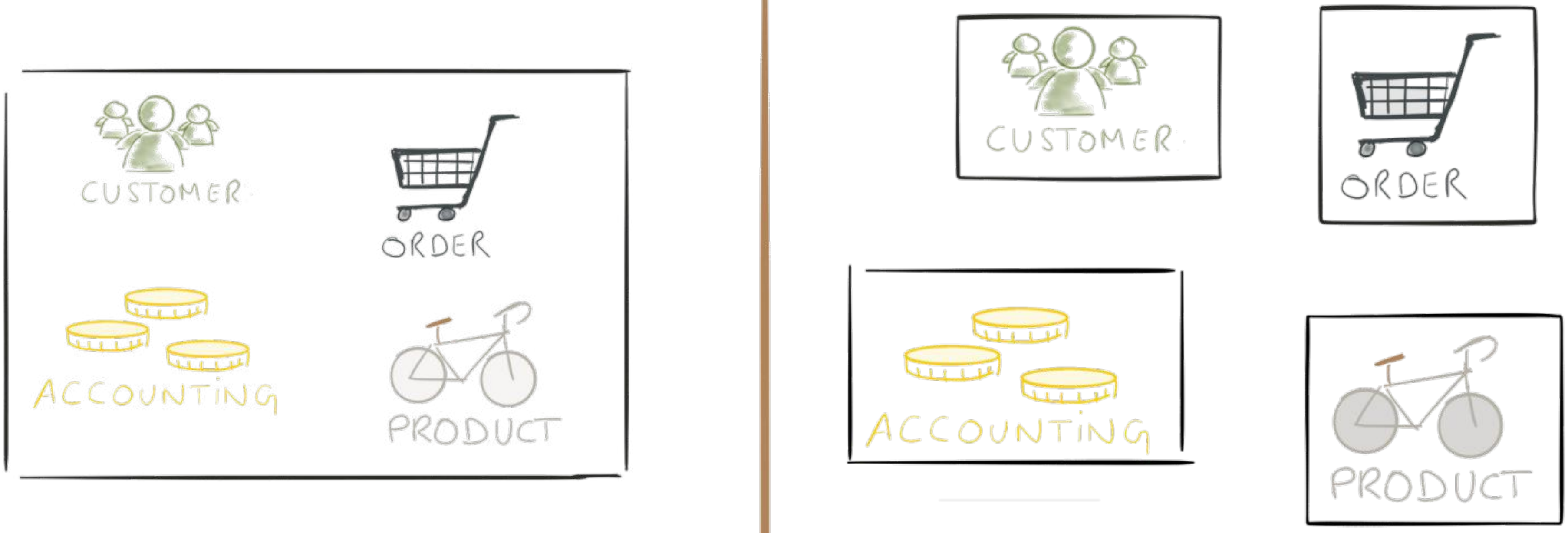
MONOLITH

MICROSERVICES

A MONOLITH


MODULE A
MODULE B
MODULE C
MODULE D
PRESENTATION LAYER
BUSINESS LOGIC LAYER
DATABASE LAYER

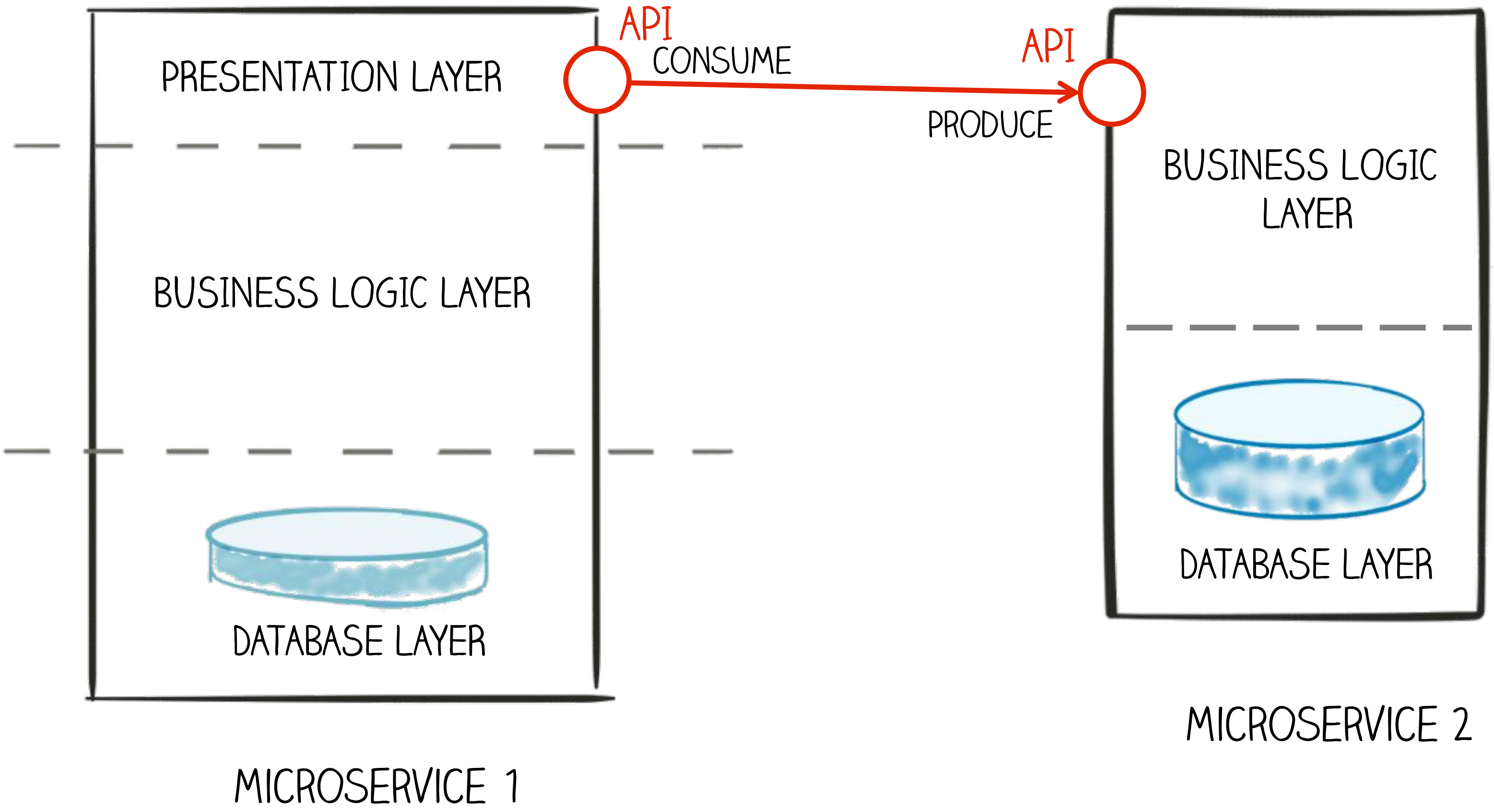
VS



MONOLITH

MICROSERVICES

MICROSERVICES



PRESENTATION LAYER

BUSINESS LOGIC LAYER

DATABASE LAYER

MICROSERVICE 1

API

CONSUME

API

PRODUCE

BUSINESS LOGIC LAYER

DATABASE LAYER

MICROSERVICE 2

PRINCIPLES

□ MODULARITY

□ AUTONOMOUS

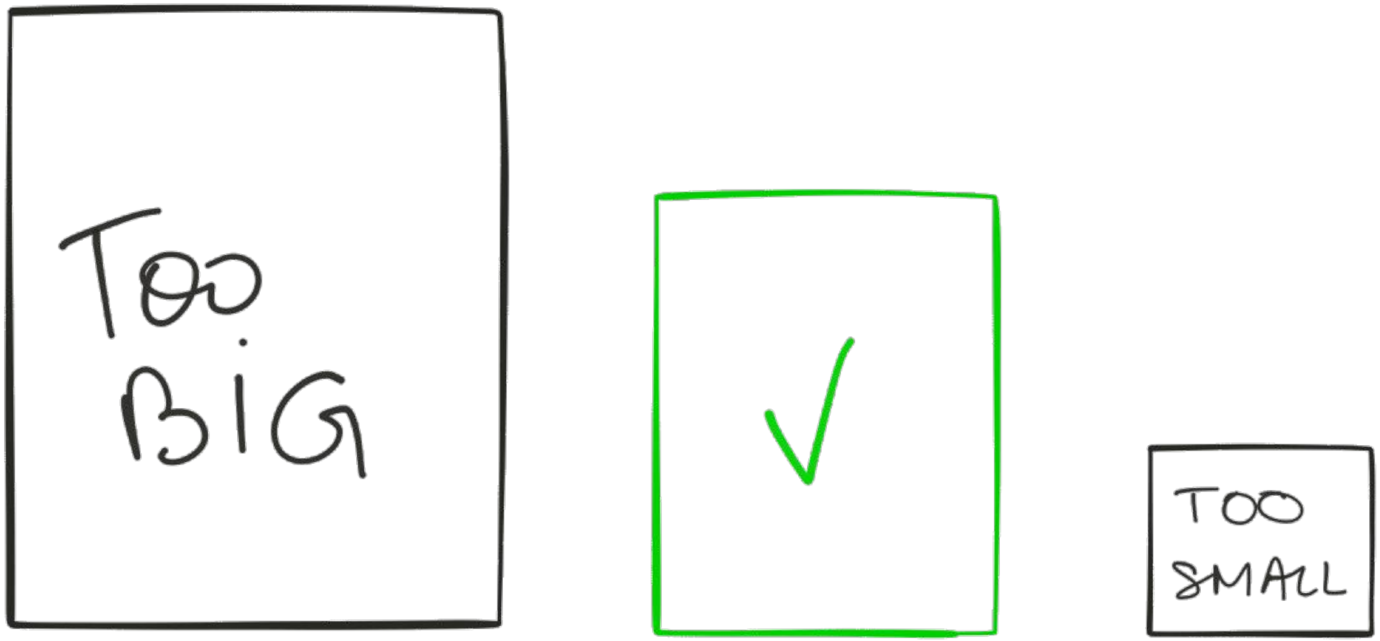
□ HIDE IMPLEMENTATION DETAILS

□ AUTOMATION

□ STATELESS

□ HIGHLY OBSERVABLE

MODULARITY



MODULARITY

▨ □ □ □ □ □

✓ MODELLED AROUND BUSINESS CAPABILITY

- SINGLE RESPONSIBILITY
- SINGLE DATA DOMAIN

✓ SEPARATION OF CONCERNS

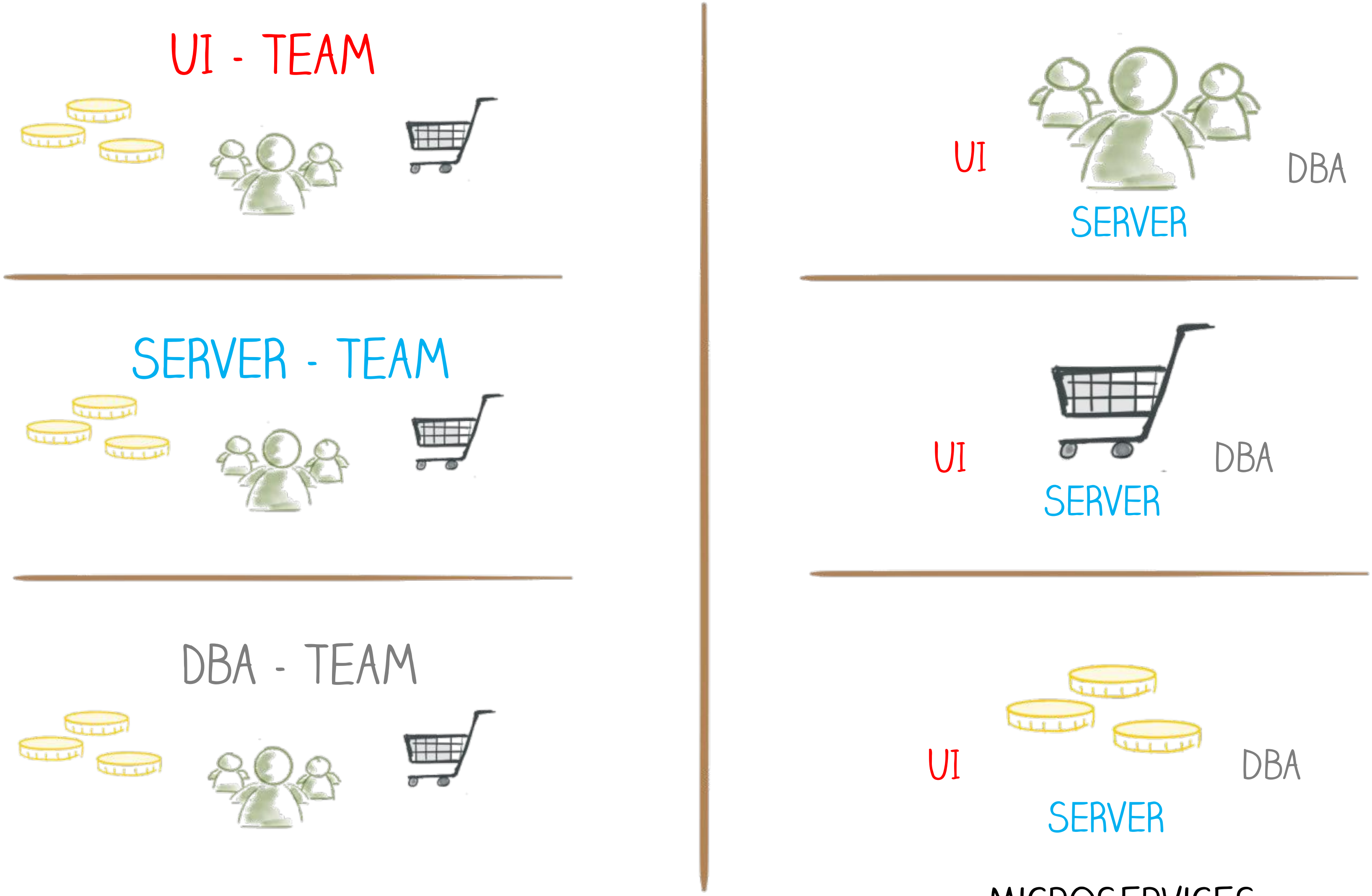
✓ LOW COUPLING

✓ UNDERSTANDABLE BY A PERSON

MODULARITY (TEAM)

▨ □ □ □ □ □

A PRODUCT NOT A PROJECT



UI - TEAM

SERVER - TEAM

DBA - TEAM

UI

SERVER

DBA

UI

SERVER

DBA

UI

SERVER

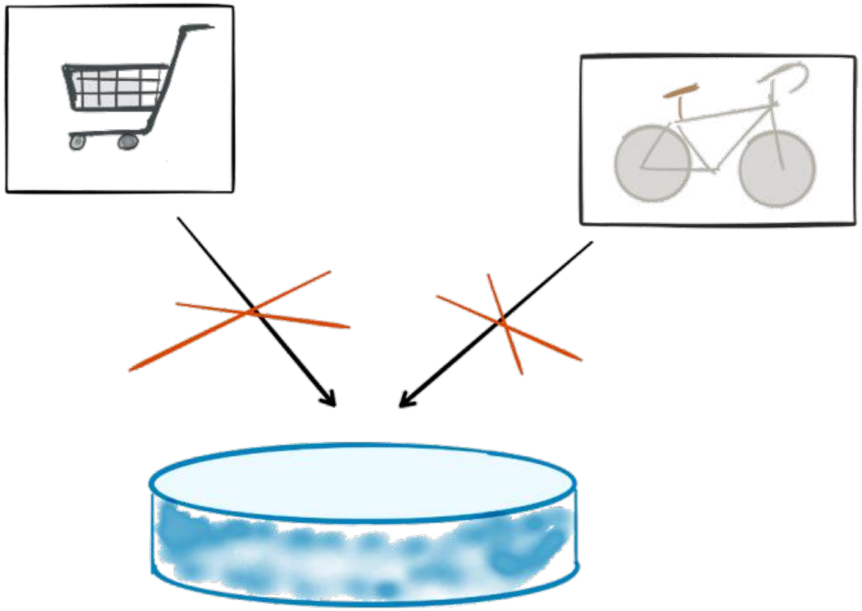
DBA

MONOLITH

MICROSERVICES

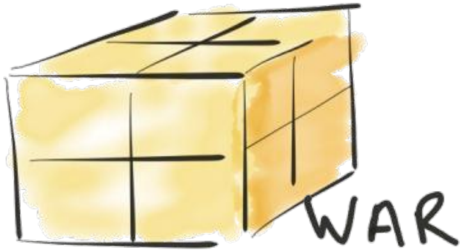
AUTONOMOUS

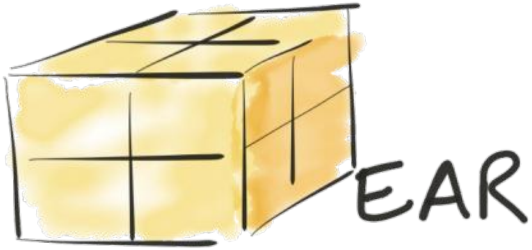
▨ ▨ □ □ □ □



AUTONOMOUS

▨ ▨ □ □ □ □

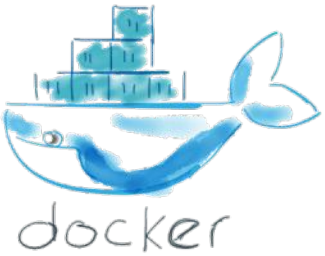




✓

- LIBRARIES
- HTTP LISTENER

✓

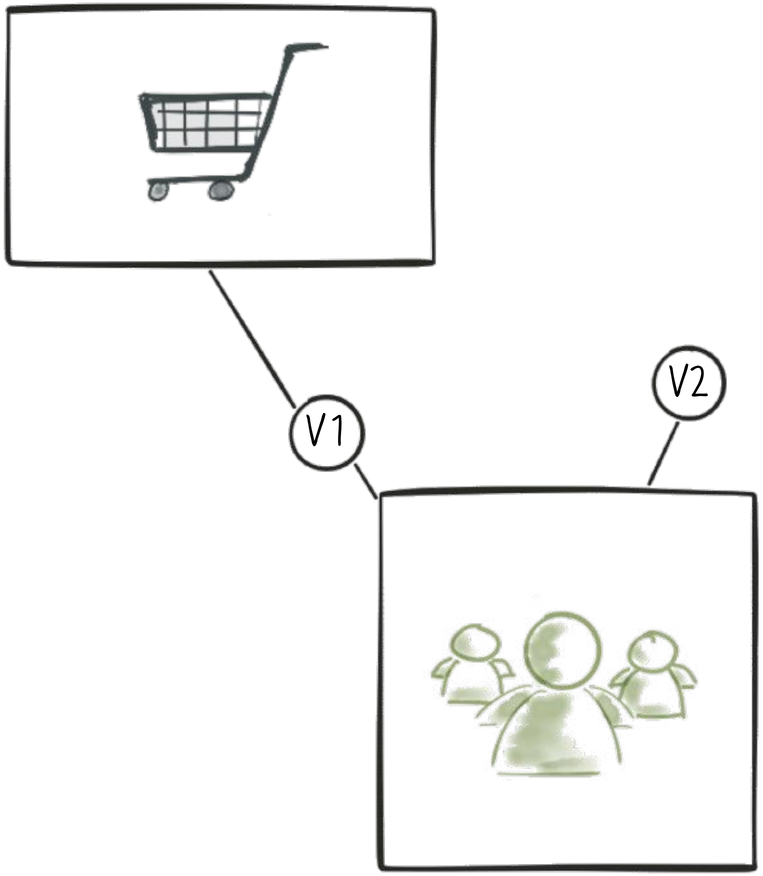


MONOLITH

MICROSERVICES

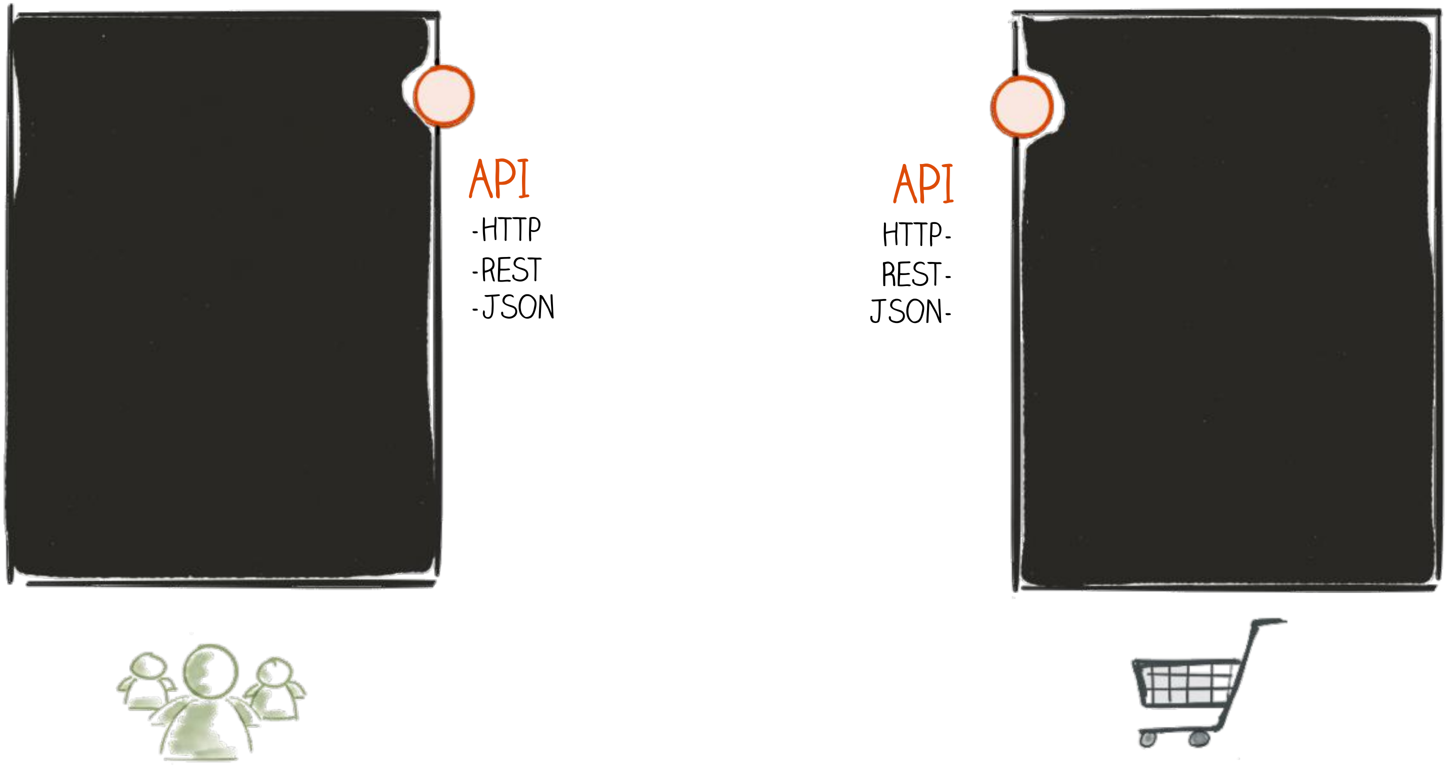
AUTONOMOUS

▨ ▨ □ □ □ □



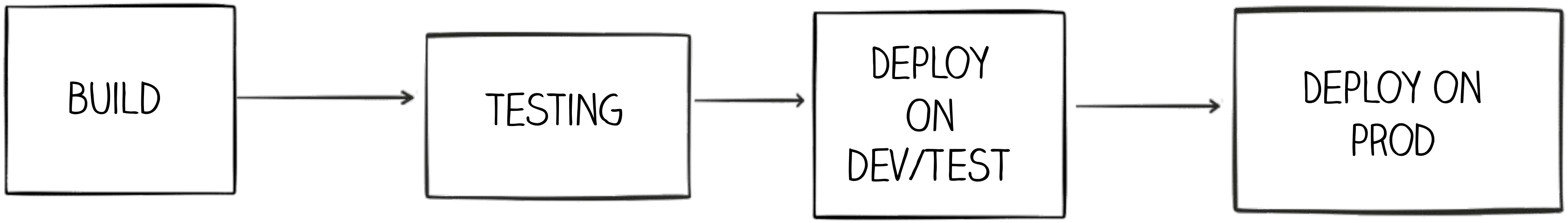
HIDE IMPLEMENTATION DETAILS

▨ ▨ ▨ □ □ □



AUTOMATION

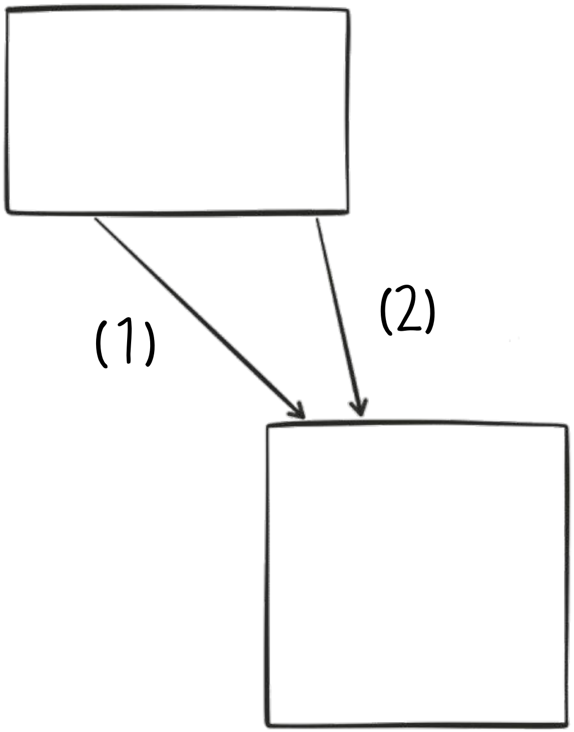
▨ ▨ ▨ ▨ □ □



- CONTINUOUS INTEGRATION
- CONTINUOUS DEPLOYMENT

STATELESS

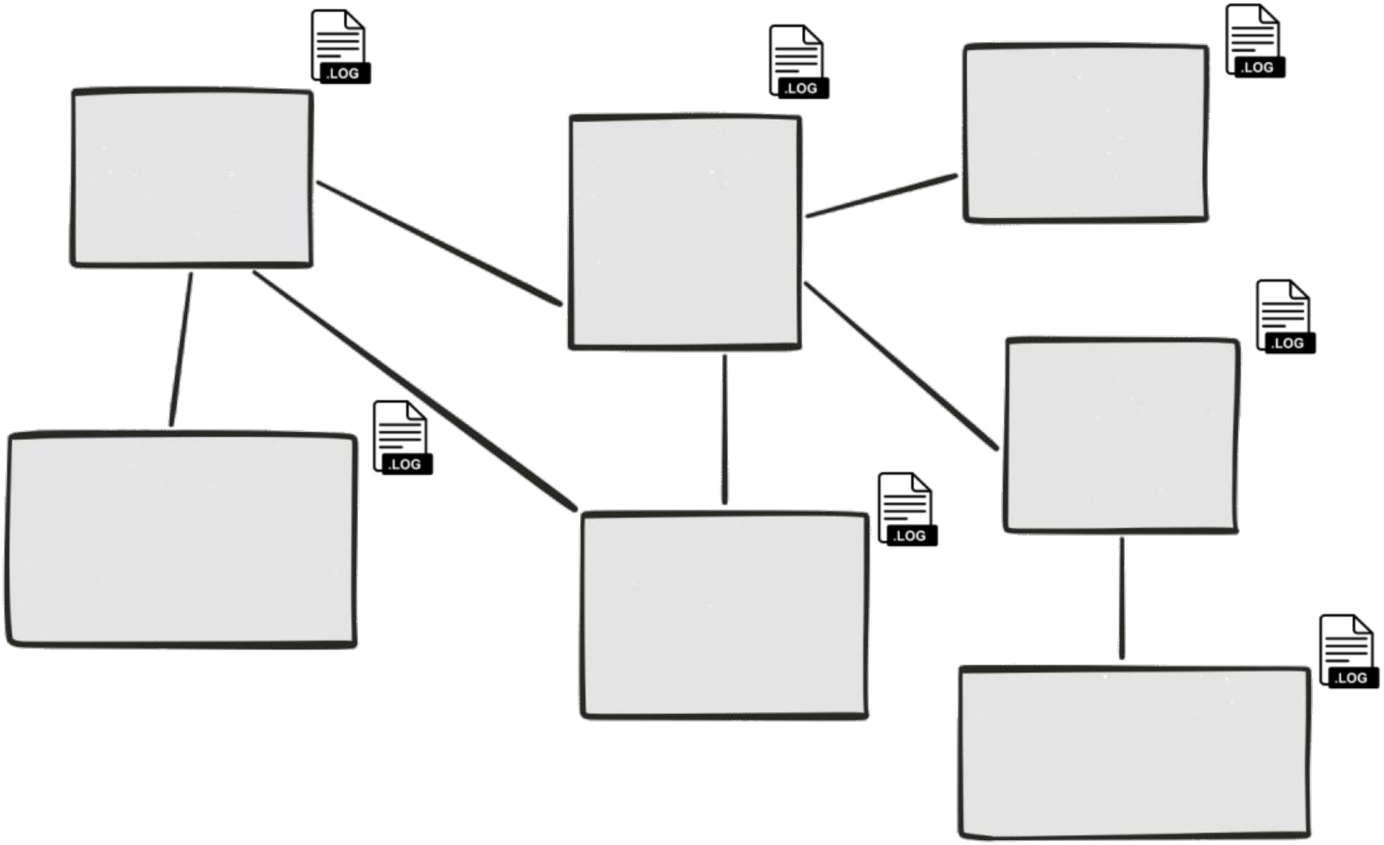
▨ ▨ ▨ ▨ ▨ □



HIGHLY OBSERVABLE

▨ ▨ ▨ ▨ ▨ ▨

LOGS



HIGHLY OBSERVABLE

▨ ▨ ▨ ▨ ▨ ▨

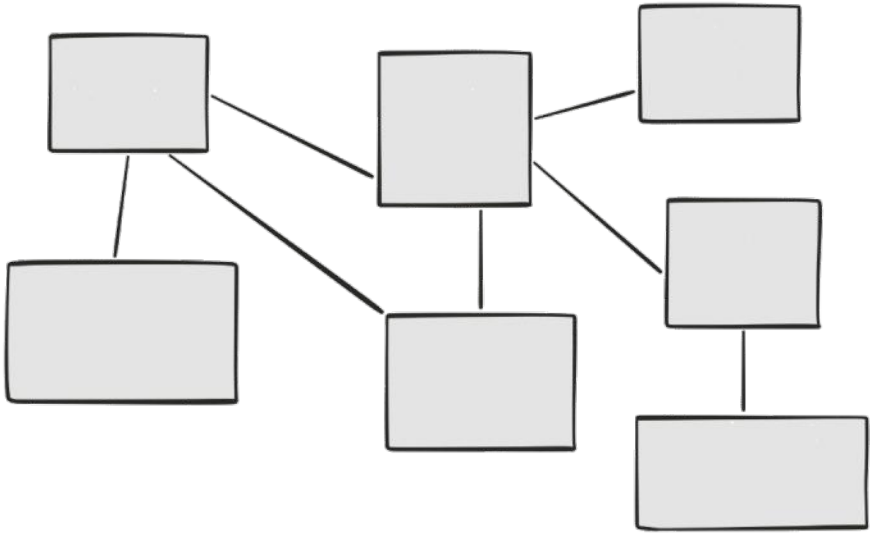
CENTRALIZED LOGGING

.LOG .LOG .LOG .LOG

elasticsearch.

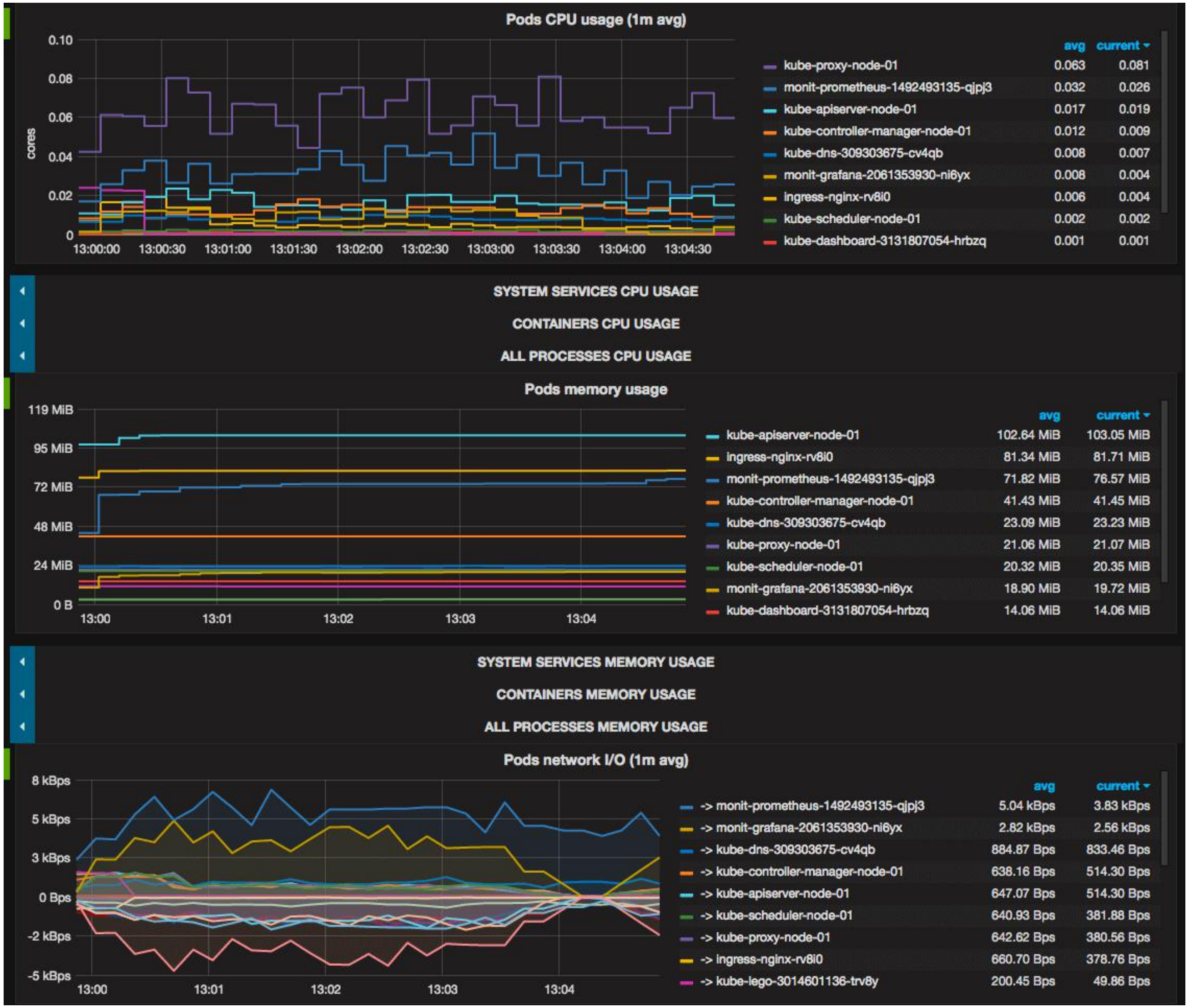
logstash

Kibana



HIGHLY OBSERVABLE

MONITORING



Pods CPU usage (1m avg)

	avg	current ▾
kube-proxy-node-01	0.063	0.081
monit-prometheus-1492493135-qjpj3	0.032	0.026
kube-apiserver-node-01	0.017	0.019
kube-controller-manager-node-01	0.012	0.009
kube-dns-309303675-cv4qb	0.008	0.007
monit-grafana-2061353930-ni6yx	0.008	0.004
ingress-nginx-rv8i0	0.006	0.004
kube-scheduler-node-01	0.002	0.002
kube-dashboard-3131807054-hrbzq	0.001	0.001

SYSTEM SERVICES CPU USAGE

CONTAINERS CPU USAGE

ALL PROCESSES CPU USAGE

Pods memory usage

	avg	current ▾
kube-apiserver-node-01	102.64 MiB	103.05 MiB
ingress-nginx-rv8i0	81.34 MiB	81.71 MiB
monit-prometheus-1492493135-qjpj3	71.82 MiB	76.57 MiB
kube-controller-manager-node-01	41.43 MiB	41.45 MiB
kube-dns-309303675-cv4qb	23.09 MiB	23.23 MiB
kube-proxy-node-01	21.06 MiB	21.07 MiB
kube-scheduler-node-01	20.32 MiB	20.35 MiB
monit-grafana-2061353930-ni6yx	18.90 MiB	19.72 MiB
kube-dashboard-3131807054-hrbzq	14.06 MiB	14.06 MiB

SYSTEM SERVICES MEMORY USAGE

CONTAINERS MEMORY USAGE

ALL PROCESSES MEMORY USAGE

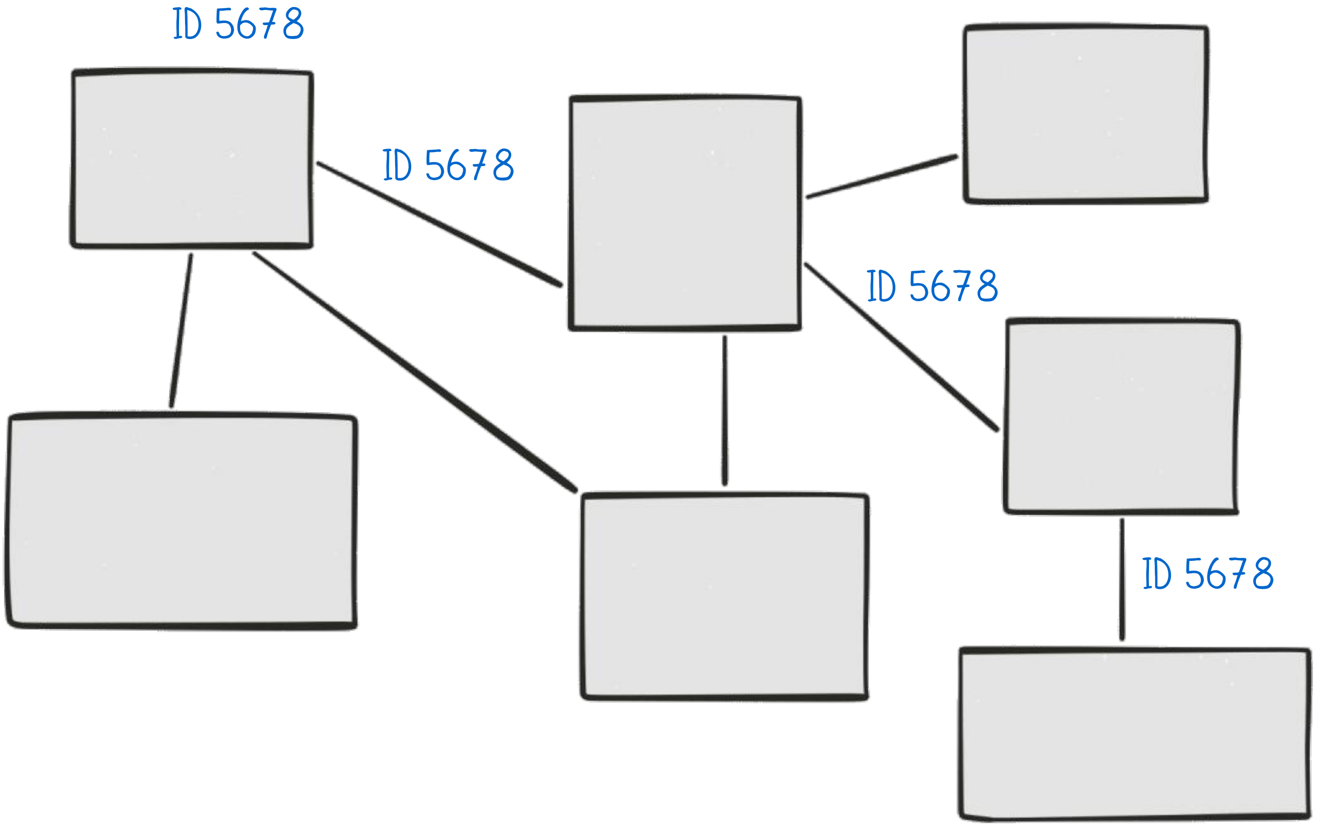
Pods network I/O (1m avg)

	avg	current ▾
-> monit-prometheus-1492493135-qjpj3	5.04 kBps	3.83 kBps
-> monit-grafana-2061353930-ni6yx	2.82 kBps	2.56 kBps
-> kube-dns-309303675-cv4qb	884.87 Bps	833.46 Bps
-> kube-controller-manager-node-01	638.16 Bps	514.30 Bps
-> kube-apiserver-node-01	647.07 Bps	514.30 Bps
-> kube-scheduler-node-01	640.93 Bps	381.88 Bps
-> kube-proxy-node-01	642.62 Bps	380.56 Bps
-> ingress-nginx-rv8i0	660.70 Bps	378.76 Bps
-> kube-lego-3014601136-trv8y	200.45 Bps	49.86 Bps

HIGHLY OBSERVABLE

▨ ▨ ▨ ▨ ▨ ▨

CORRELATION IDS



PRINCIPLES

▨ MODULARITY

▨ AUTONOMOUS

▨ HIDE IMPLEMENTATION DETAILS

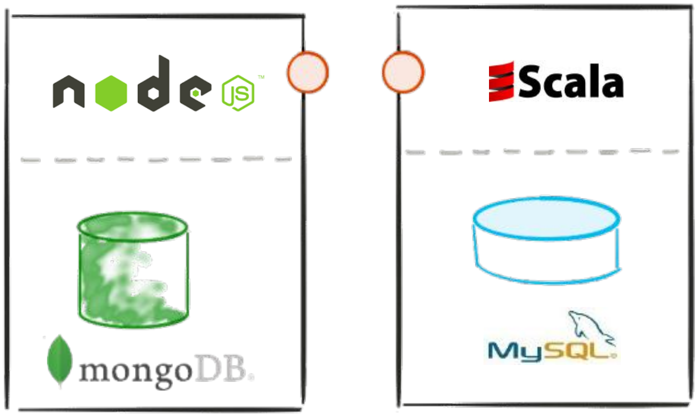
▨ AUTOMATION

▨ STATELESS

▨ HIGHLY OBSERVABLE

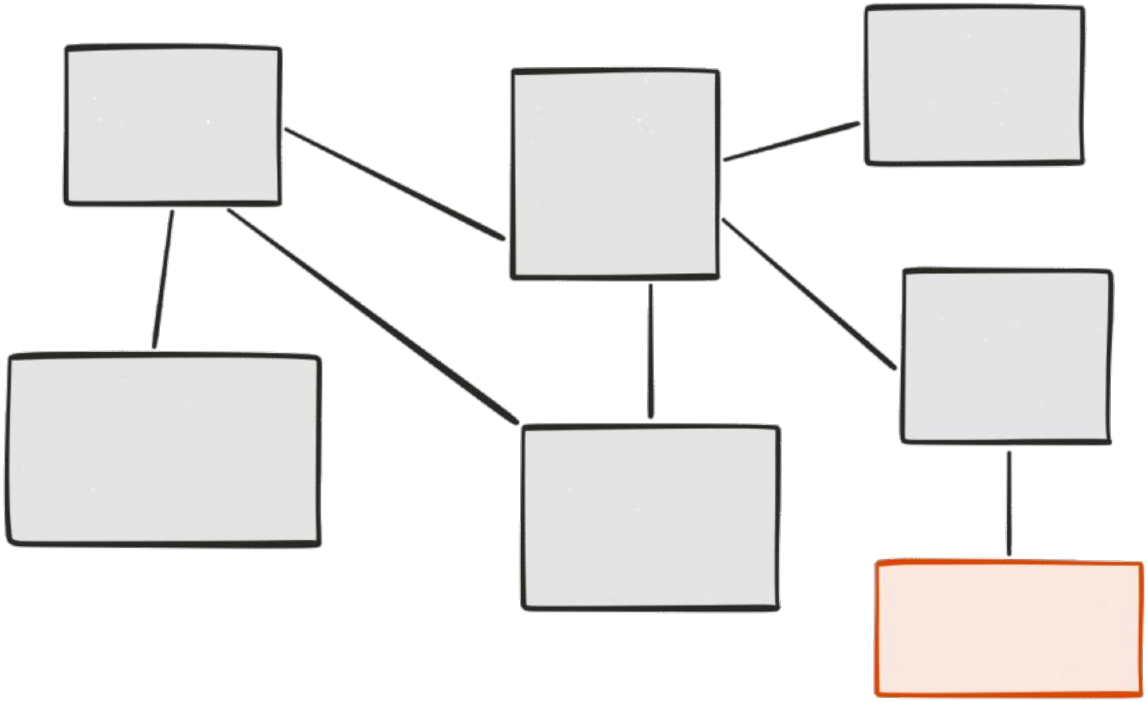
ADVANTAGES

POLYGLOT ARCHITECTURE



- THE RIGHT TECHNOLOGY FOR THE JOB
- REDUCE TECHNICAL DEBT

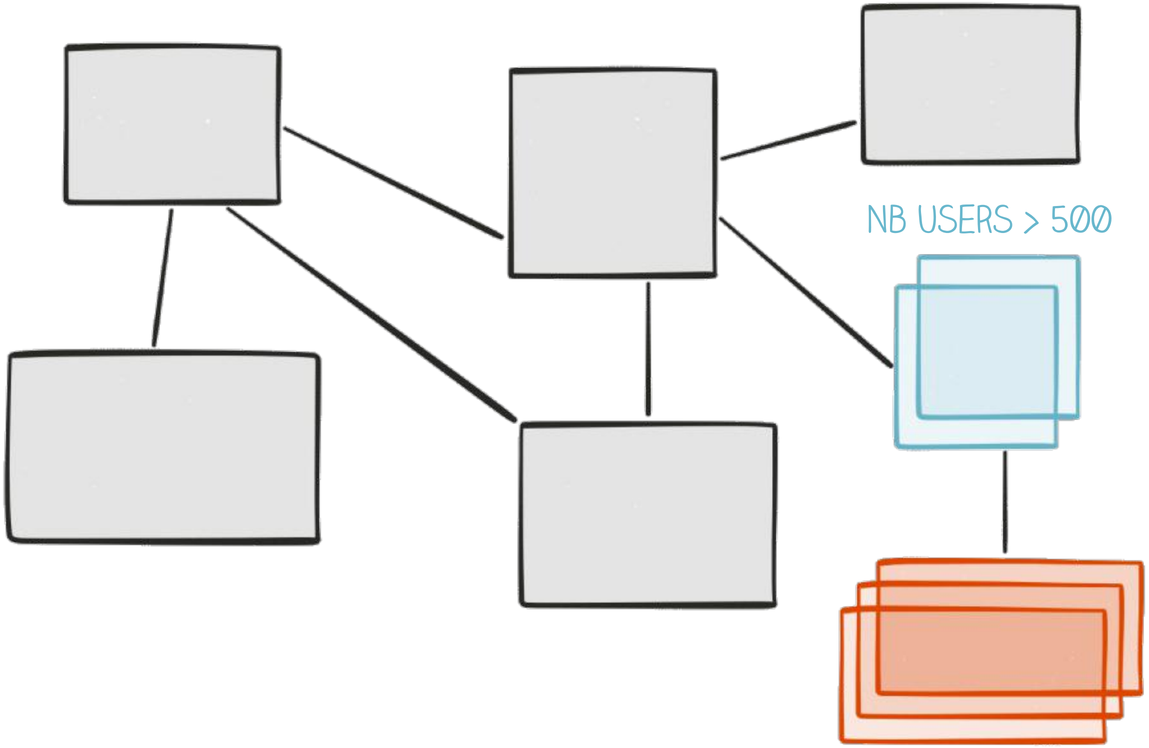
EVOLUTIONARY DESIGN



- REMOVE
- ADD
- REPLACE
- EXPERIMENTAL MICROSERVICE
- GROW AT "NO" COST

SELECTIVE SCALABILITY

▨ ▨ ▨



BIG VS SMALL

- ✓ SMALLER CODE BASE
- ✓ SIMPLER TO DEVELOP / TEST / DEPLOY / SCALE
- ✓ START FASTER
- ✓ EASIER FOR NEW DEVELOPERS

DRAWBACKS

- DISTRIBUTED SYSTEM
    - CONSISTENCY
    - TRANSACTION
    - REQUEST TRAVELLING
- SLOW (HTTP)
- REQUIRES AN ECOSYSTEM
- SYNCHRONOUS VS ASYNCHRONOUS
- INTEGRATION TESTS

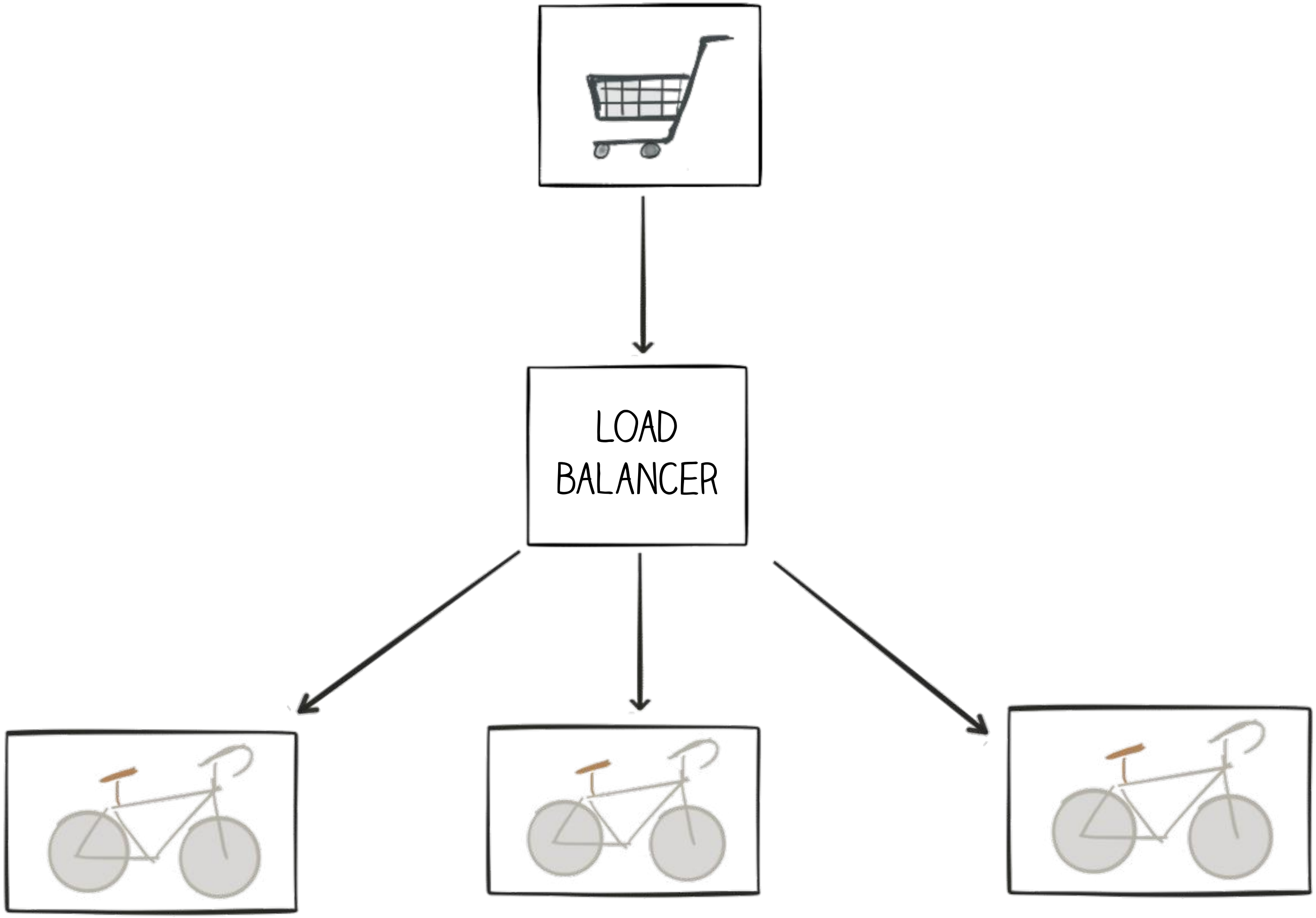
CONCLUSION:

- THE MICROSERVICES ARCHITECTURE IS MORE COMPLEX THAN A MONOLITH.
- AT THE BENEFIT OF GROWING AND SCALING EASILY

MICROSERVICES ECOSYSTEM

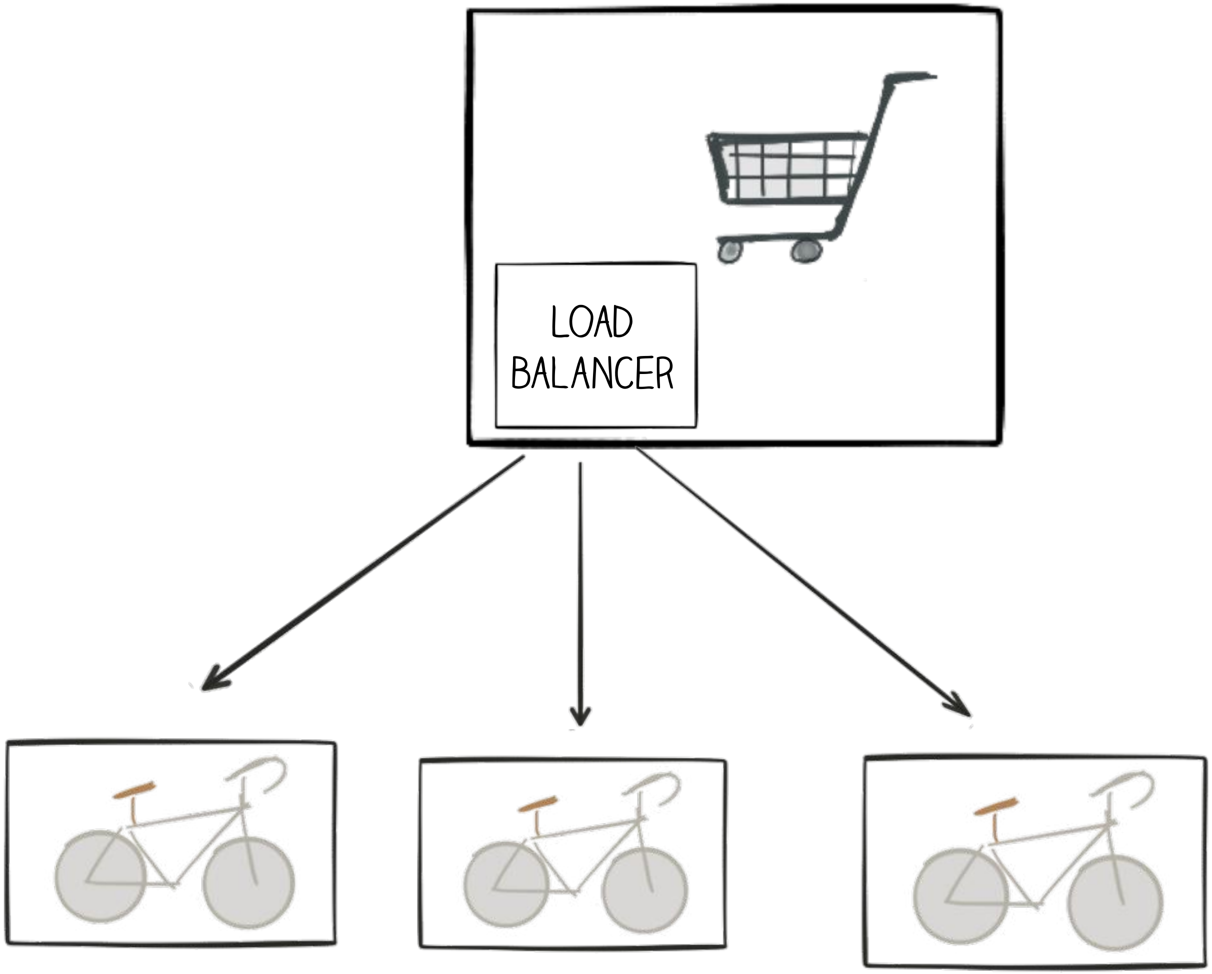
LOAD BALANCER

▨

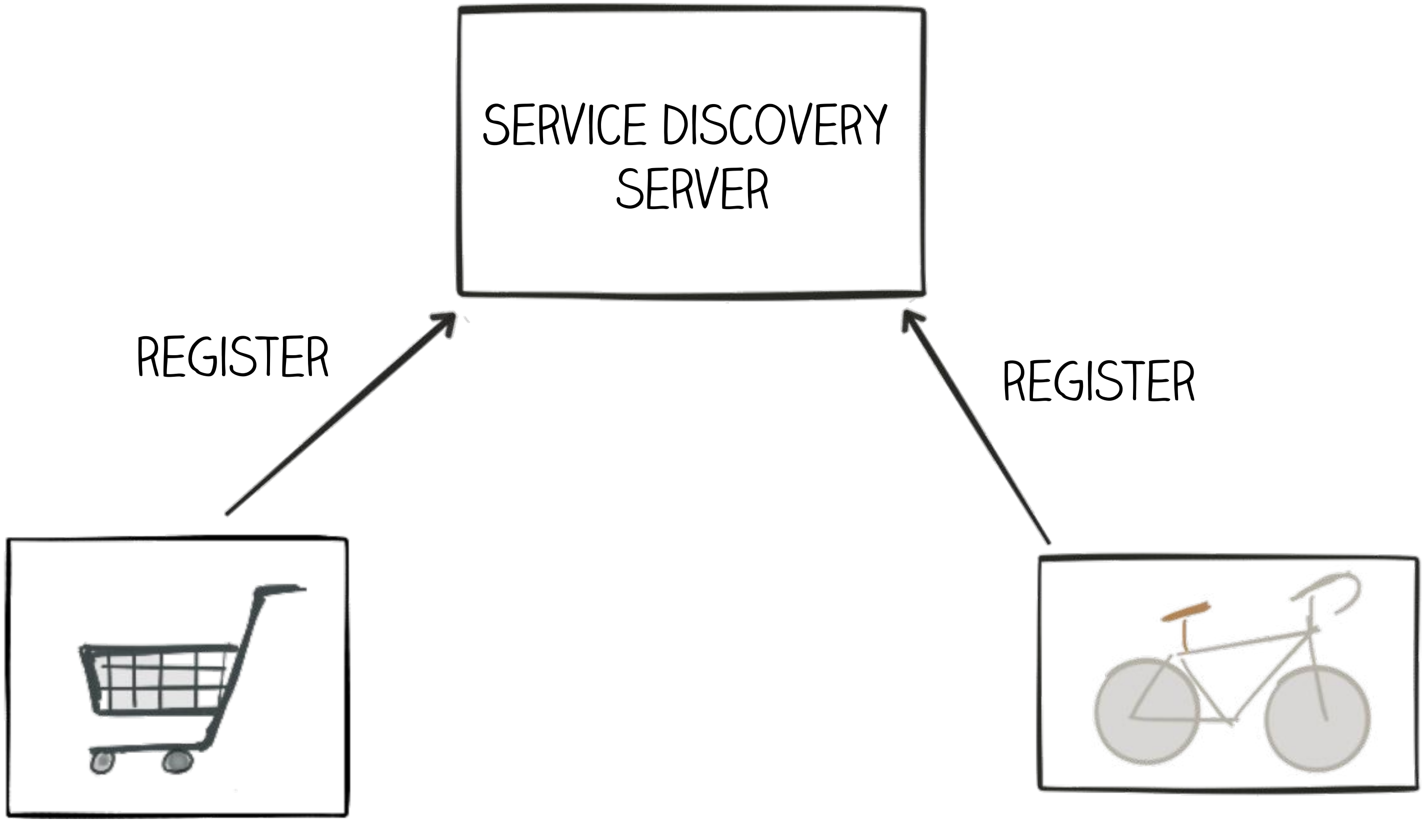


LOAD BALANCER (CLIENT SIDE)

▨



SERVICE DISCOVERY

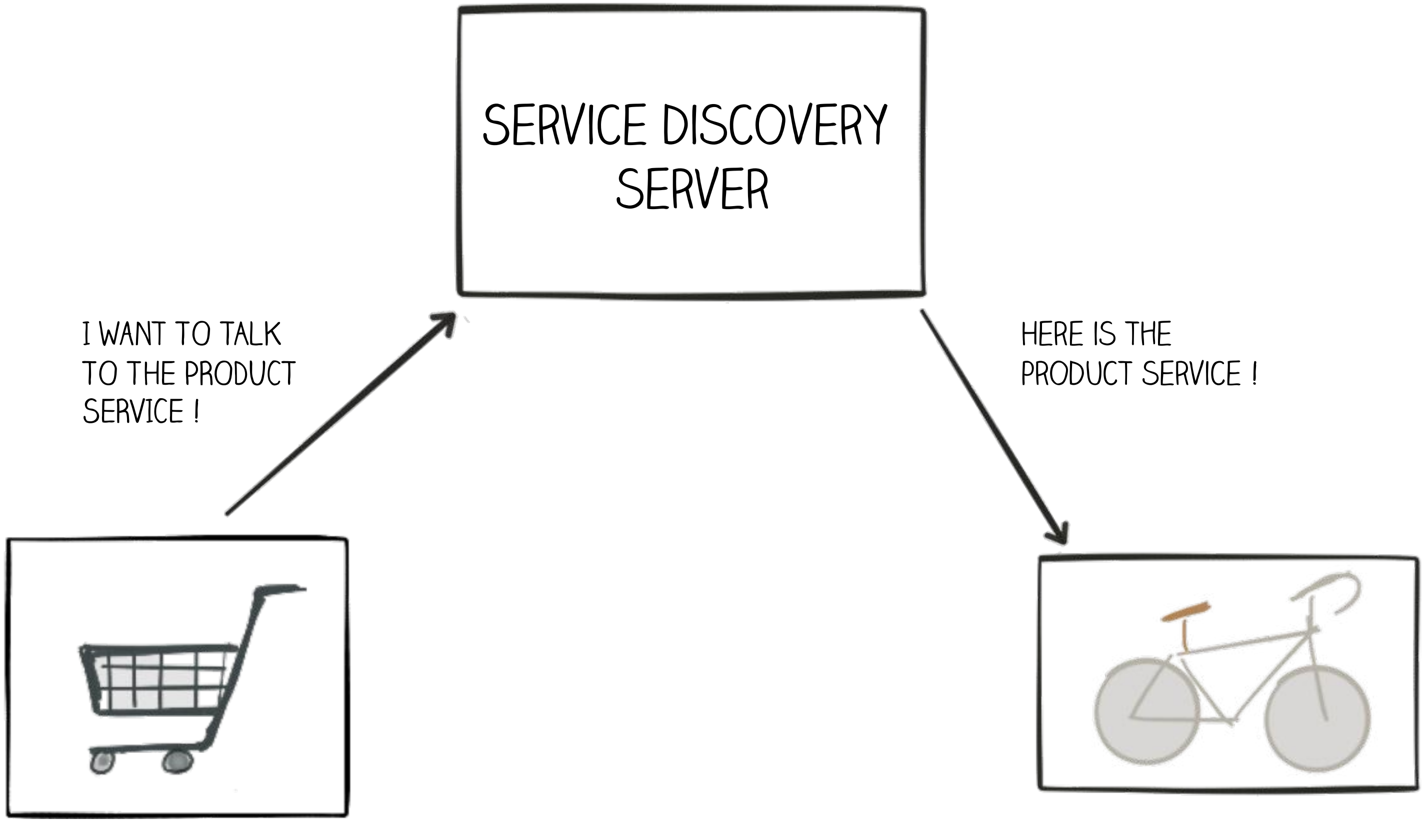


SERVICE DISCOVERY SERVER

REGISTER

REGISTER

SERVICE DISCOVERY

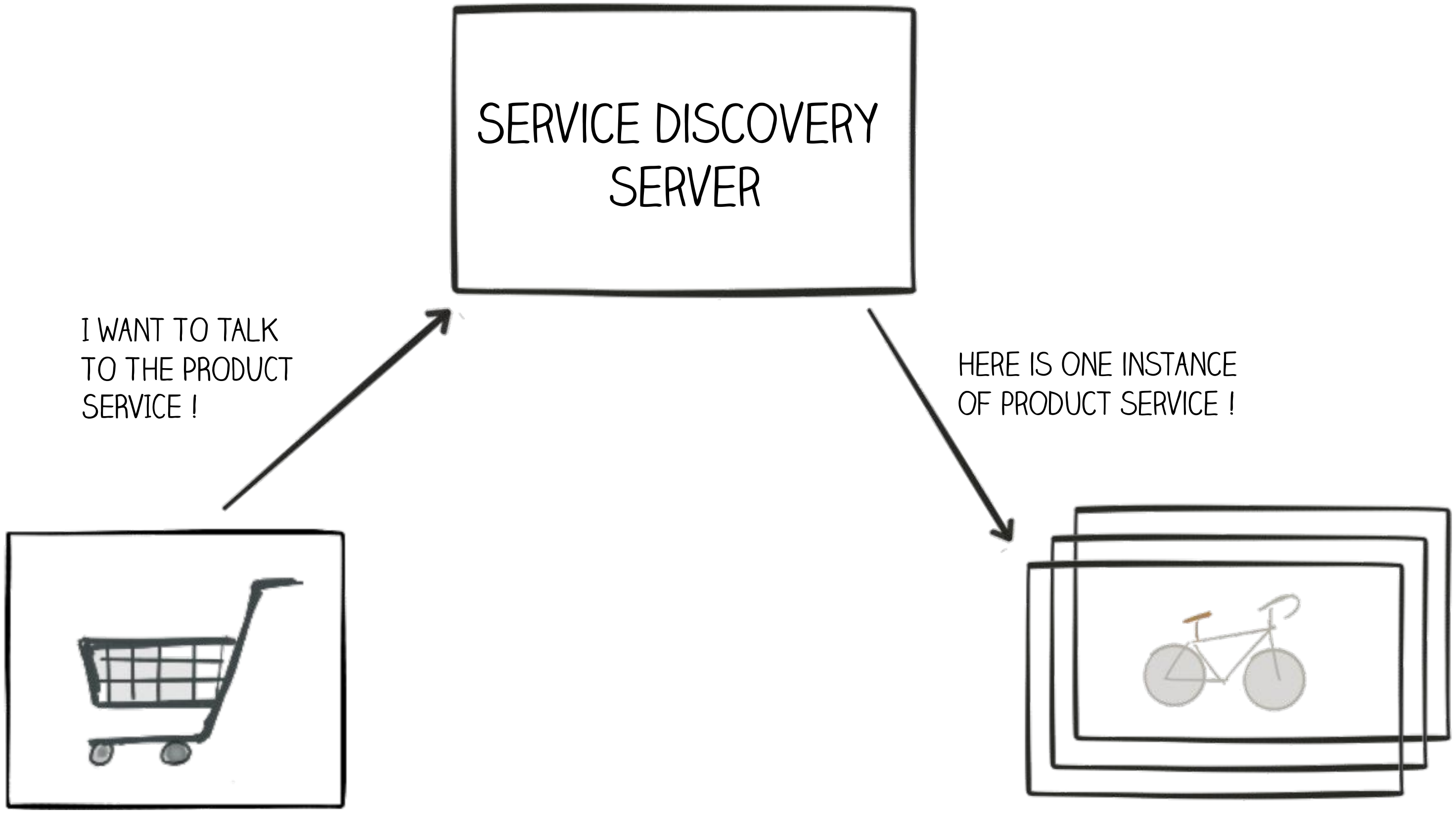


SERVICE DISCOVERY SERVER

I WANT TO TALK TO THE PRODUCT SERVICE !

HERE IS THE PRODUCT SERVICE !

SERVICE DISCOVERY (LOAD BALANCING)

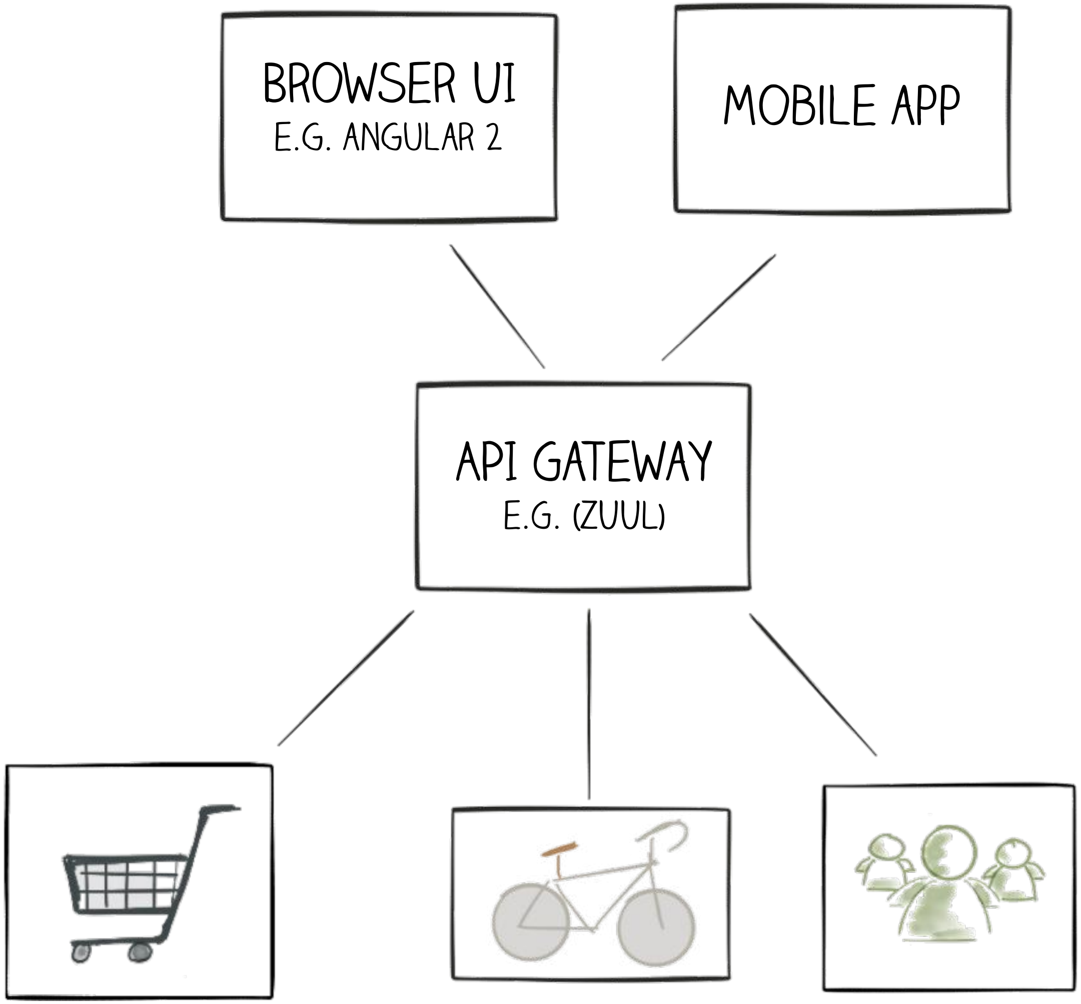


SERVICE DISCOVERY SERVER

I WANT TO TALK TO THE PRODUCT SERVICE !

HERE IS ONE INSTANCE OF PRODUCT SERVICE !

API GATEWAY



BROWSER UI

E.G. ANGULAR 2

MOBILE APP

API GATEWAY

E.G. (ZUUL)

THIS IS NOT NEW!

THE OLD NEW THING...

PRINCIPLES

▨ MODULARITY
▨ AUTONOMOUS
▨ HIDE IMPLEMENTATION DETAILS
▨ AUTOMATION
▨ STATELESS
▨ HIGHLY OBSERVABLE

ADVANTAGES

▨ POLYGLOT ARCHITECTURE
▨ EVOLUTIONARY DESIGN
▨ SELECTIVE SCALABILITY
▨ BIG VS SMALL

DRAWBACKS

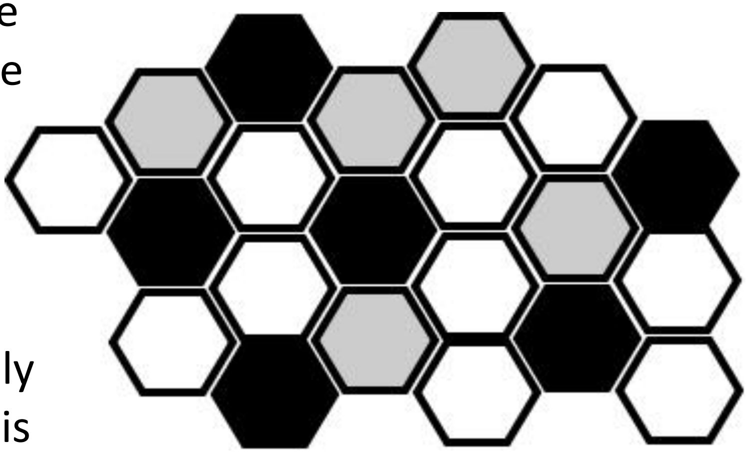
▨ DISTRIBUTED SYSTEM
▨ SYNCHRONOUS VS ASYNCHRONOUS
▨ SLOW (HTTP)
▨ REQUIRES AN ECOSYSTEM

ECOSYSTEM

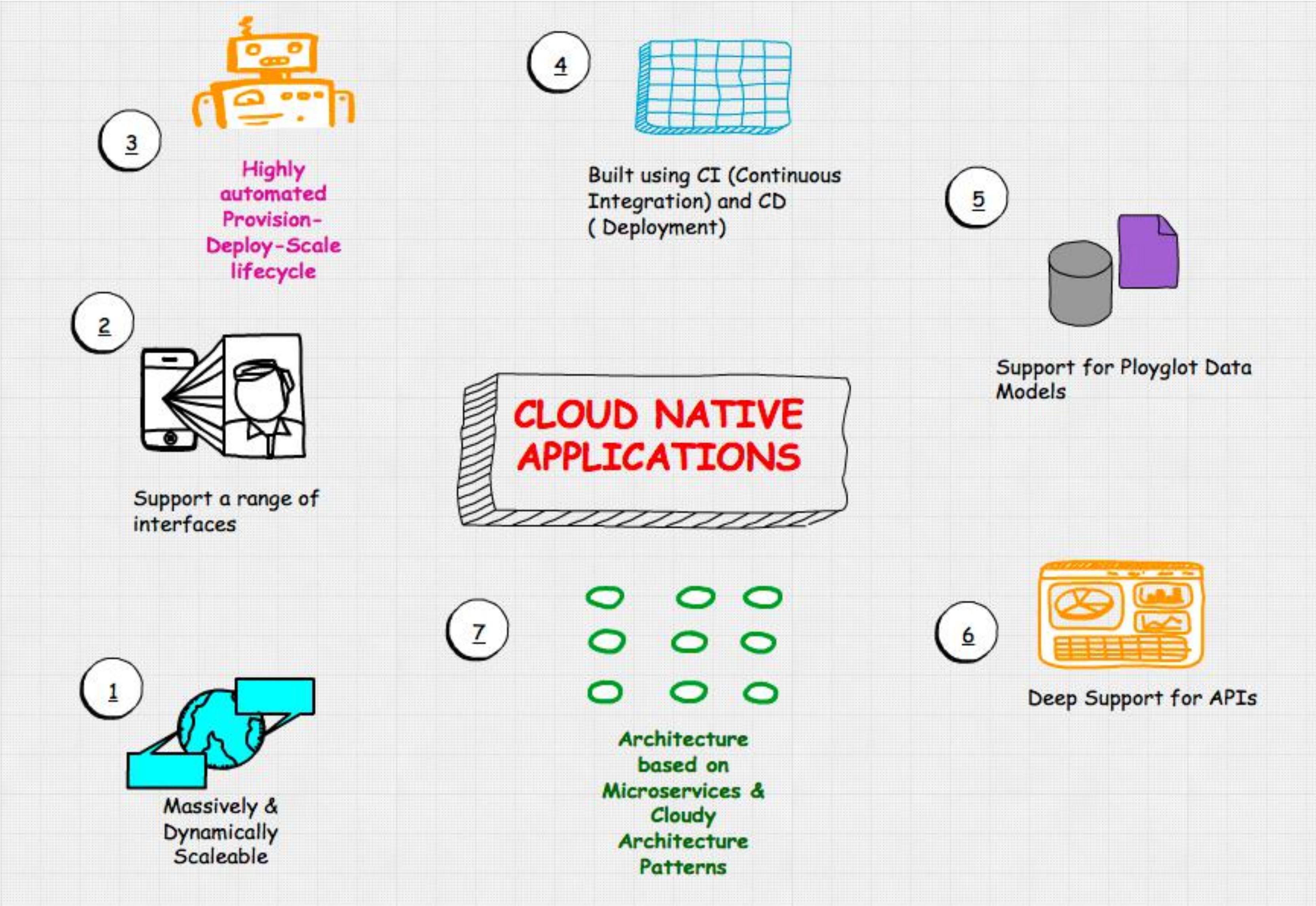
▨ LOAD BALANCER
▨ SERVICE DISCOVERY
▨ API GATEWAY

Honeycomb analogy

- In the real world, bees build a honeycomb by aligning hexagonal wax cells.
- They start small, using different materials to build the cells. Construction is based on what is available at the time of building. Repetitive cells form a pattern and result in a strong fabric structure. Each cell in the honeycomb is independent but also integrated with other cells.
- By adding new cells, the honeycomb grows organically to a big, solid structure. The content inside each cell is abstracted and not visible outside. Damage to one cell does not damage other cells, and bees can reconstruct these cells without impacting the overall honeycomb.

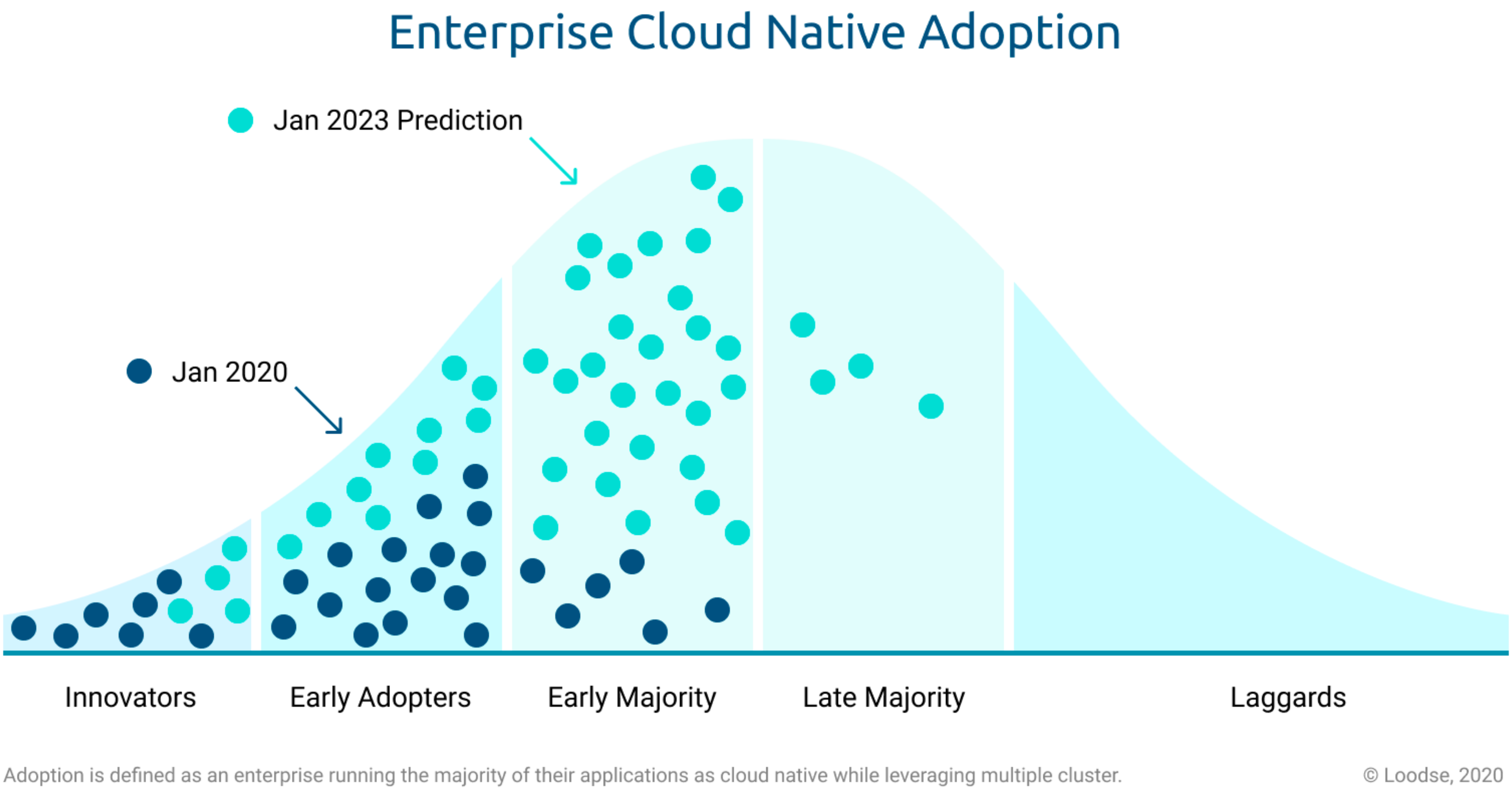


Cloud Native Apps



CLOUD NATIVE APPLICATIONS
1
Massively & Dynamically Scaleable
2
Support a range of interfaces
3
Highly automated Provision-Deploy-Scale lifecycle
4
Built using CI (Continuous Integration) and CD (Deployment)
5
Support for Ployglot Data Models
6
Deep Support for APIs
7
Architecture based on Microservices & Cloudy Architecture Patterns

Cloud Native Adoption

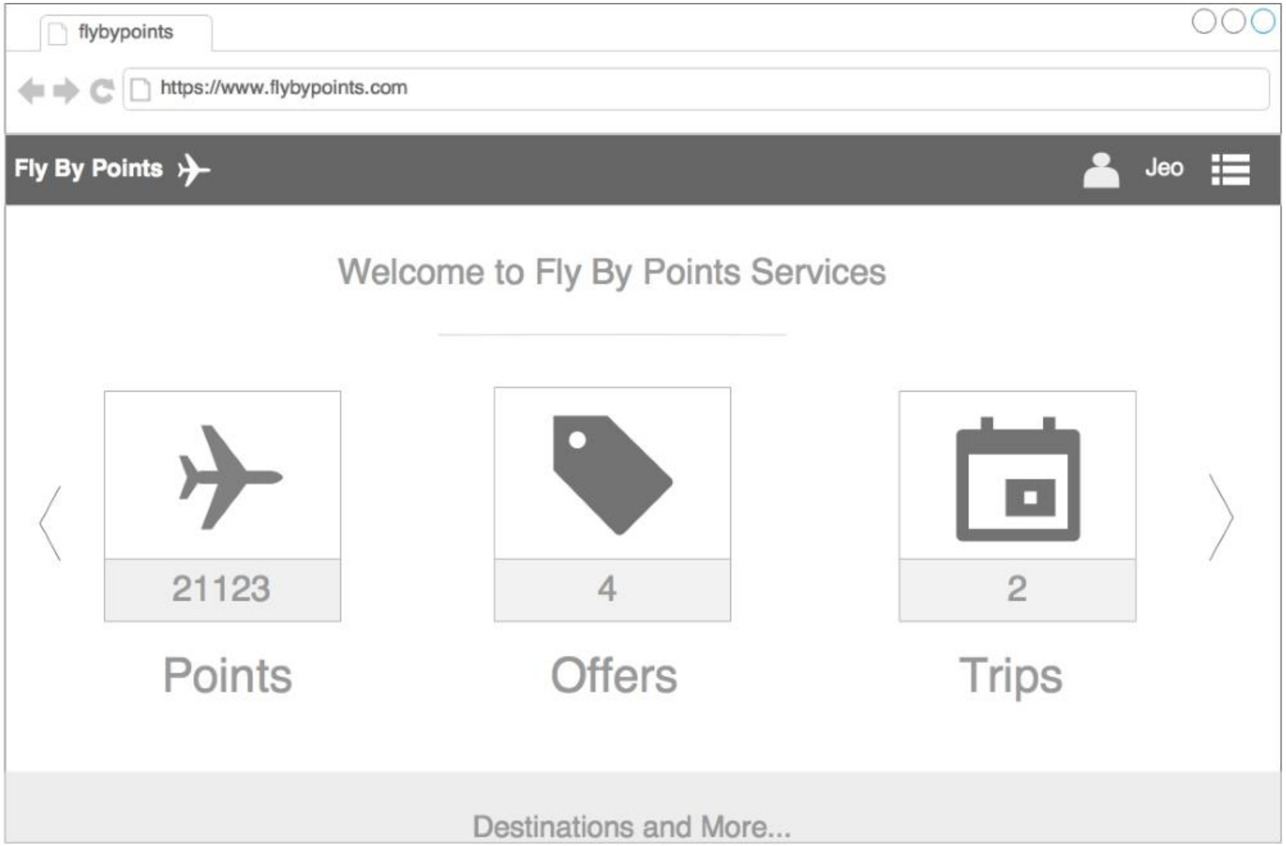


Adoption is defined as an enterprise running the majority of their applications as cloud native while leveraging multiple cluster.

© Loodse, 2020

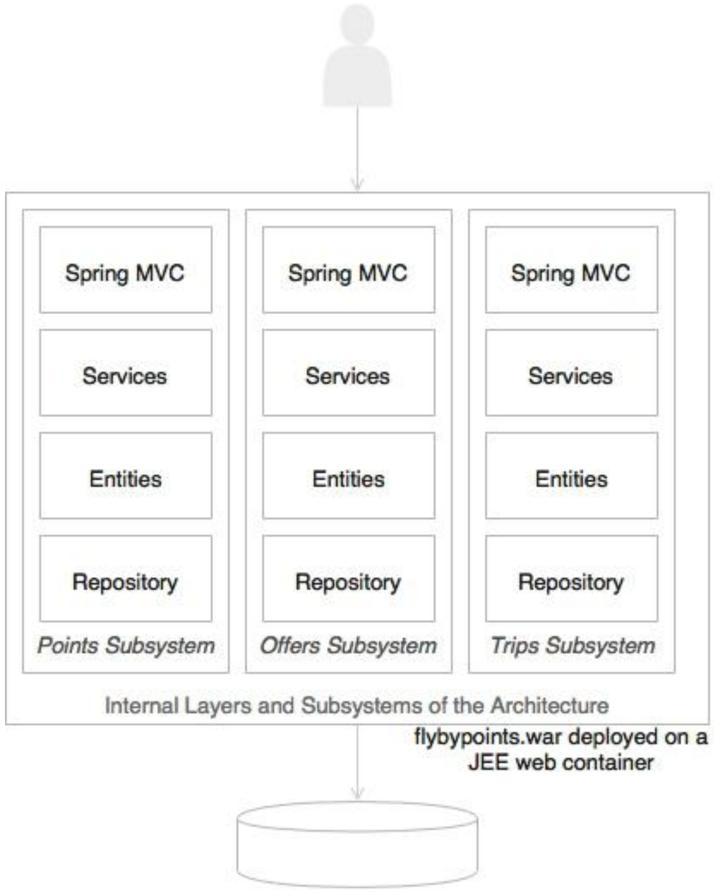
Microservices examples

- There is no "one size fits all" approach when implementing microservices. In this section, different examples are analyzed to crystalize the microservices concept.
- Fly By Points collects points that are accumulated when a customer books a hotel, flight, or car through the online website. When the customer logs in to the Fly By Points website, he/she is able to see the points accumulated, personalized offers that can be availed of by redeeming the points, and upcoming trips if any.
- Let's assume that the following page is the home page after login. There are two upcoming trips for **Jeo**, four personalized offers, and 21,123 loyalty points. When the user clicks on each of the boxes, the details are queried and displayed.



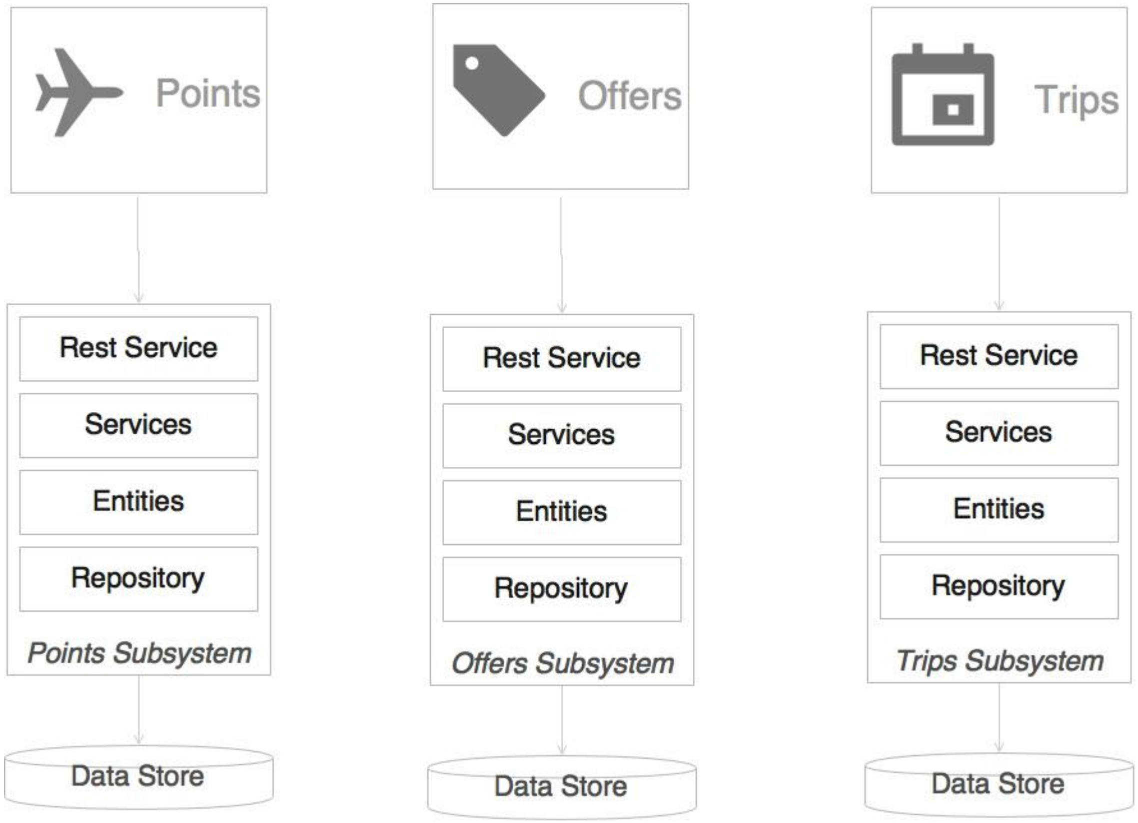
Welcome to Fly By Points Services

Monolithic



- The holiday portal has a Java Spring-based traditional monolithic application architecture.
- the holiday portal's architecture is web-based and modular, with a clear separation between layers. Following the usual practice, the holiday portal is also deployed as a single WAR file on a web server such as Tomcat.
- Data is stored on an all-encompassing backing relational database.
- This is a good fit for the purpose architecture when the complexities are few.

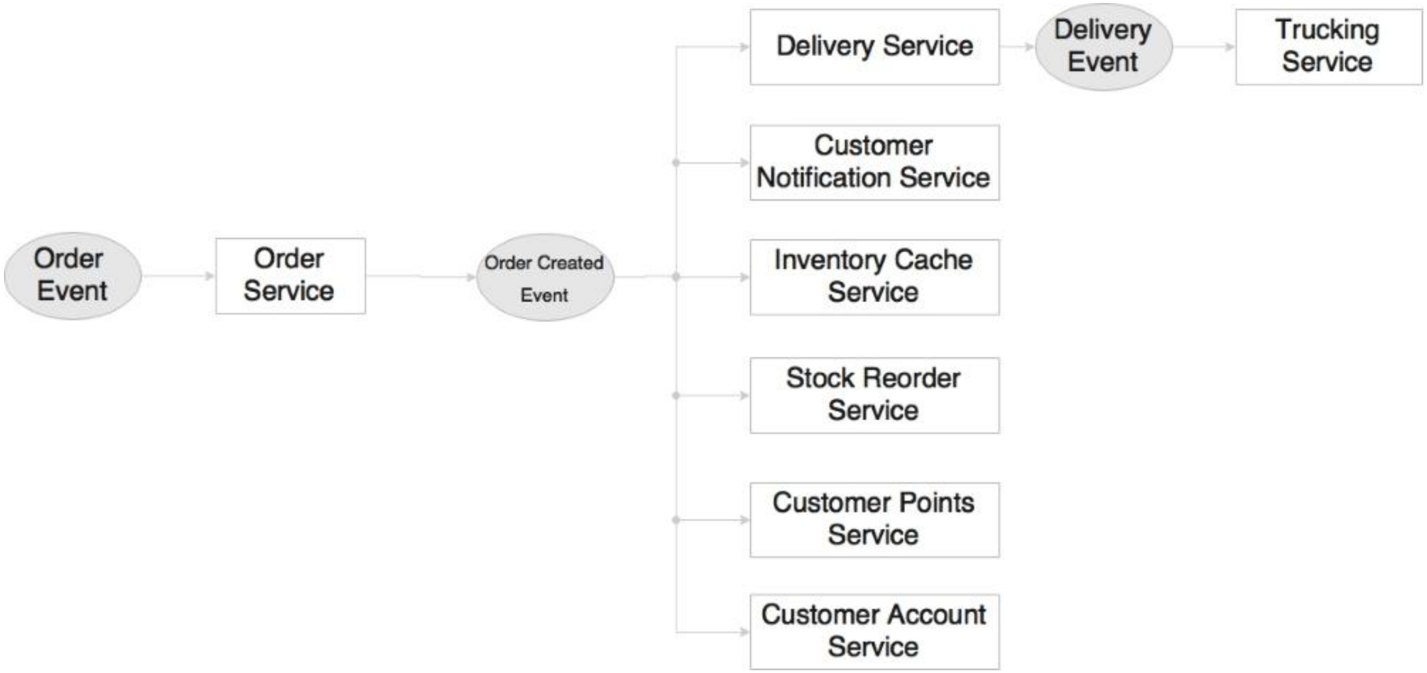
Fly By Point microservices



- As the business grows, the user base expands, and the complexity also increases.
- This results in a proportional increase in transaction volumes.
- At this point, enterprises should look to rearchitecting the monolithic application to microservices for better speed of delivery, agility, and manageability.

Fly By Point microservices

- Examining the simple microservices version of this application, we can immediately note a few things in this architecture:
  - Each subsystem has now become an independent system by itself, a microservice. There are three microservices representing three business functions: **Trips**, **Offers**, and **Points**. Each one has its internal data store and middle layer. The internal structure of each service remains the same.
  - Each service encapsulates its own database as well as its own HTTP listener. As opposed to the previous model, there is no web server or WAR. Instead, each service has its own embedded HTTP listener, such as Jetty, Tomcat, and so on.
  - Each microservice exposes a REST service to manipulate the resources/entity that belong to this service.
- When the web page is loaded, all the three boxes, Trips, Offers, and Points will be displayed with details such as points, the number of offers, and the number of trips. This will be done by each box independently making asynchronous calls to the respective backend microservices using REST.
- There is no dependency between the services at the service layer. When the user clicks on any of the boxes, the screen will be transitioned and will load the details of the item clicked on. This will be done by making another call to the respective microservice.



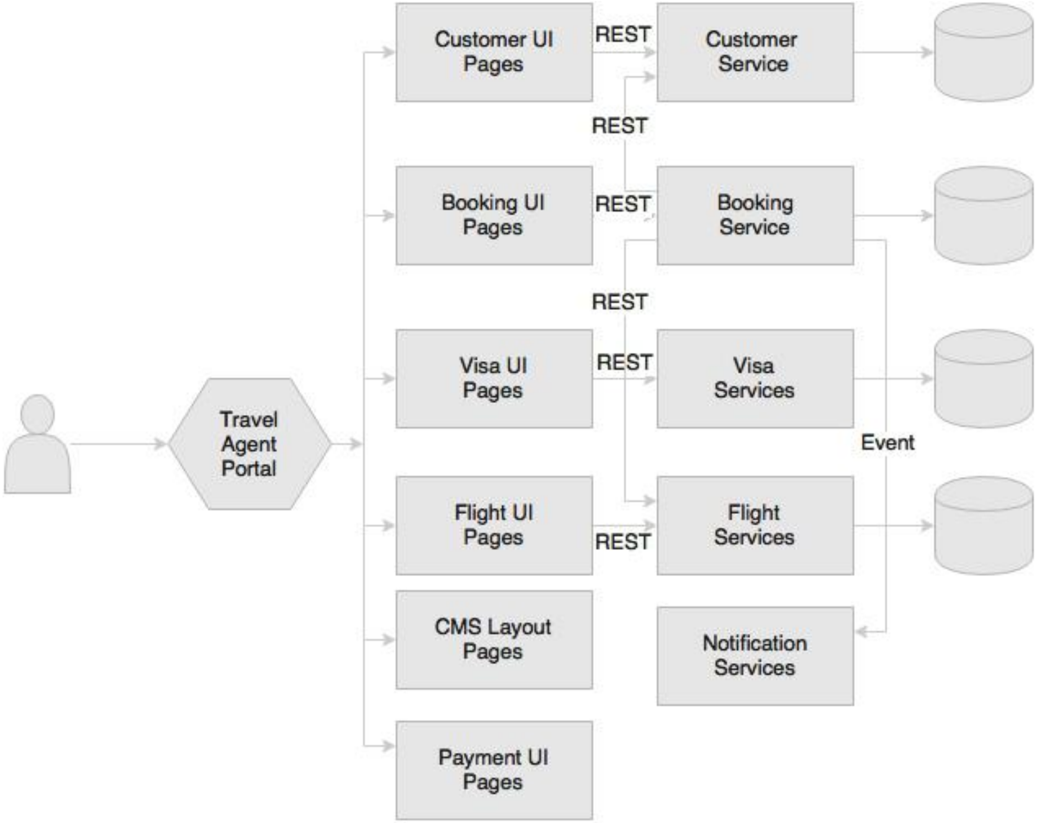
- Order Service which processes the Order Event generated when a customer places an order through the website.
- This microservices system is completely designed based on reactive programming practices.

Microservices Event Driven Example

- When an event is published, a number of microservices are ready to kick-start upon receiving the event. Each one of them is independent and does not rely on other microservices. The advantage of this model is that we can keep adding or replacing microservices to achieve specific needs.
- In the preceding diagram, there are eight microservices shown. The following activities take place upon the arrival of **Order Event**:
  - Order Service kicks off when Order Event is received. Order Service creates an order and saves the details to its own database.
  - If the order is successfully saved, Order Successful Event is created by Order Service and published.
  - A series of actions take place when Order Successful Event arrives.
  - Delivery Service accepts the event and places Delivery Record to deliver the order to the customer. This, in turn, generates Delivery Event and publishes the event.

Microservices Event Driven Example

- The following activities take place upon the arrival of **Order Event**:
  - Trucking Service picks up Delivery Event and processes it. For instance, Trucking Service creates a trucking plan.
  - Customer Notification Service sends a notification to the customer informing the customer that an order is placed.
  - Inventory Cache Service updates the inventory cache with the available product count.
  - Stock Reorder Service checks whether the stock limits are adequate and generates Replenish Event if required.
  - Customer Points Service recalculates the customer's loyalty points based on this purchase.
  - **Customer Account Service** updates the order history in the customer's account.
- Each service is responsible for only one function. Services accept and generate events. Each service is independent and is not aware of its neighborhood. Hence, the neighborhood can organically grow as mentioned in the honeycomb analogy. New services can be added as and when necessary. Adding a new service does not impact any of the existing services.

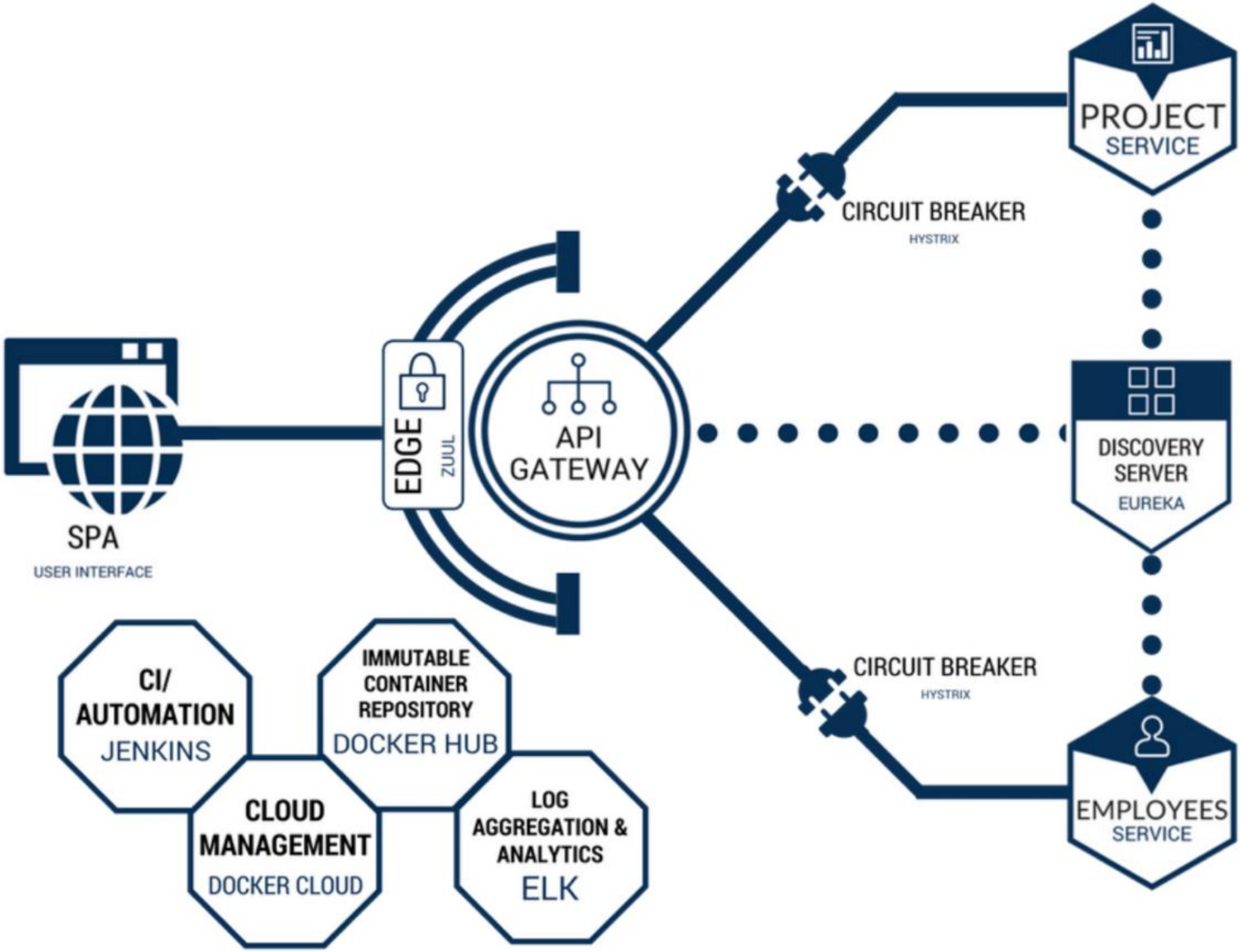


- we will see both synchronous REST calls as well as asynchronous events.
- the portal is just a container application with multiple menu items or links in the portal.
- When specific pages are requested—for example, when the menu or a link is clicked on—they will be loaded from the specific microservices.

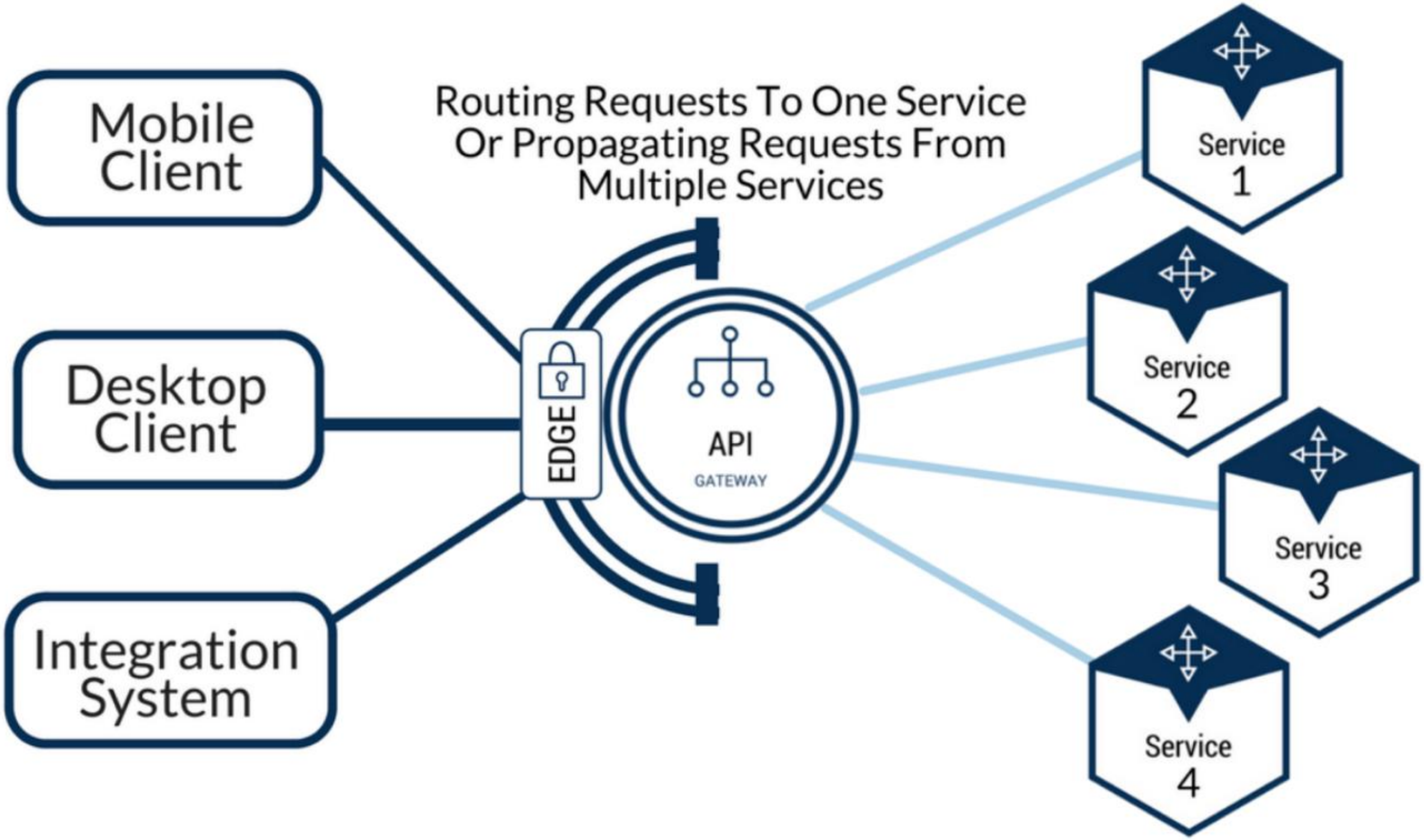
a travel agent portal

- The interesting factor here is that we can change the user interface, logic, and data of a microservice without impacting any other microservices.
- This is a clean and neat approach. A number of portal applications can be built by composing different screens from different microservices, especially for different user communities. The overall behavior and navigation will be controlled by the portal application.
- The approach has a number of challenges unless the pages are designed with this approach in mind.

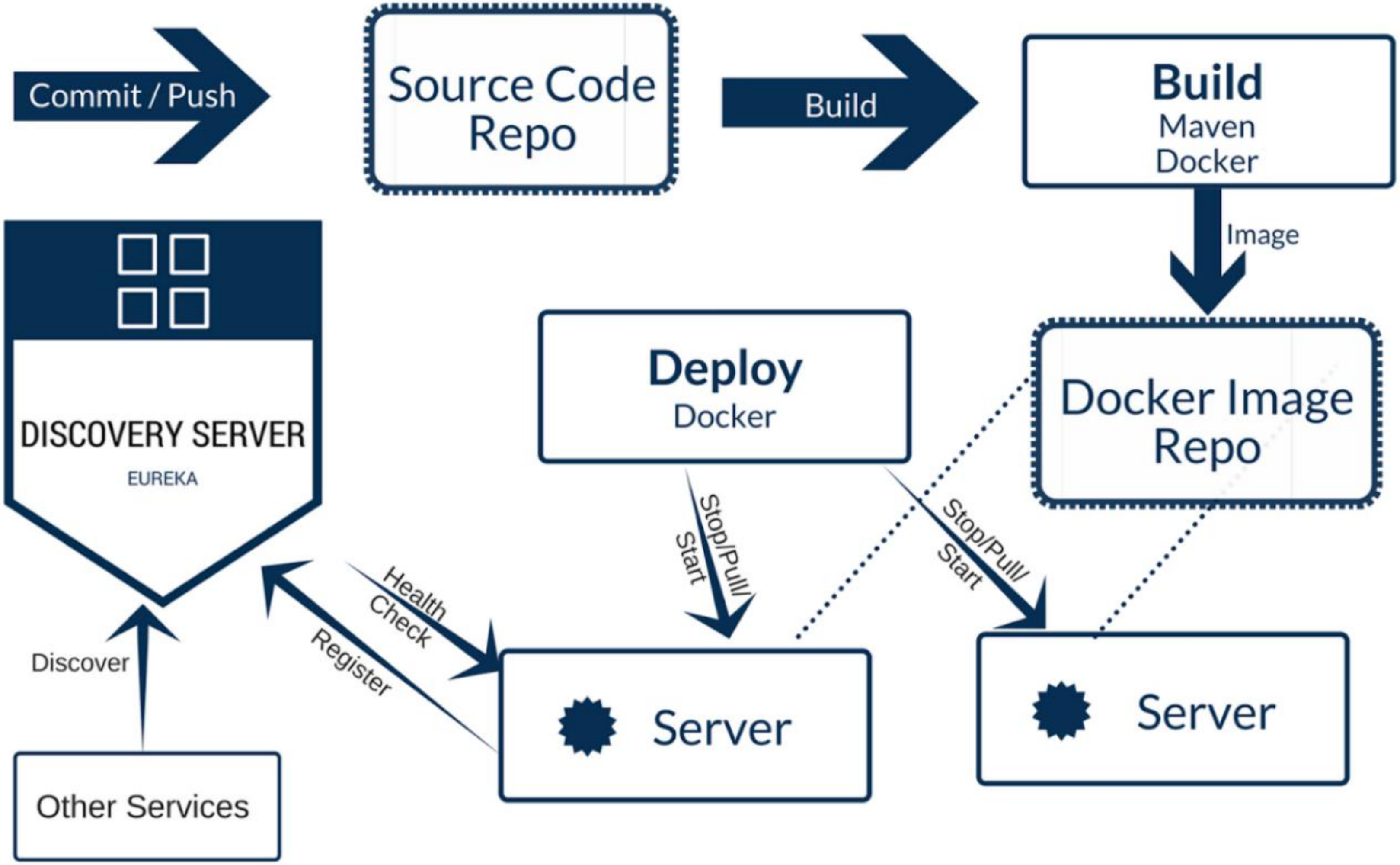
Common Micro Service Patterns

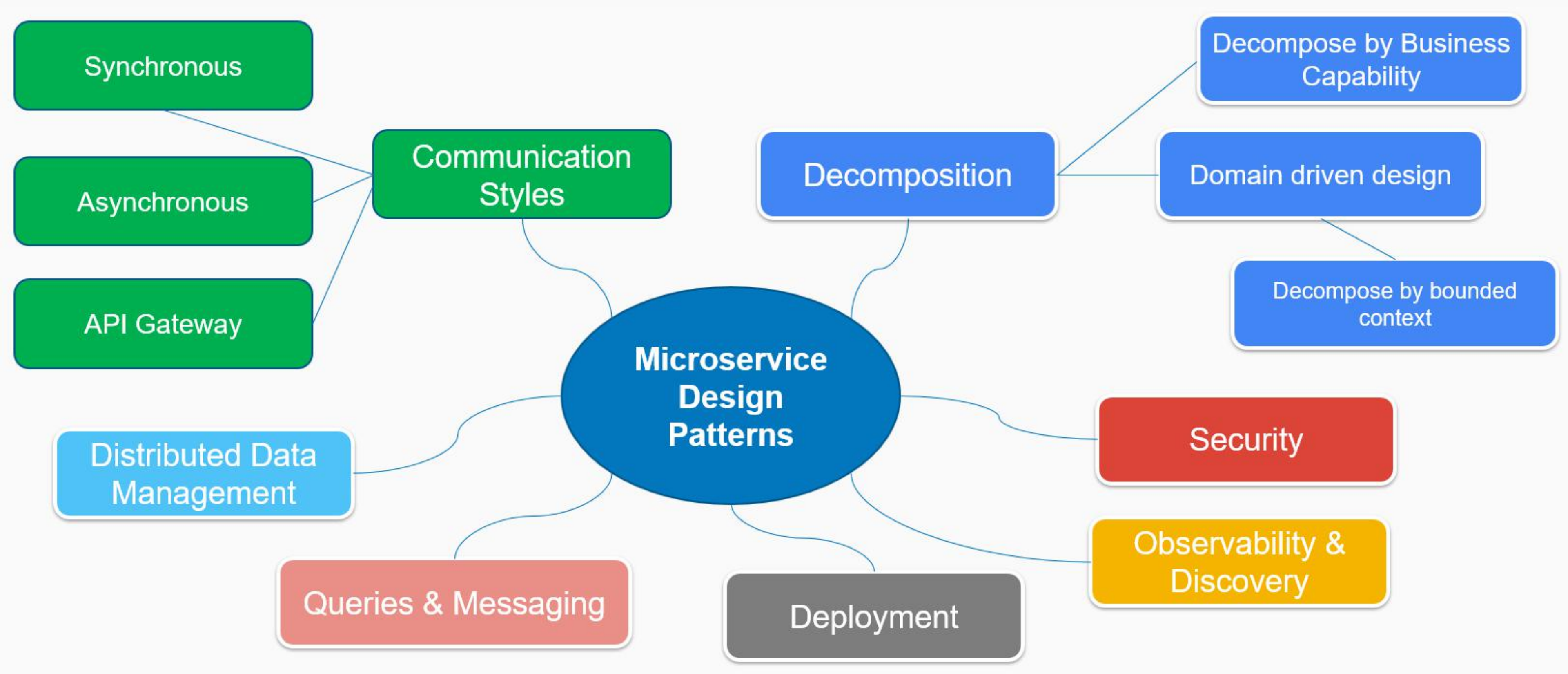


API Gateway


Mobile Client
Desktop Client
Integration System
Routing Requests To One Service Or Propagating Requests From Multiple Services
EDGE
API
GATEWAY
Service 1
Service 2
Service 3
Service 4

CI/CD generic





Microservice Design Patterns

Communication Styles

Synchronous

Asynchronous

API Gateway

Decomposition

Decompose by Business Capability

Domain driven design

Decompose by bounded context

Distributed Data Management

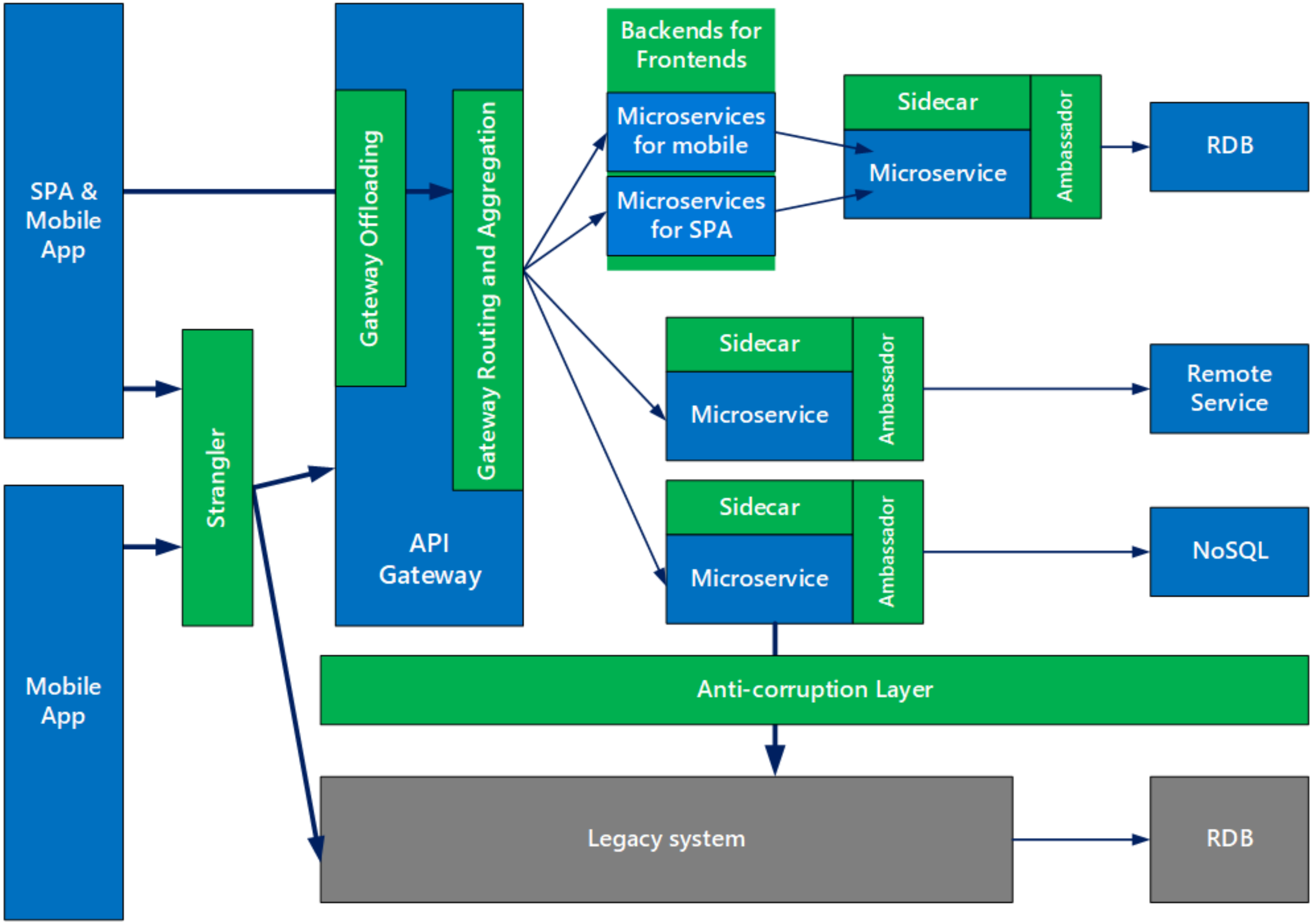
Security

Queries & Messaging

Deployment

Observability & Discovery

Common Patterns



Common Patterns

- **Ambassador** can be used to offload common client connectivity tasks such as monitoring, logging, routing, and security (such as TLS) in a language agnostic way.
- **Anti-corruption layer** implements a façade between new and legacy applications, to ensure that the design of a new application is not limited by dependencies on legacy systems.
- **Backends for Frontends** creates separate backend services for different types of clients, such as desktop and mobile. That way, a single backend service doesn’t need to handle the conflicting requirements of various client types. This pattern can help keep each microservice simple, by separating client-specific concerns.
- **Bulkhead** isolates critical resources, such as connection pool, memory, and CPU, for each workload or service. By using bulkheads, a single workload (or service) can’t consume all of the resources, starving others. This pattern increases the resiliency of the system by preventing cascading failures caused by one service.

Common Patterns Contd..

- **Gateway Aggregation** aggregates requests to multiple individual microservices into a single request, reducing chattiness between consumers and services.
- **Gateway Offloading** enables each microservice to offload shared service functionality, such as the use of SSL certificates, to an API gateway.
- **Gateway Routing** routes requests to multiple microservices using a single endpoint, so that consumers don't need to manage many separate endpoints.
- **Sidecar** deploys helper components of an application as a separate container or process to provide isolation and encapsulation.
- **Strangler** supports incremental migration by gradually replacing specific pieces of functionality with new services.

References

- https://azure.microsoft.com/pt-br/blog/design-patterns-for-microservices/
- https://microservices.io/index.html

THANKS !