Event Driven Architecture

Júlio Alcântara Tavares, MSc
Instrutor

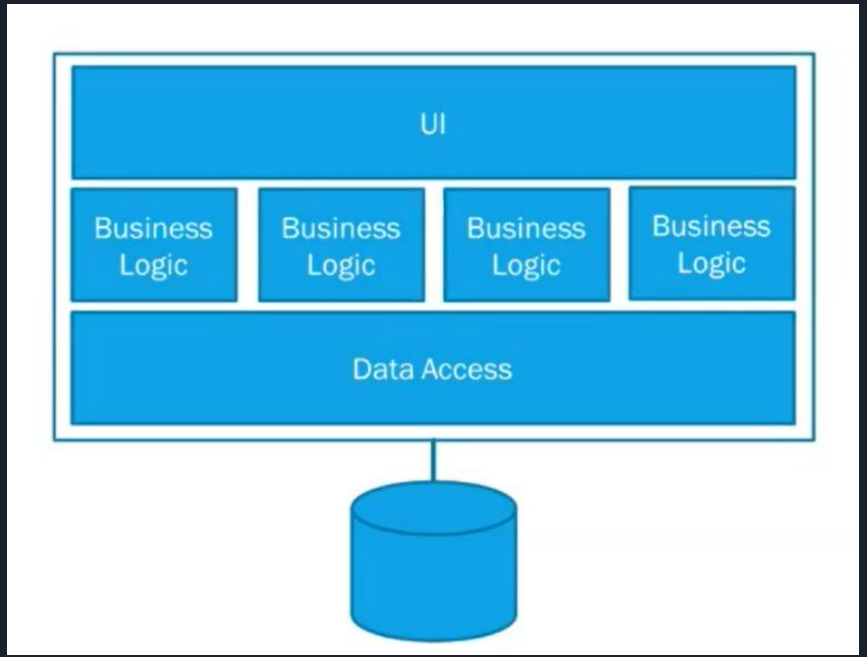
“NO INÍCIO:”

ARQUITETURA “MONOLÍTICA”

Multi Tier Architecture

Vantagens

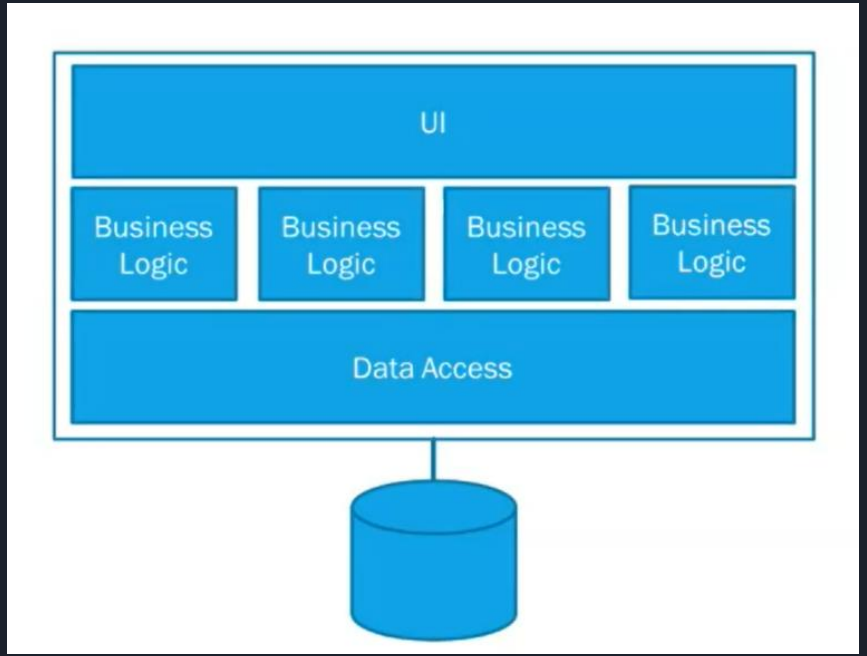
- Possível acoplamento das responsabilidades (se não controlado).
- Baixa complexidade de orquestração de serviços.
- Fácil de testar por inversão de controle.



Multi Tier Architecture

Desvantagens

- Mudança em um módulo pode impactar em todos os demais.
- Acoplamento forte entre as camadas.
- Difícil de mudar sem implantações em larga escala.



Contexto

Necessidades das aplicações “modernas”:

- Suportar uma ampla variedade de clientes (desktop, browsers, mobile, etc).
- Necessidade crescente de exposição e fornecimento de APIs.
- Integração frequente com outras aplicações.
- Executar regras de negócio complexas, acessar BDs e repositórios de dados, trocar mensagens com outros sistemas.
- Processamento e Armazenamento Escalável (Escalabilidade).

Contexto

Cenário “usualmente” encontrado:

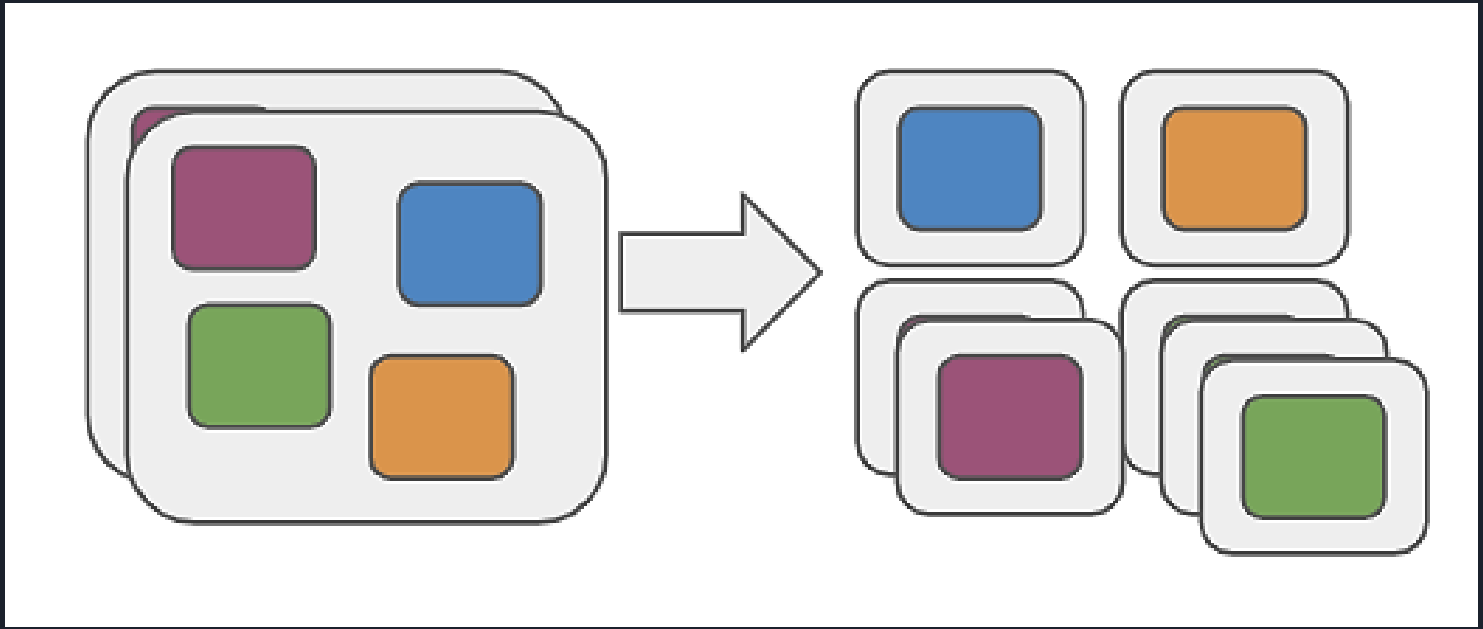
- Há uma equipe de desenvolvedores trabalhando no aplicativo.
- Novos membros da equipe devem se tornar rapidamente produtivos.
- O aplicativo deve ser fácil de entender e modificar.
- Você deseja praticar a implantação contínua do aplicativo.
- Você deve executar várias instâncias do aplicativo em várias máquinas para satisfazer os requisitos de escalabilidade e disponibilidade.
- É necessário aproveitar as tecnologias emergentes (estruturas, linguagens de programação etc).

QUAL ARQUITETURA DE DEPLOYMENT UTILIZAR?

Uma Possível Solução

"Definir uma arquitetura que estrutura o aplicativo como um conjunto de serviços colaborativos, fracamente acoplados."

Análise de Cenário



MICROSERVICES ARCHITECTURE

Diretrizes desta Arquitetura

- Altamente manutenível e testável - permite desenvolvimento e implantação rápidos e frequentes.
- Acoplado livremente a outros serviços - permite que uma equipe trabalhe de forma independente a maior parte do tempo em seus serviços, sem ser impactada por alterações em outros serviços e sem afetar outros serviços.

Diretrizes desta Arquitetura

- Implantável independentemente - permite que uma equipe implante seu serviço sem precisar se coordenar com outras equipes.
- Capaz de ser desenvolvido por uma equipe pequena - essencial para alta produtividade, evitando o alto chefe de comunicação de equipes grandes.

Diretrizes desta Arquitetura

- Os serviços se comunicam usando protocolos síncronos como HTTP / REST ou protocolos assíncronos como AMQP.
- Os serviços podem ser desenvolvidos e implantados independentemente um do outro.
- Cada serviço possui seu próprio banco de dados para ser dissociado de outros serviços.
- A consistência dos dados entre serviços é mantida usando o padrão Saga (Saga Patterns).

Benefícios

Permite a entrega e implantação contínuas de aplicativos grandes e complexos (CI/CD):

- Manutenção melhorada - cada serviço é relativamente pequeno e, portanto, é mais fácil de entender e mudar.
- Melhor testabilidade - os serviços são menores e mais rápidos para testar.
- Melhor capacidade de implantação - os serviços podem ser implantados independentemente.

Benefícios

Permite a entrega e implantação contínuas de aplicativos grandes e complexos (CI/CD):

- Permite organizar o esforço de desenvolvimento em torno de várias equipes autônomas.
- Cada equipe possui e é responsável por um ou mais serviços.
- Cada equipe pode desenvolver, testar, implantar e dimensionar seus serviços independentemente de todas as outras equipes.

Benefícios

Cada microservice é relativamente pequeno:

- Mais fácil para um desenvolvedor entender.
- Ambiente de desenvolvimento mais rápido, tornando os desenvolvedores mais produtivos.
- O aplicativo inicia mais rápido, o que torna os desenvolvedores mais produtivos e acelera as implantações.

Benefícios

Melhor isolamento de falhas:

- Se ocorrerem problemas em um serviço, apenas esse serviço será afetado e os outros serviços continuarão a lidar com solicitações.

- Em comparação, um componente que se comporta mal de uma arquitetura monolítica pode derrubar todo o sistema.

Benefícios

Elimina o "compromisso" de longo prazo com uma "tech stack":

- Ao desenvolver cada novo serviço, é possível escolher novas tecnologias.
- Da mesma forma, ao realizar grandes alterações em um serviço existente, é possível reescrevê-lo usando uma nova stack tecnológica.

Desvantagens

Os desenvolvedores devem lidar com a complexidade adicional de criar um sistema distribuído:

- Os desenvolvedores devem implementar o mecanismo de comunicação entre serviços e lidar com falhas parciais.

- Implementar solicitações que abrangem vários serviços é mais difícil
- Testar as interações entre serviços é mais difícil.

Desvantagens

Os desenvolvedores devem lidar com a complexidade adicional de criar um sistema distribuído:

- A implementação de solicitações que abrangem vários serviços requer uma coordenação cuidadosa entre as equipes.

- As ferramentas / IDEs para desenvolvedores são orientadas para a criação de aplicativos monolíticos e não fornecem suporte explícito ao desenvolvimento de aplicativos distribuídos.

Desvantagens

- Complexidade operacional de implantar e gerenciar (deployment) um sistema composto por muitos serviços diferentes.

- Maior consumo de memória: A arquitetura de microservices substitui N instâncias de aplicativos monolíticos por NxM instâncias de serviços.
    - Se cada serviço for executado em sua própria JVM (ou equivalente), o que geralmente é necessário para isolar as instâncias, haverá uma sobrecarga de M vezes o tempo de execução da JVM.

QUANDO USAR A ARQUITETURA BASEADA EM MICROSERVICES?

Quando Usar?

- Um dos principais desafios é perceber o melhor momento para usar esta arquitetura.

- Ao desenvolver a primeira versão de um aplicativo, geralmente não temos todos os problemas que essa abordagem resolve. Além disso, o uso de uma arquitetura elaborada e distribuída tornará o desenvolvimento mais lento.

- Mais tarde, no entanto, quando o desafio é como escalar e você precisa usar a decomposição funcional, as dependências emaranhadas podem dificultar a decomposição do aplicativo monolítico em um conjunto de serviços.

COMO DECOMPOR A APLICAÇÃO EM MICROSERVICES?

Como Decompor?

- Decompor por business capability: Uma business capability é um conceito da modelagem de arquitetura de negócios. É algo que uma empresa faz para gerar valor.

- Decompor por subdomínio de DDD (Domain Driven Design).

- Decomponha por verbo ou caso de uso e defina serviços responsáveis por ações específicas. por exemplo. um serviço de remessa responsável pelo envio de pedidos completos.

Como Decompor?

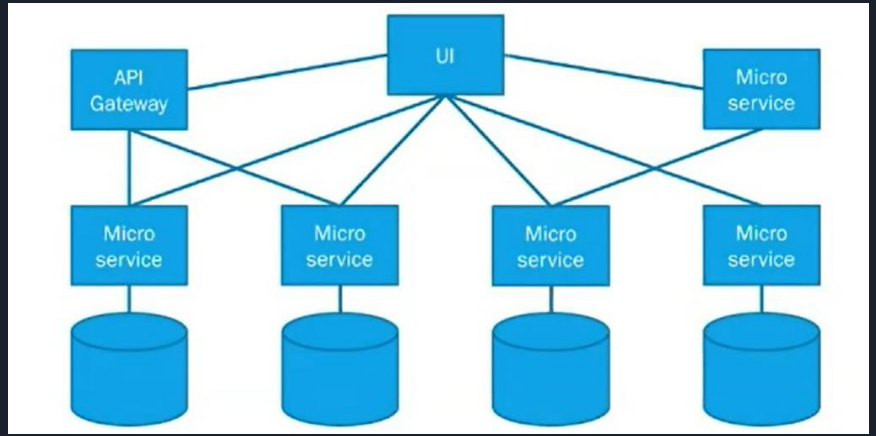
- Decompor por substantivos ou recursos, definindo um serviço que é responsável por todas as operações em entidades / recursos de um determinado tipo.
  - Por exemplo, um serviço de conta seria responsável pelo gerenciamento de contas de usuário.

TRANSIÇÃO: MONOLITHIC PARA MICROSERVICES

Microservices Architecture

Vantagens

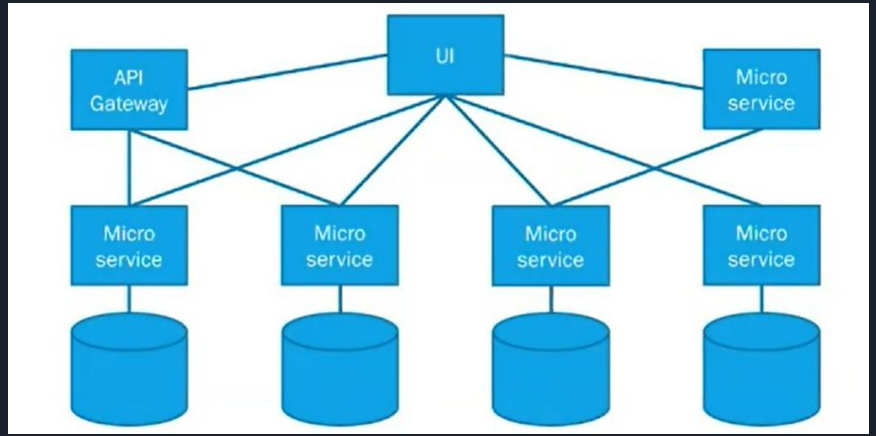
- Limite claro de responsabilidades.
- Deploy extremamente ágil.
- Flexibilidade para usar as melhores habilidades e tecnologias para cada serviço.

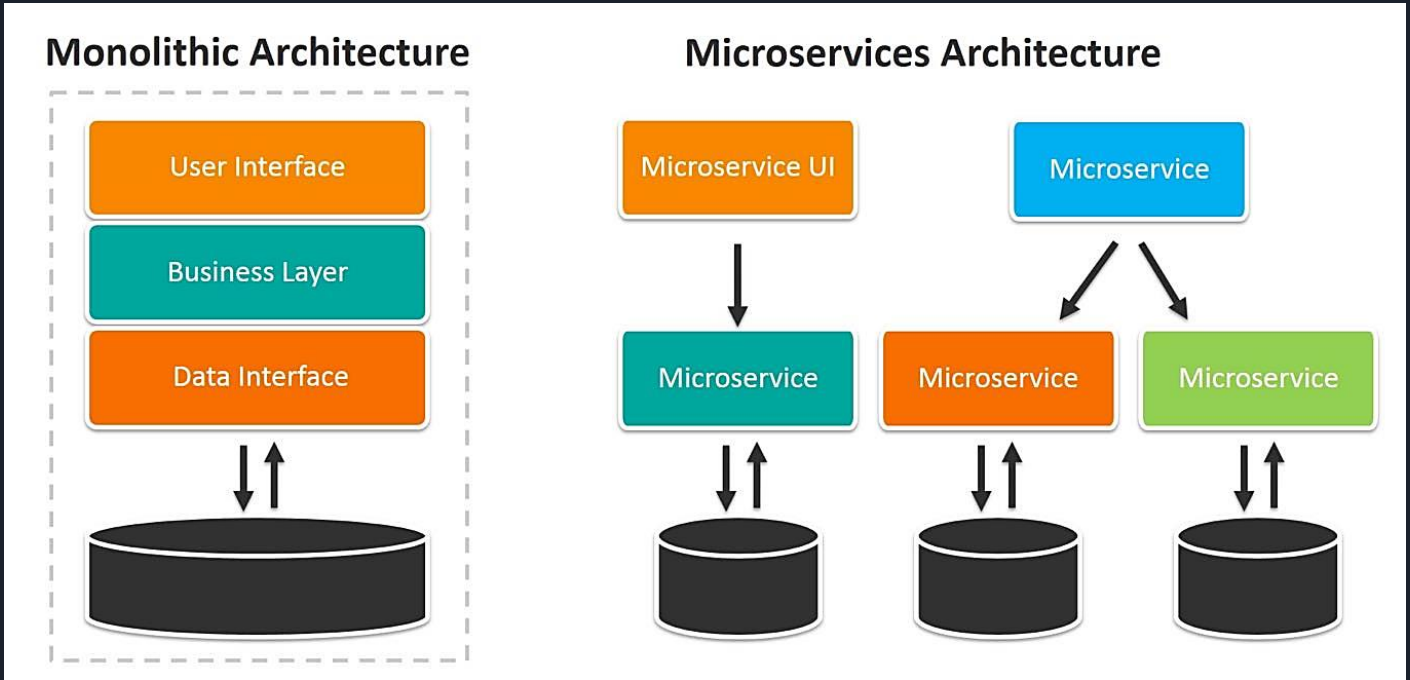


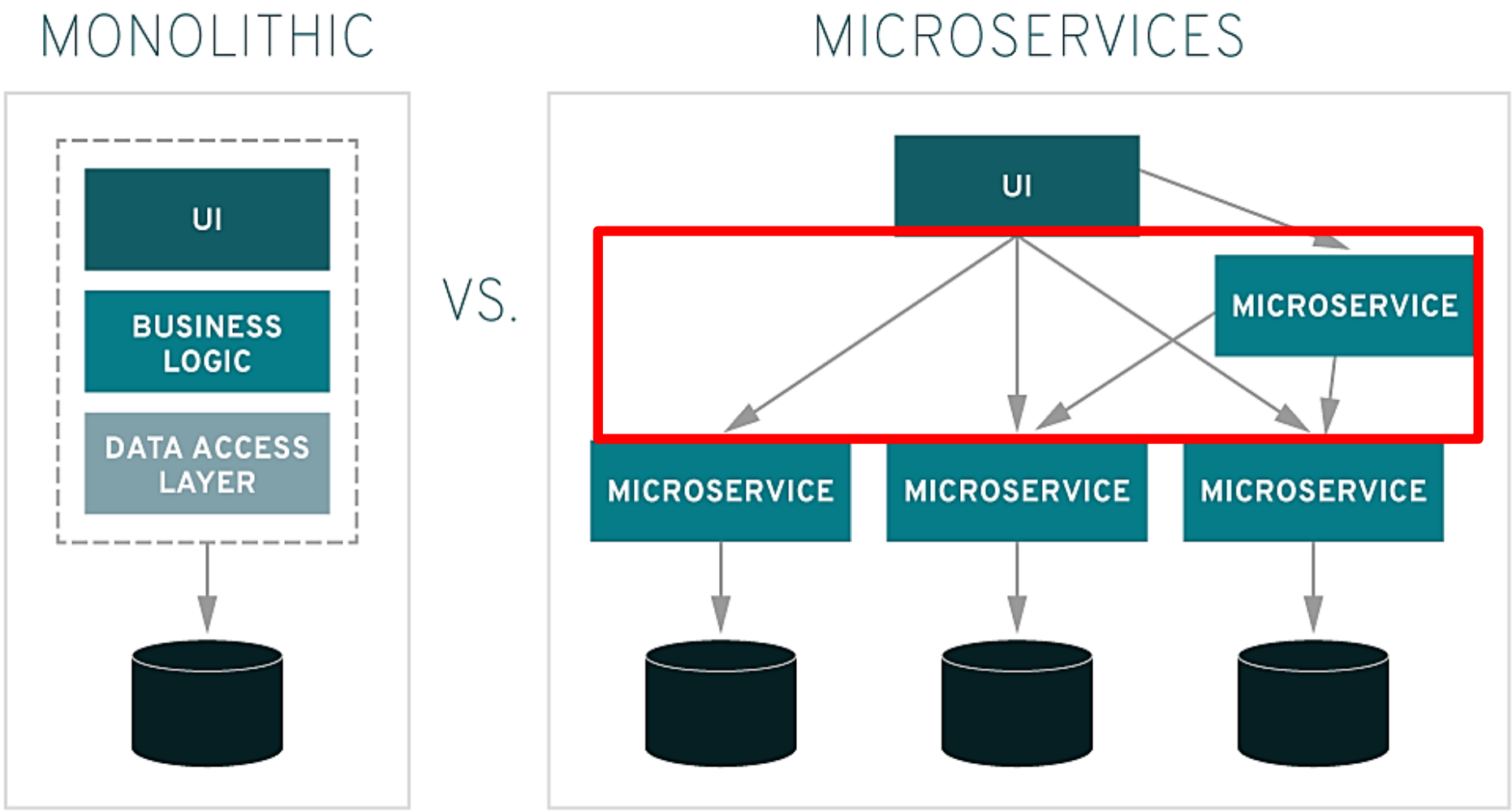
Microservices Architecture

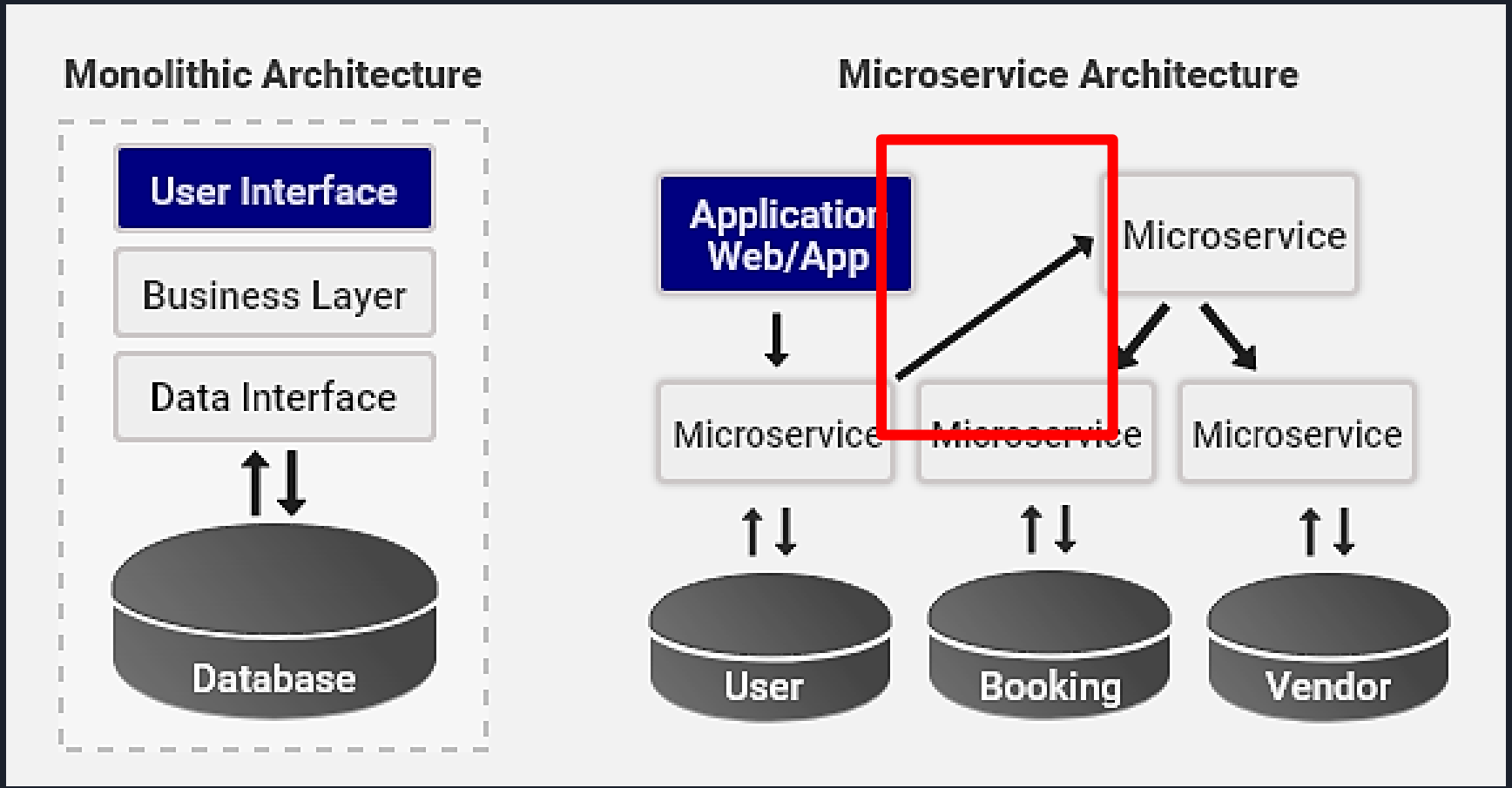
Desvantagens

- Maior complexidade (orquestração).
- Mais difícil manter a consistência: difícil de manter o estado.
- Precisa lidar com todas as transações de maneira diferente.




Monolithic Architecture
User Interface
Business Layer
Data Interface
Microservices Architecture
Microservice UI
Microservice
Microservice
Microservice
Microservice


MONOLITHIC
UI
BUSINESS LOGIC
DATA ACCESS LAYER
VS.
MICROSERVICES
UI
MICROSERVICE
MICROSERVICE
MICROSERVICE
MICROSERVICE


Monolithic Architecture
User Interface
Business Layer
Data Interface
Database
Microservice Architecture
Application
Web/App
Microservice
Microservice
Microservice
Microservice
User
Booking
Vendor

Visão Geral: Arquitetura de Integração

EIP - Enterprise Integration Patterns

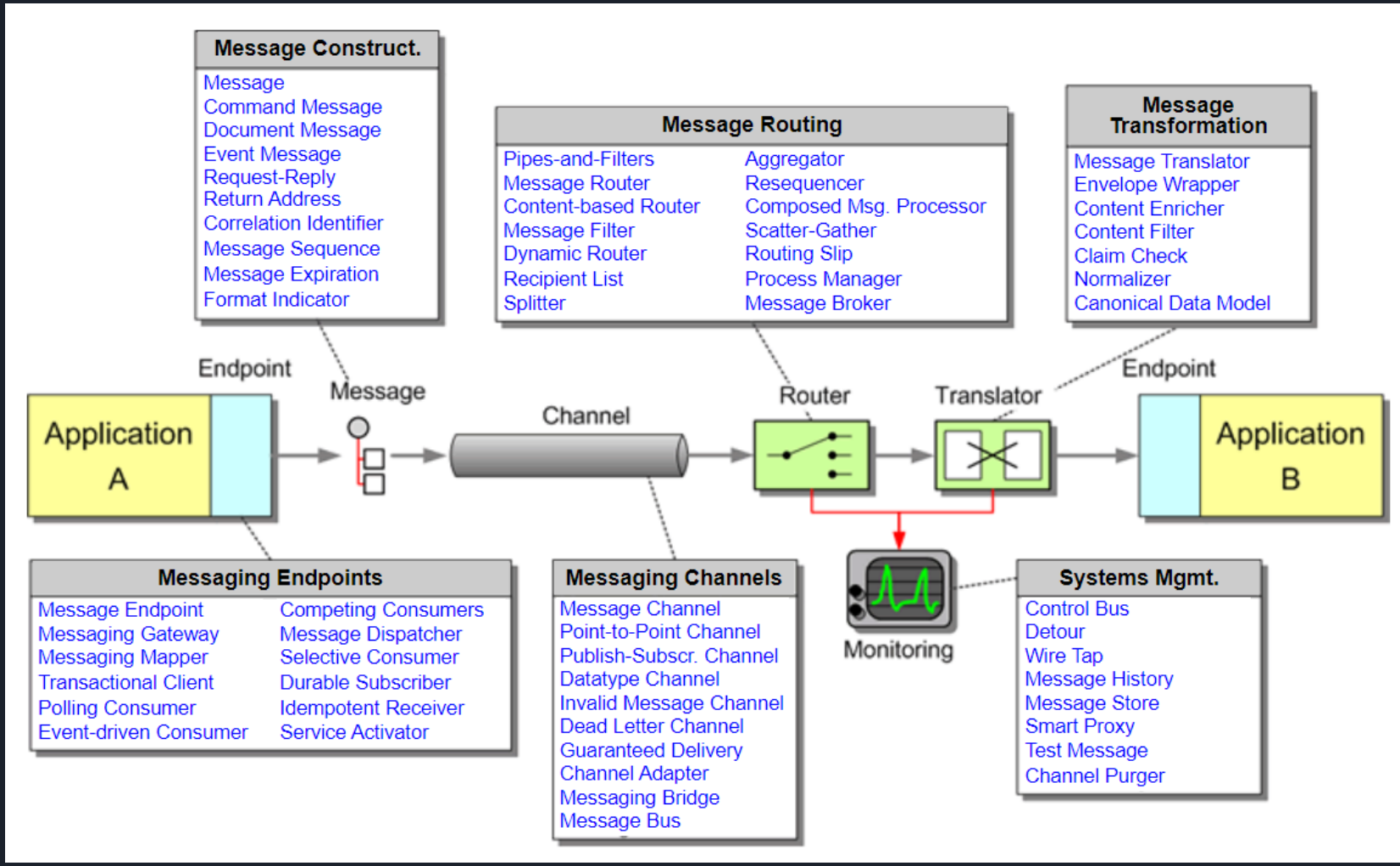
Enterprise Integration Patterns

- Padrões já conhecidos e com funcionamento testado e comprovado.
- Atualmente existem mais de 60 padrões documentados especificamente para mensageria.
- Referência Geral: https://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.html
- Referência Mensageria: https://www.enterpriseintegrationpatterns.com/patterns/messaging/

Enterprise Integration Patterns

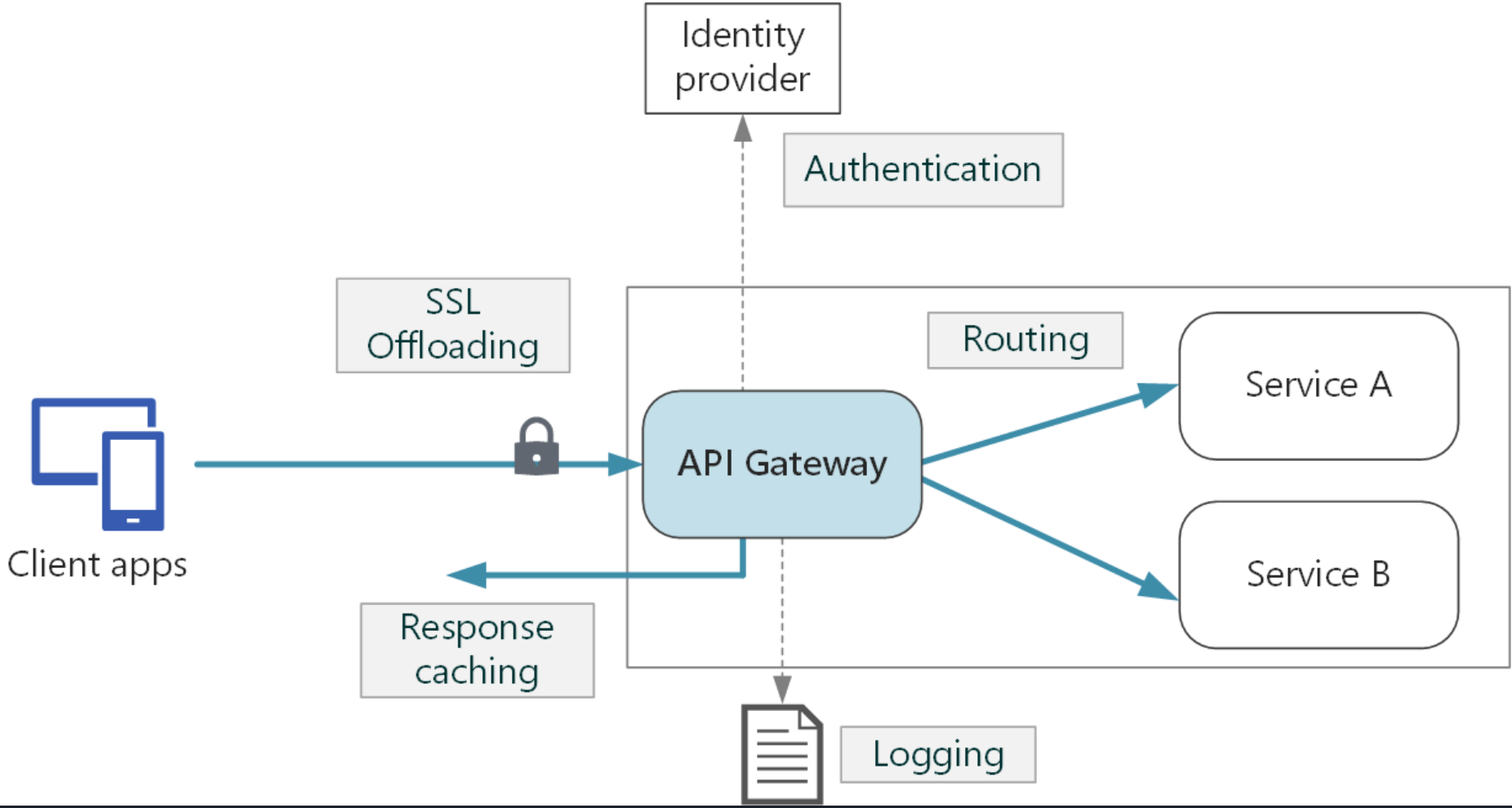
Os padrões estão divididos nestas principais categorias :

- Integration Styles
- Channel Patterns
- Message Construction Patterns
- Routing Patterns
- Transformation Patterns
- Endpoint Patterns
- System Management Patterns


Message Construct.
Message
Command Message
Document Message
Event Message
Request-Reply
Return Address
Correlation Identifier
Message Sequence
Message Expiration
Format Indicator
Message Routing
Pipes-and-Filters
Message Router
Content-based Router
Message Filter
Dynamic Router
Recipient List
Splitter
Aggregator
Resequencer
Composed Msg. Processor
Scatter-Gather
Routing Slip
Process Manager
Message Broker
Message Transformation
Message Translator
Envelope Wrapper
Content Enricher
Content Filter
Claim Check
Normalizer
Canonical Data Model
Endpoint
Message
Channel
Router
Translator
Endpoint
Application A
Application B
Messaging Endpoints
Message Endpoint
Messaging Gateway
Messaging Mapper
Transactional Client
Polling Consumer
Event-driven Consumer
Competing Consumers
Message Dispatcher
Selective Consumer
Durable Subscriber
Idempotent Receiver
Service Activator
Messaging Channels
Message Channel
Point-to-Point Channel
Publish-Subscr. Channel
Datatype Channel
Invalid Message Channel
Dead Letter Channel
Guaranteed Delivery
Channel Adapter
Messaging Bridge
Message Bus
Monitoring
Systems Mgmt.
Control Bus
Detour
Wire Tap
Message History
Message Store
Smart Proxy
Test Message
Channel Purger

API GATEWAY

https://microservices.io/patterns/data/database-per-service.html

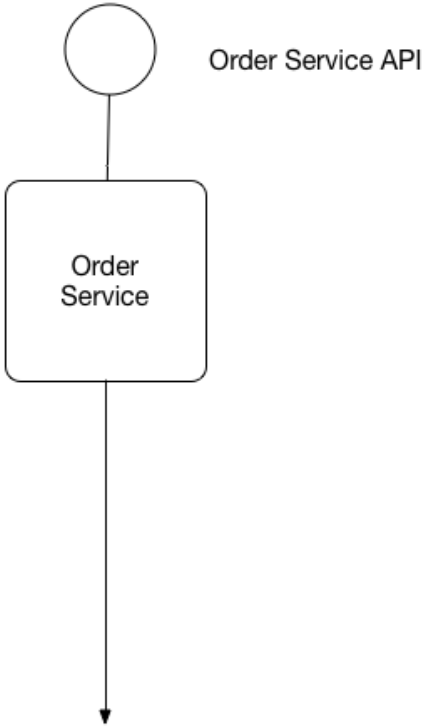

Identity provider
Authentication
SSL Offloading
Routing
Service A
API Gateway
Client apps
Service B
Response caching
Logging

REFERÊNCIA:

https://microservices.io/patterns/apigateway.html

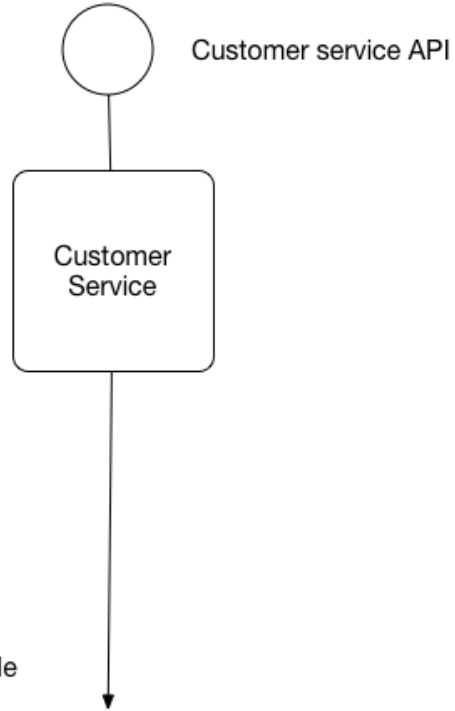
DATABASE PER SERVICE

CENÁRIO ORIGINAL



ORDER table

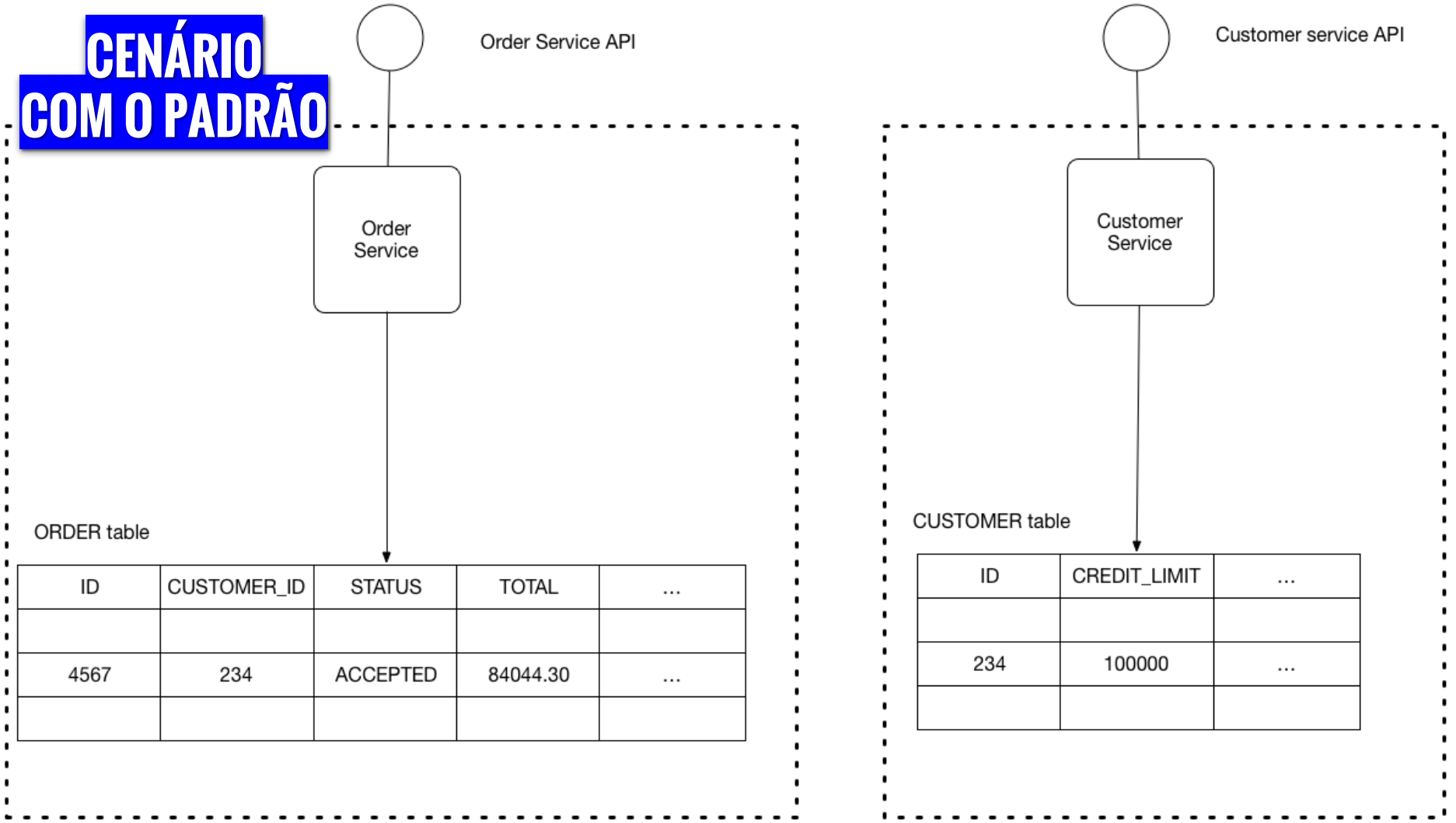
ID	CUSTOMER_ID	STATUS	TOTAL	...
4567	234	ACCEPTED	84044.30	...



CUSTOMER table

ID	CREDIT_LIMIT	...
234	100000	...

CENÁRIO COM O PADRÃO



Database Per Service

Existem diferentes maneiras de manter de forma privada os dados persistentes de um serviço.

Não é obrigatório provisionar um servidor de banco de dados para cada serviço.

Database Per Service

Algumas opções são (seja para o relacional com as tabelas ou para os BDs NoSQL com as collections):

- Tabelas privadas por serviço: cada serviço possui um conjunto de tabelas que devem ser acessadas apenas por esse serviço

- Esquema por serviço: cada serviço tem um esquema de banco de dados privado para esse serviço

- Servidor de banco de dados por serviço: cada serviço tem seu próprio servidor de banco de dados.

REFERÊNCIA:

https://microservices.io/patterns/data/database-per-service.html

Ao usar o Padrão **Database Per Service**, muito provavelmente será necessário implementar o **SAGA PATTERN.**

SAGA PATTERN

SAGAS

Hector Garcia-Molina
Kenneth Salem

Department of Computer Science
Princeton University
Princeton, N J 08544

Abstract

Long lived transactions (LLTs) hold on to database resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions To alleviate these problems we propose the notion of a saga A LLT is a saga if it can be written as a sequence of transactions that can be interleaved with other transactions The database management system guarantees that either all the transactions in a saga are successfully completed or compensating transactions are run to amend a partial execution Both the concept of saga and its implementation are relatively simple, but they have the potential to improve performance significantly We analyze the various implementation issues related to sagas, including how they can be run on an existing system that does not directly support them We also discuss techniques for database and LLT design that make it feasible to break up LLTs into sagas

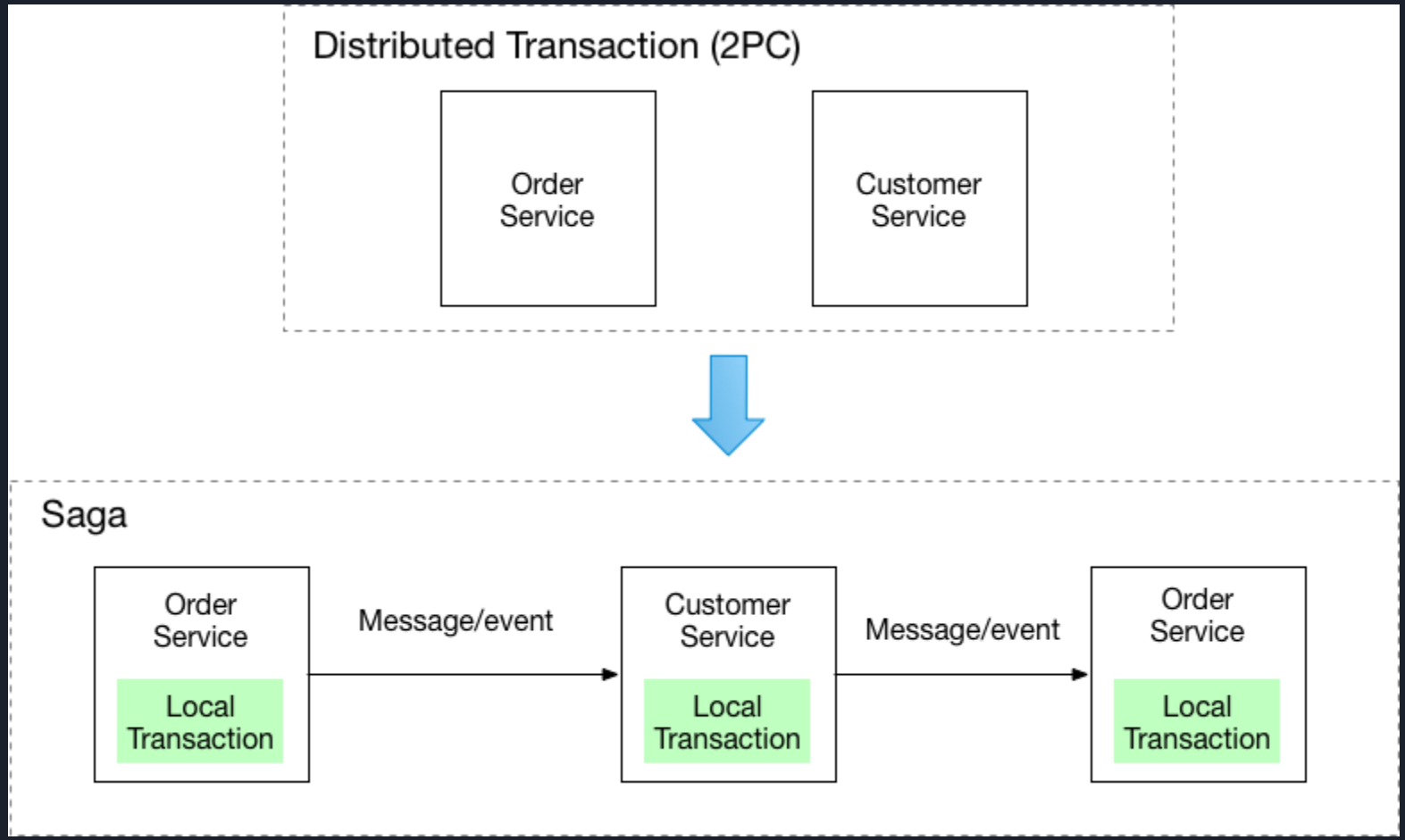
1. INTRODUCTION

As its name indicates, a *long lived transaction* is a transaction whose execution, even without interference from other transactions, takes a substantial amount of time, possibly on the order of hours or days A long lived transaction, or LLT, has a long duration compared to the majority of other transactions either because it accesses many database objects, it has lengthy computations, it pauses for inputs from the users, or a combination of these factors Examples of LLTs are transactions to produce monthly account statements at a bank, transactions to process claims at an insurance company, and transactions to collect statistics over an entire database [Gray81a]

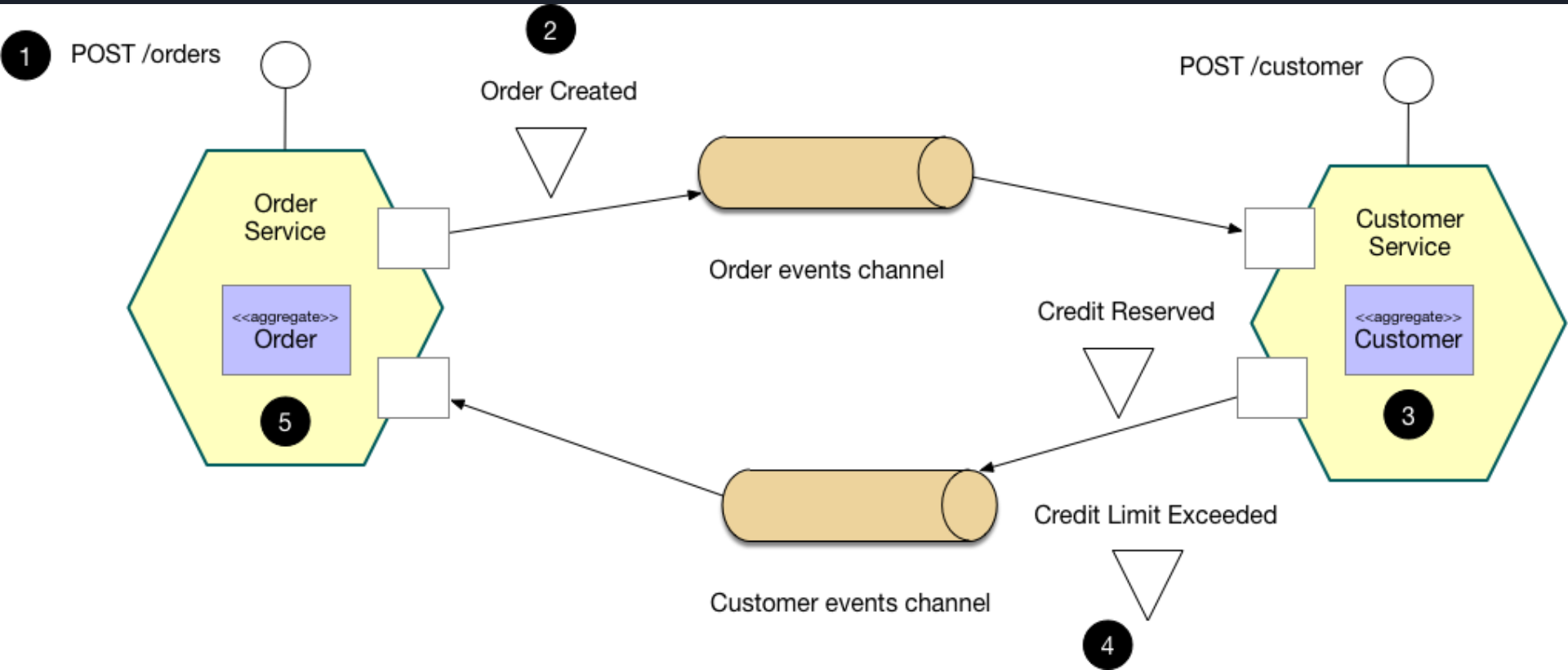
In most cases, LLTs present serious performance problems Since they are transactions, the system must execute them as atomic actions, thus preserving the consistency of the database [Date81a, Ullm82a] To make a transaction atomic, the system usually locks the objects accessed by the transaction until it commits, and this typically occurs at the end of the transaction As a consequence, other transactions wishing to access the LLT's objects suffer a long locking delay If LLTs are long because they access many database objects then other transactions are likely to suffer from an increased blocking rate as well, i e they are more likely to conflict with an LLT than with a shorter transaction

Furthermore, the transaction abort rate can also be increased by LLTs As discussed in [Gray81b], the frequency of deadlock is very sensitive to the "size" of transactions, that is, to how many objects transactions access (In the analysis of [Gray81b] the deadlock frequency grows with the fourth power of the transaction size) Hence, since LLTs access many objects, they may cause many deadlocks, and correspondingly, many abortions From the point of view of system crashes, LLTs have a higher probability of encountering a failure (because of their duration), and are thus more likely to encounter yet more delays and more likely to be aborted themselves

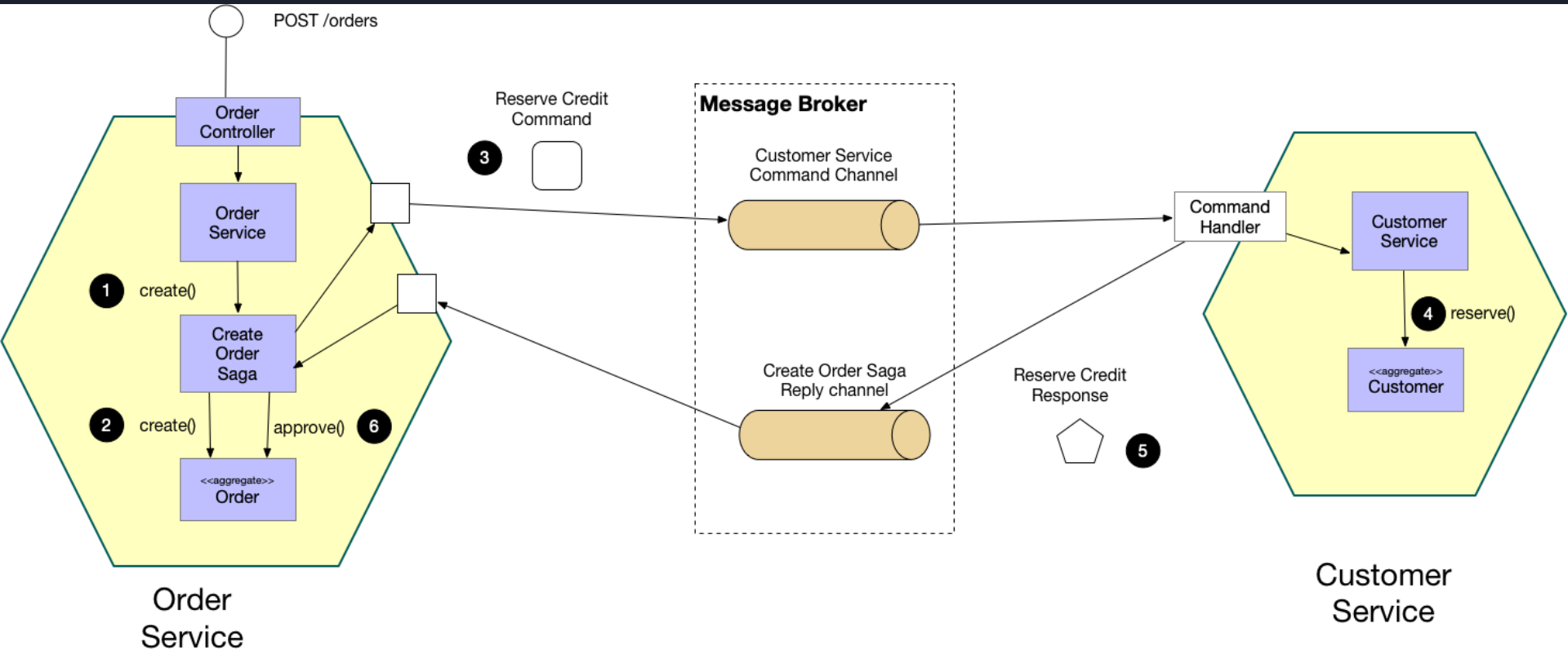
https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf


Distributed Transaction (2PC)
Order Service
Customer Service
Saga
Order Service
Local Transaction
Message/event
Customer Service
Local Transaction
Message/event
Order Service
Local Transaction

COREOGRAPHY BASED



ORCHESTRATION BASED



SAGA PATTERN

Benefícios:
Permite que um aplicativo mantenha a consistência dos dados em vários serviços sem usar transações distribuídas

Desvantagens:
O modelo de programação é mais complexo. Por exemplo, um desenvolvedor deve projetar transações compensatórias que desfazem explicitamente as alterações feitas anteriormente em uma saga.

SAGA PATTERN

Existem também os seguintes problemas para resolver:

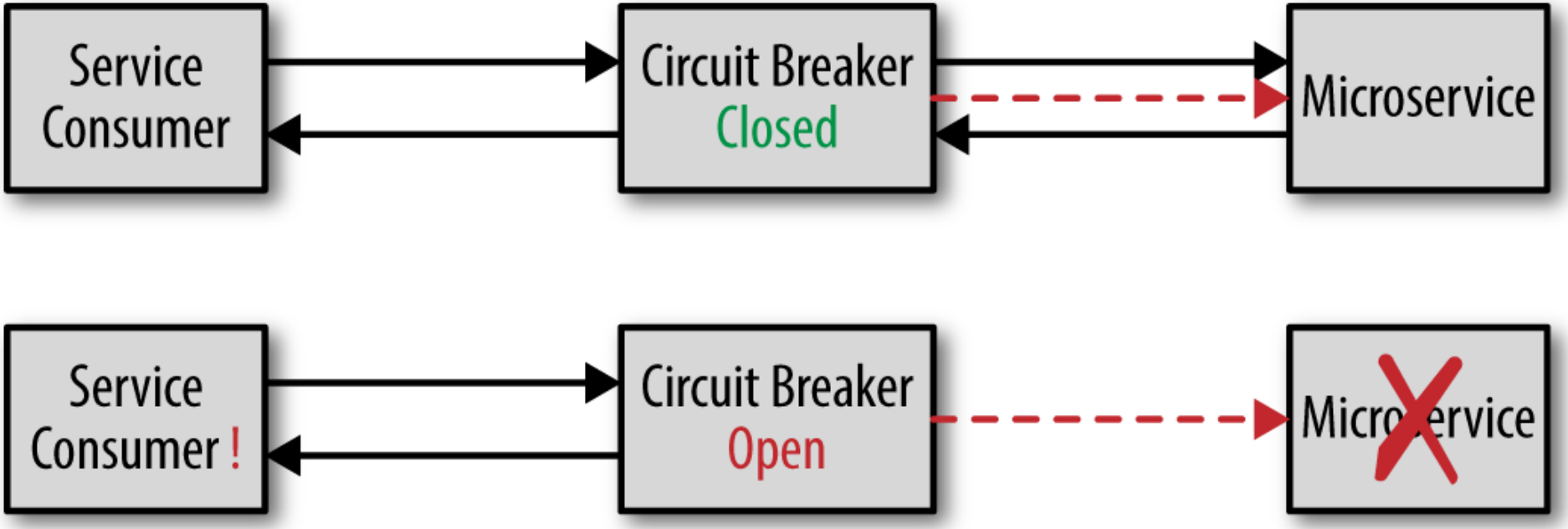
Para ser confiável, um serviço deve atualizar atomicamente seu banco de dados E publicar uma mensagem / evento.

Ele não pode usar o mecanismo tradicional de uma transação distribuída que abrange o banco de dados e o intermediário de mensagens.

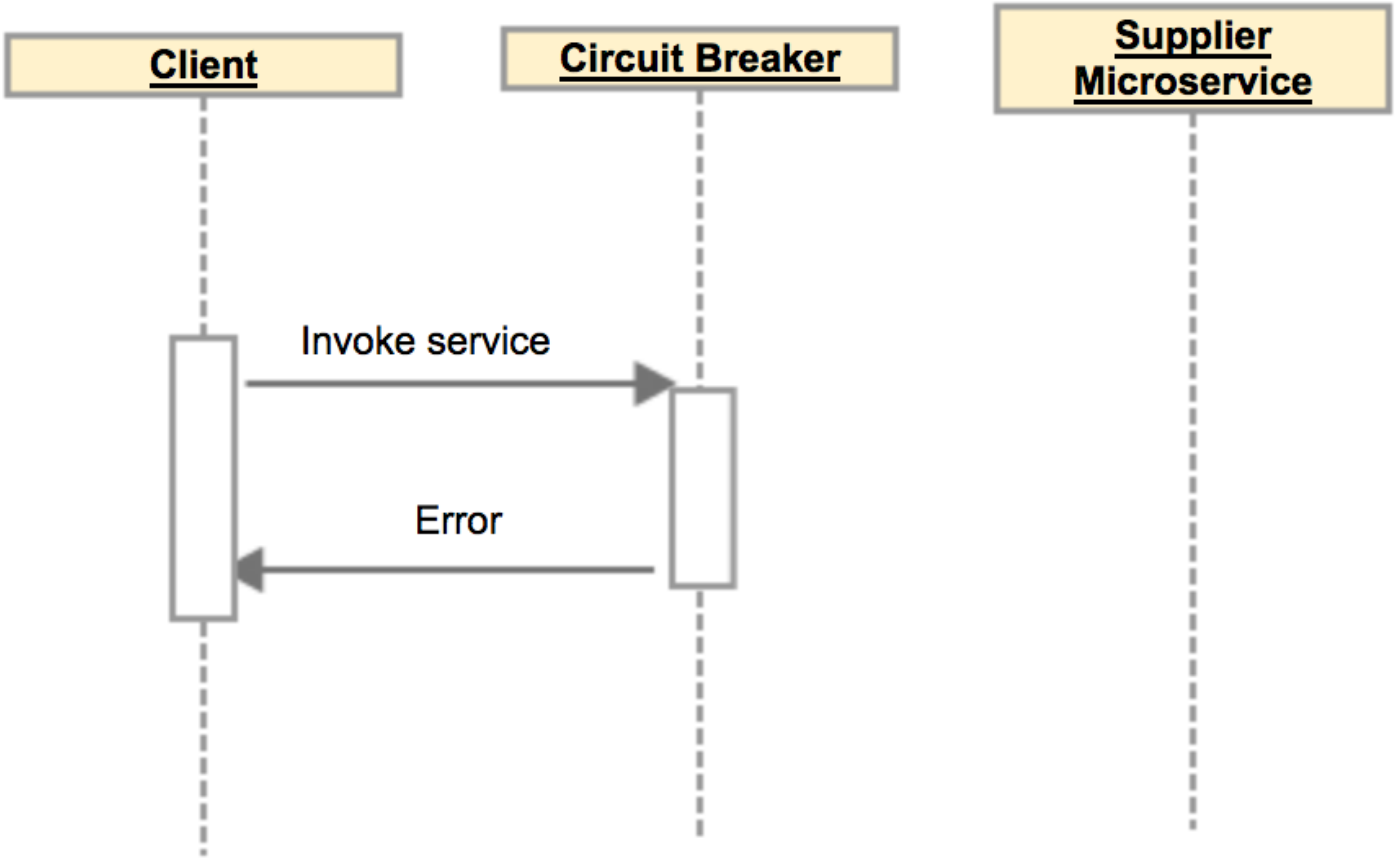
REFERÊNCIA:

https://microservices.io/patterns/data/saga.html

CIRCUIT BREAKER


Service Consumer
Circuit Breaker Closed
Microservice
Service Consumer !
Circuit Breaker Open
Microservice

OPEN state

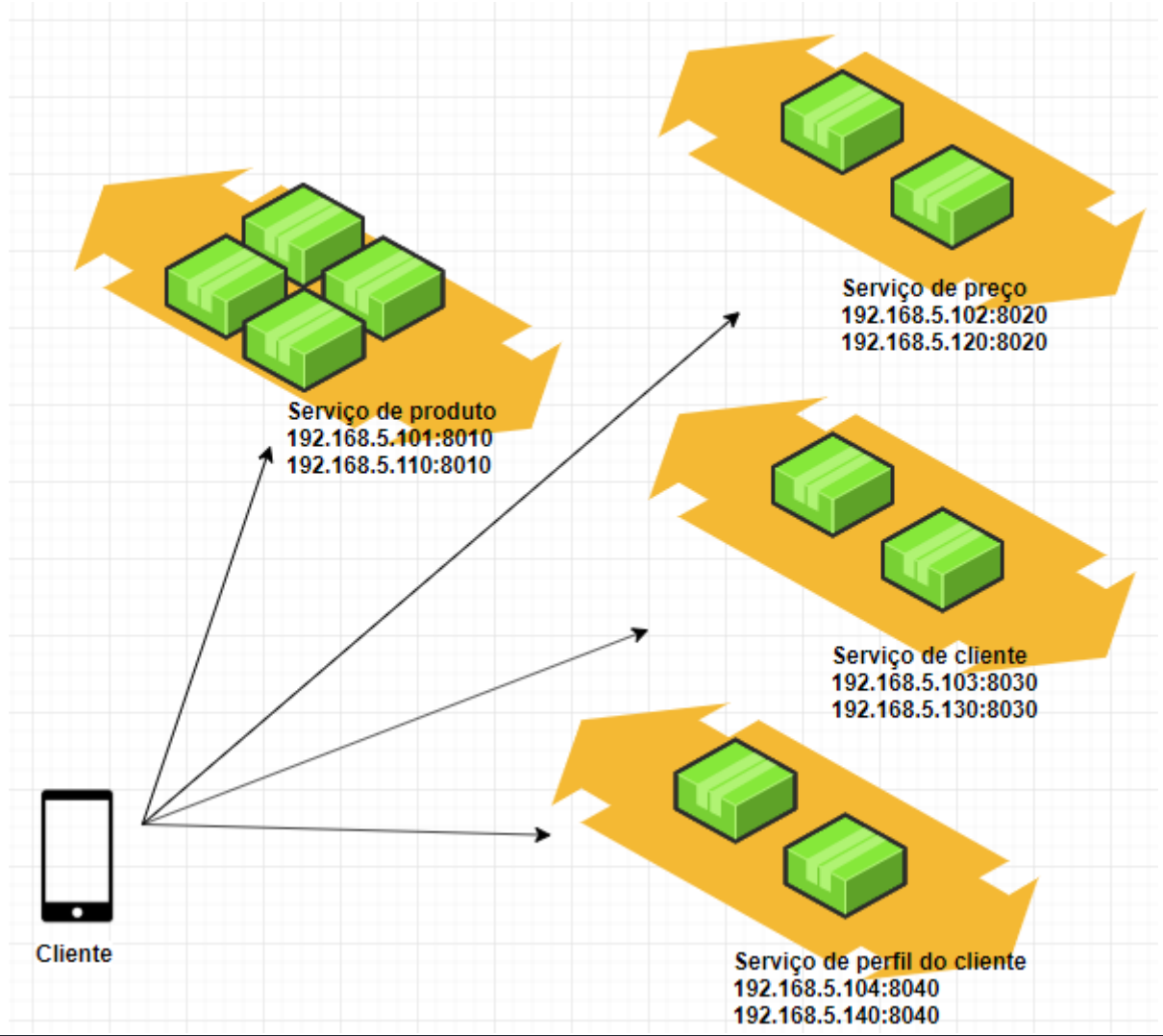

Client
Circuit Breaker
Supplier Microservice
Invoke service
Error

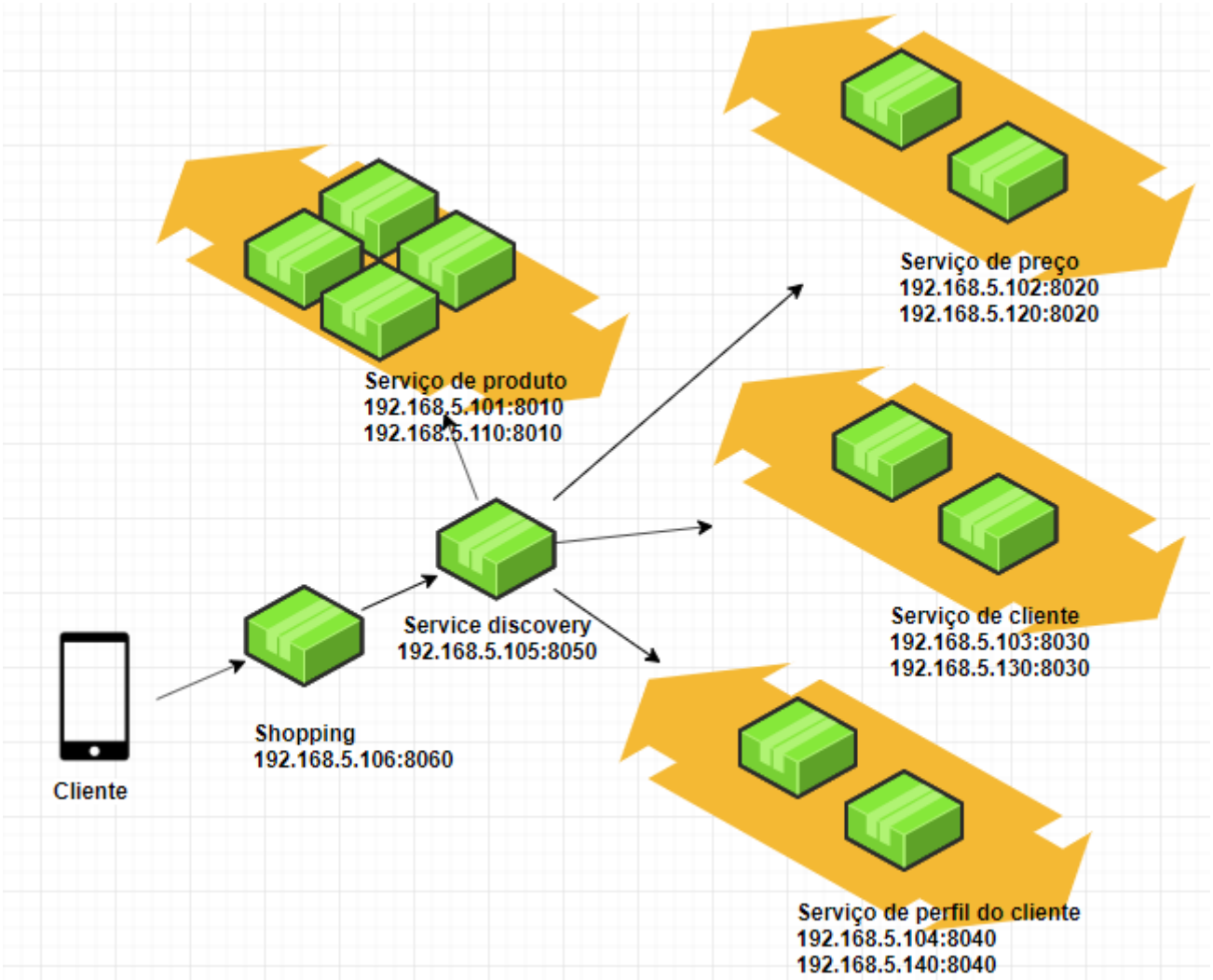
REFERÊNCIAS:

https://microservices.io/patterns/reliability/circuit-breaker.html

https://pypi.org/project/circuitbreaker/

SERVICE DISCOVERY

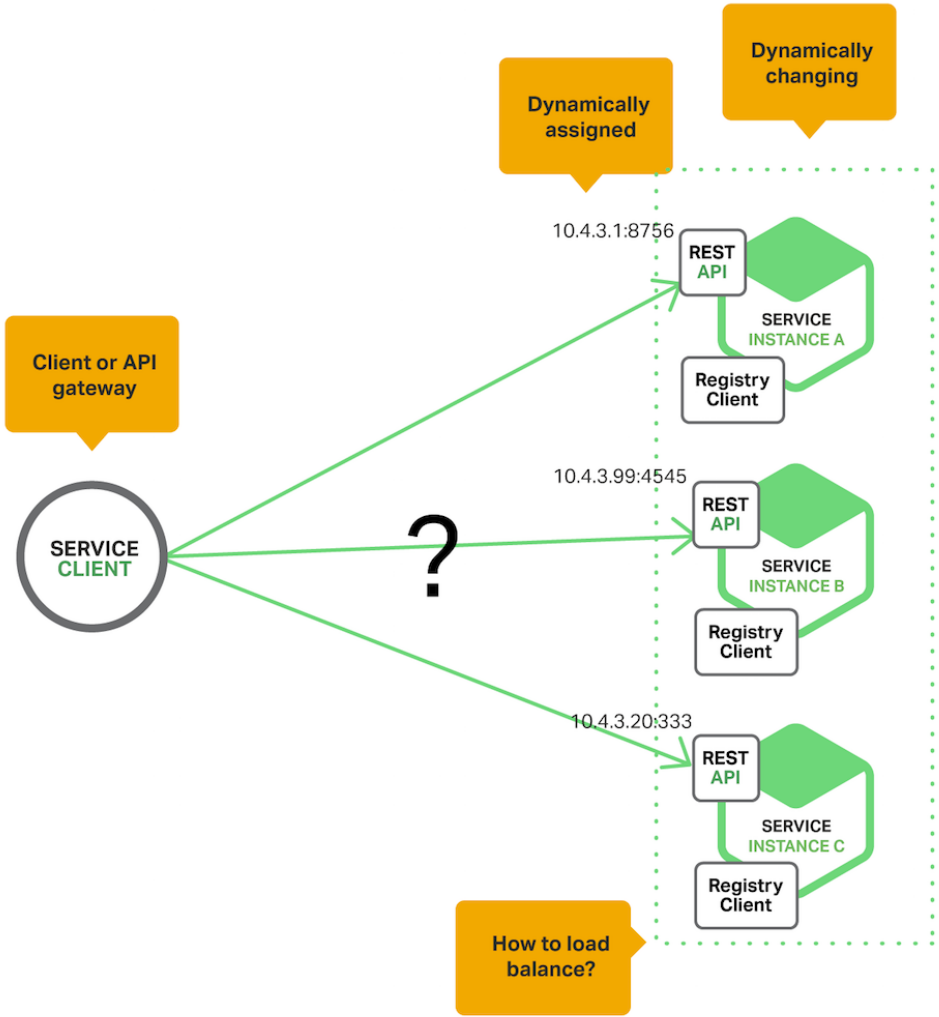

Serviço de produto
192.168.5.101:8010
192.168.5.110:8010
Serviço de preço
192.168.5.102:8020
192.168.5.120:8020
Serviço de cliente
192.168.5.103:8030
192.168.5.130:8030
Serviço de perfil do cliente
192.168.5.104:8040
192.168.5.140:8040
Cliente

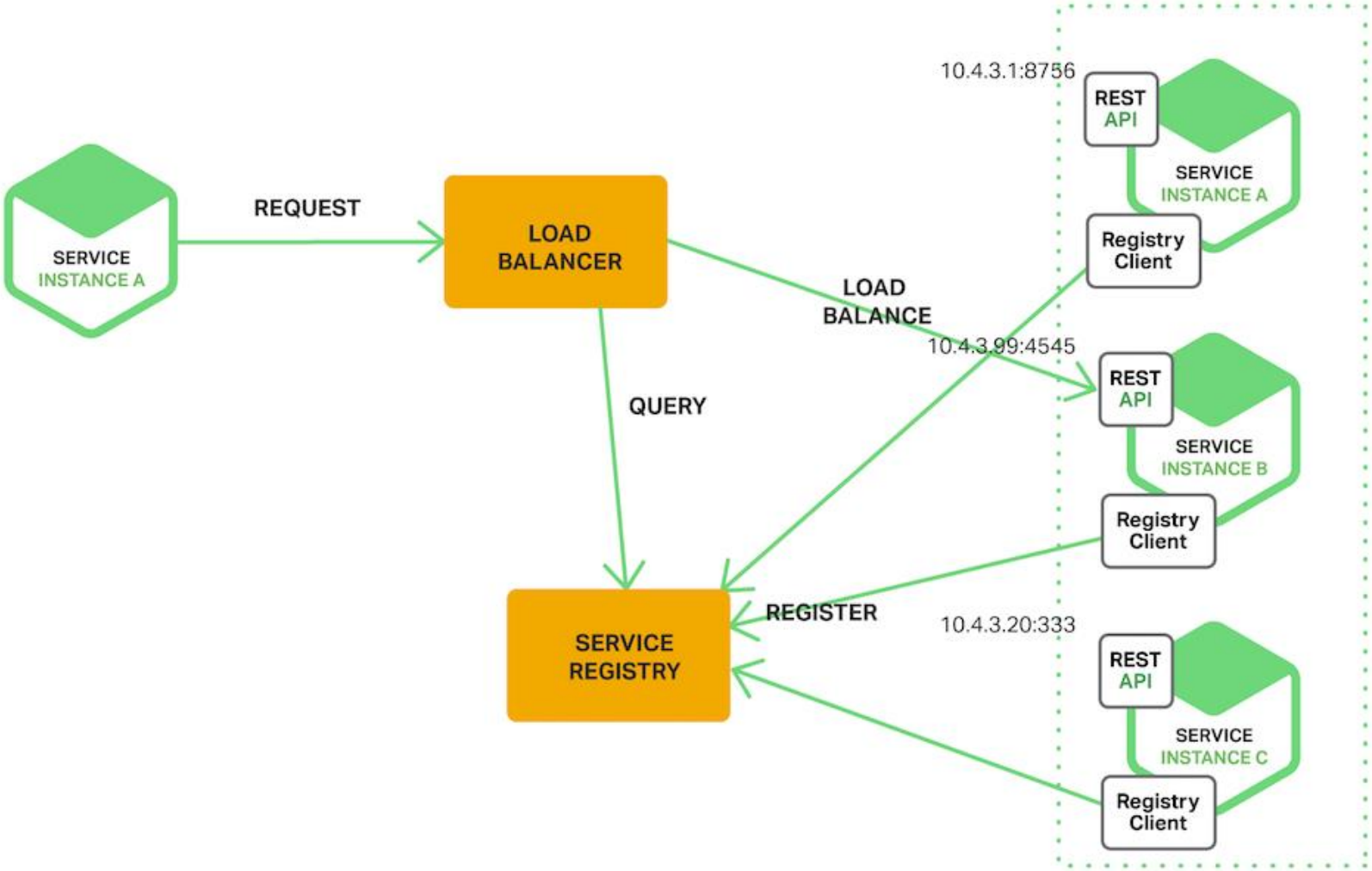

Serviço de produto
192.168.5.101:8010
192.168.5.110:8010
Serviço de preço
192.168.5.102:8020
192.168.5.120:8020
Service discovery
192.168.5.105:8050
Serviço de cliente
192.168.5.103:8030
192.168.5.130:8030
Shopping
192.168.5.106:8060
Cliente
Serviço de perfil do cliente
192.168.5.104:8040
192.168.5.140:8040

REFERÊNCIA:

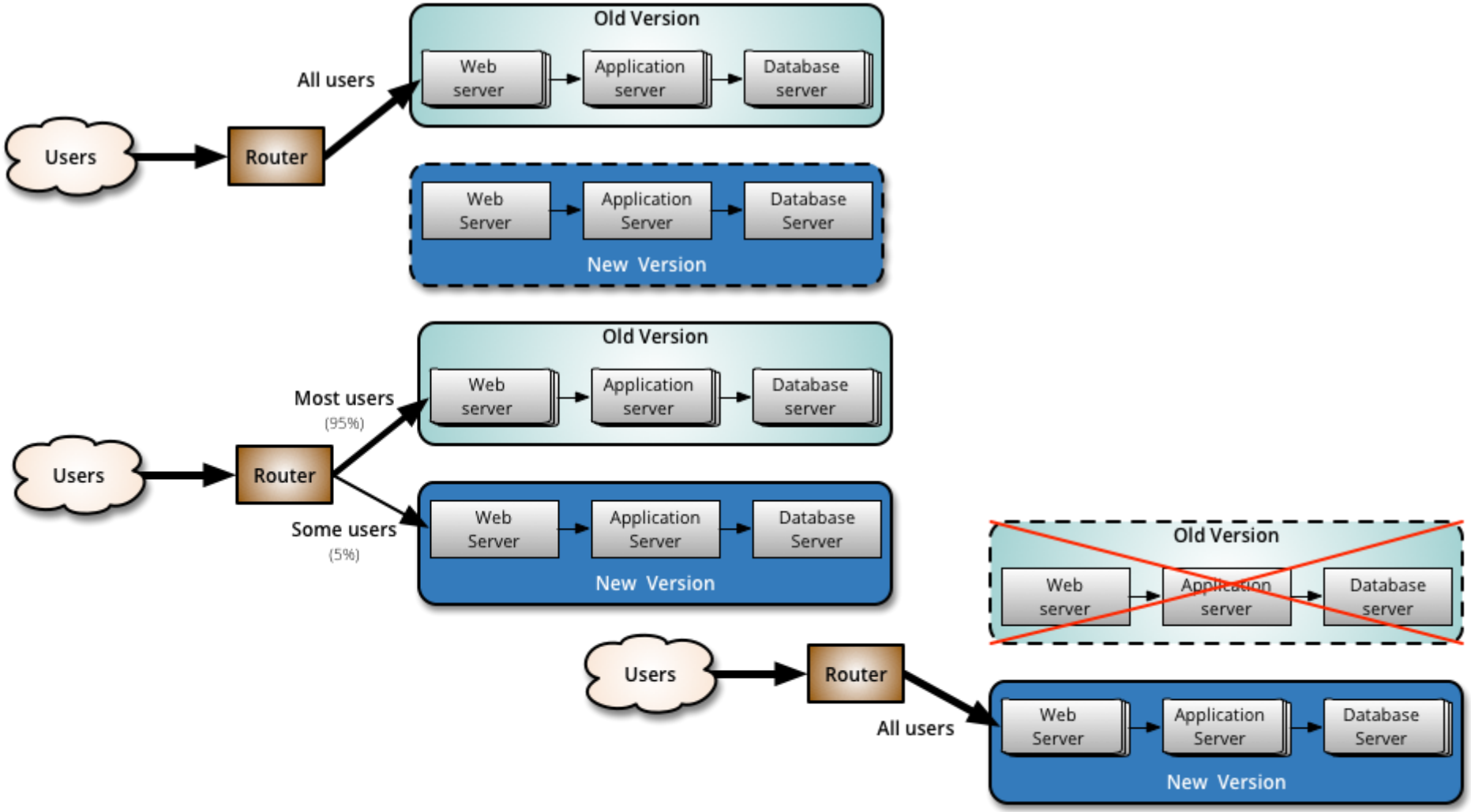
https://medium.com/codigorefinado/padrões-de-microserviços-service-discovery-309c84422446

LOAD BALACING COM MICROSERVICES


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?


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

CANARY DEPLOYMENT


Old Version
Web server
Application server
Database server
All users
Users
Router
Web Server
Application Server
Database Server
New Version
Old Version
Web server
Application server
Database server
Most users
(95%)
Users
Router
Some users
(5%)
Web Server
Application Server
Database Server
New Version
Old Version
Web server
Application server
Database server
Users
Router
All users
Web Server
Application Server
Database Server
New Version

REFERÊNCIA:

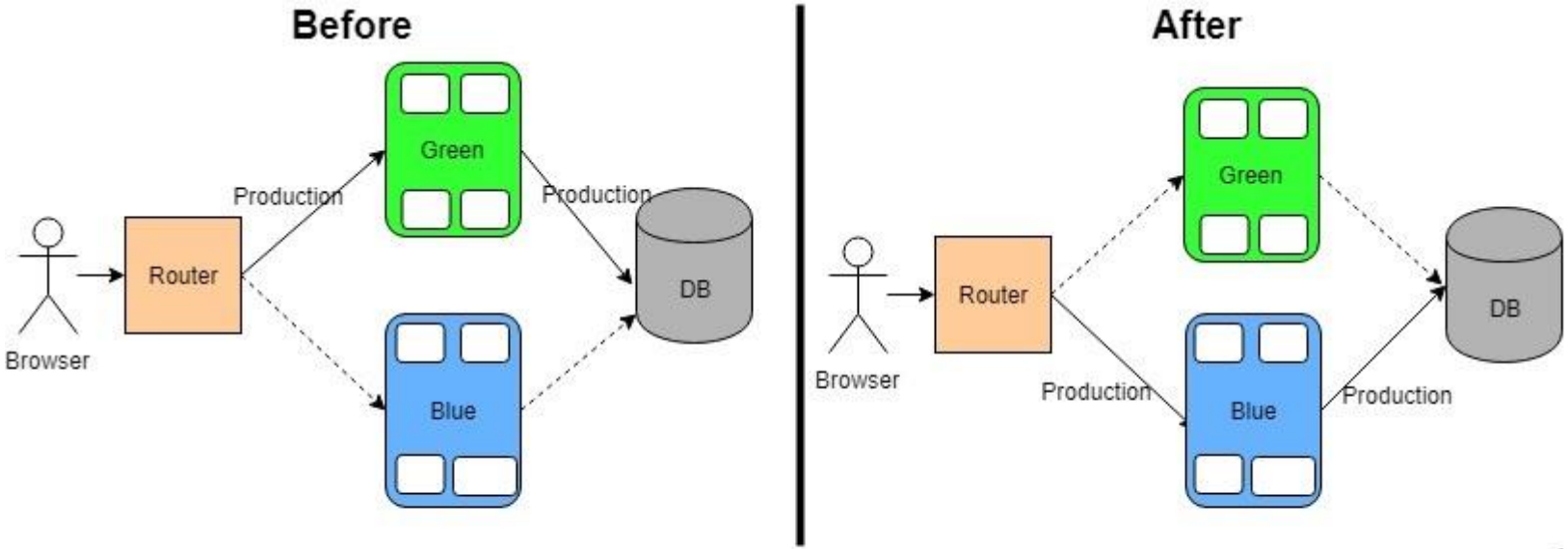
https://martinfowler.com/bliki/CanaryRelease.html

BLUEGREEN DEPLOYMENT

BLUE GREEN DEPLOYMENT

Reduzir os atrasos que surgem entre a conclusão do software e a agregação de valor que ele gera.

- Precisamos ter dois ambientes de produção, o mais idênticos possível (blue e green).
- A qualquer momento, um deles (seja o azul ou o verde), está ativo.
- Ao preparar uma nova versão do seu software, realizamos o deploy de teste no ambiente verde.
- Depois que o software estiver funcionando no ambiente verde, alternamos o roteador para que todas as solicitações recebidas passem para o ambiente verde - o azul fica ocioso.
- Usualmente implementado com um load balancer.

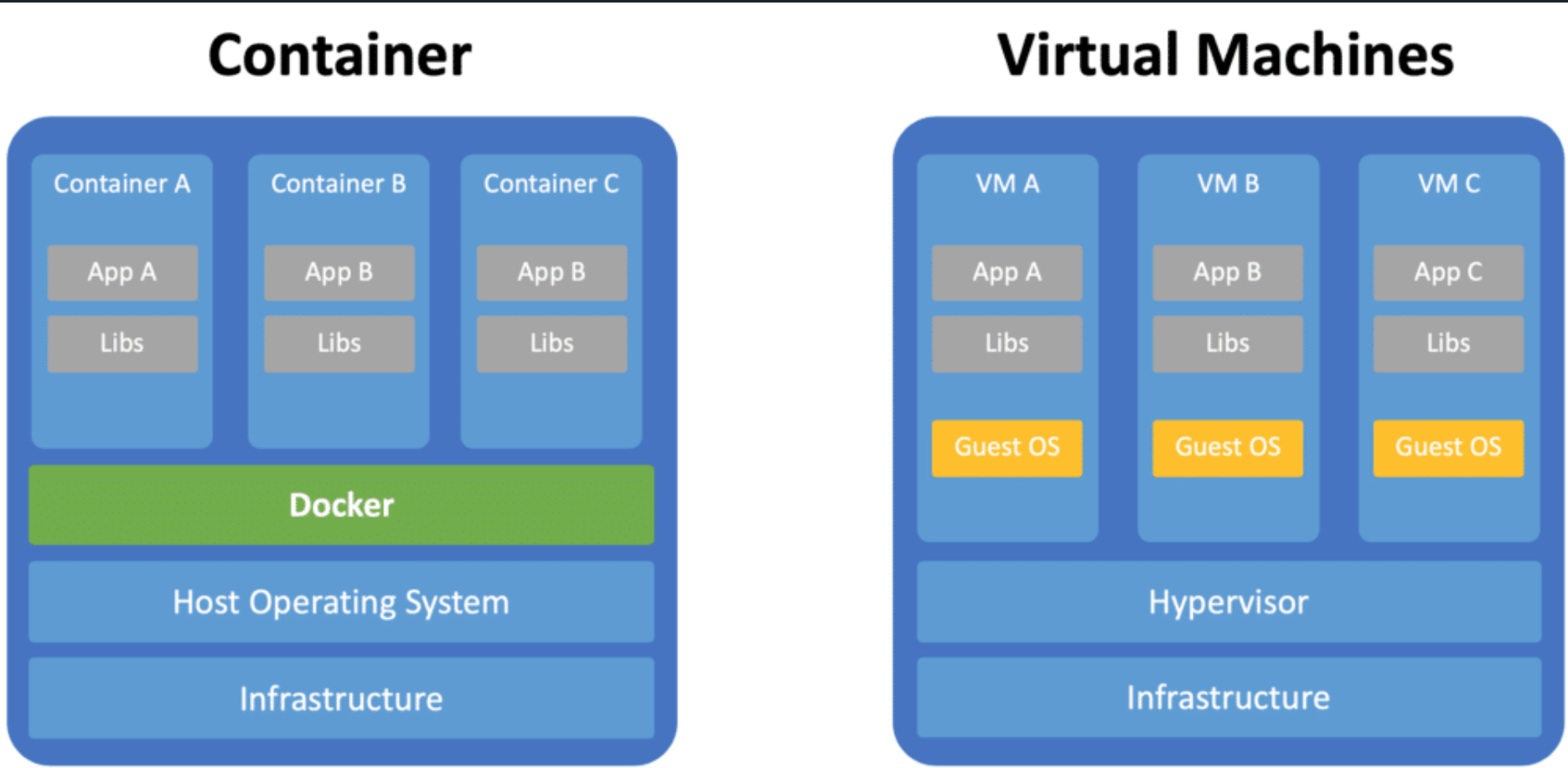

Before
Green
Production
Production
Router
DB
Browser
Blue
After
Green
Router
DB
Browser
Production
Blue
Production

REFERÊNCIA:

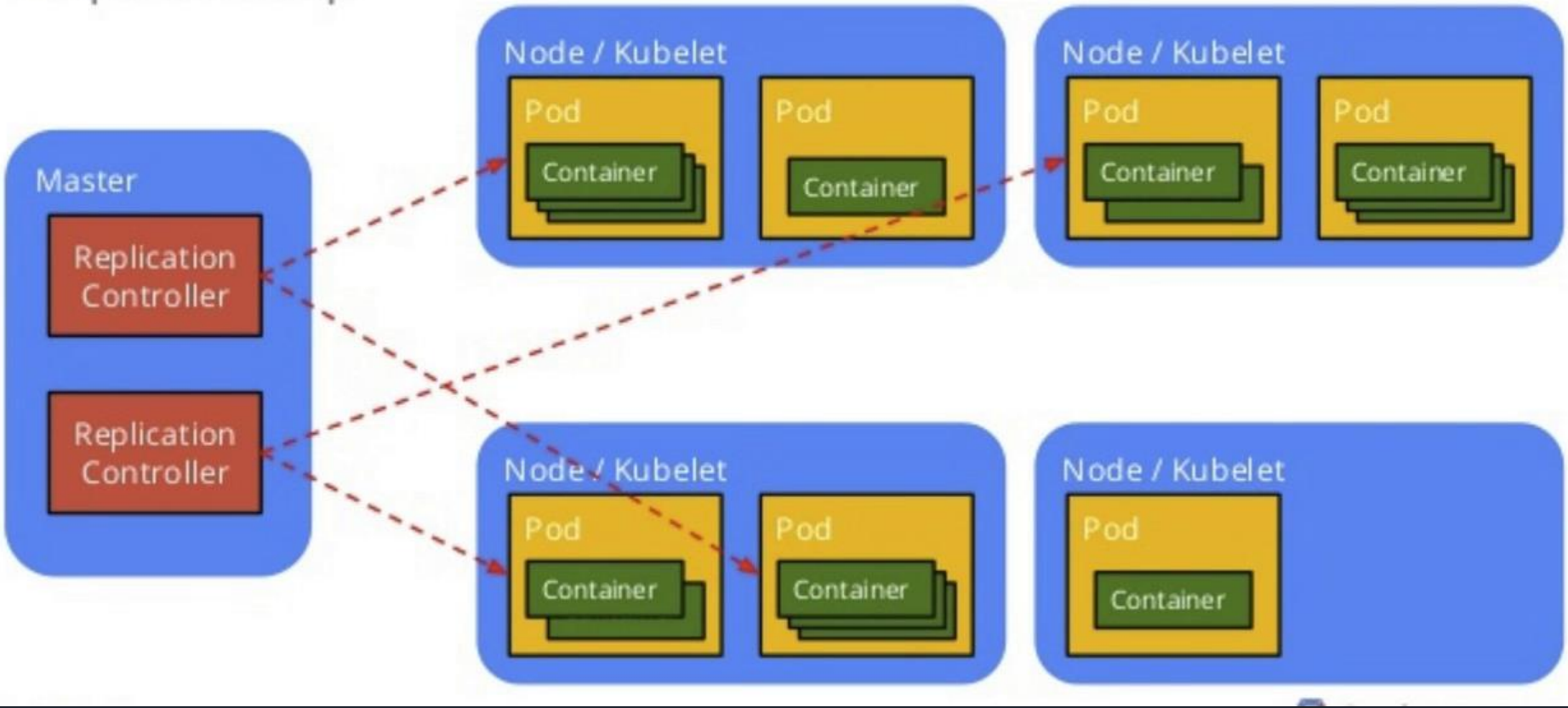
https://martinfowler.com/bliki/BlueGreenDeployment.html

COMO COLOCAR UM MICROSERVICE EM PRODUÇÃO?

CONTAINERS VS MICROSERVICES


Container
Container A
App A
Libs
Container B
App B
Libs
Container C
App B
Libs
Docker
Host Operating System
Infrastructure
Virtual Machines
VM A
App A
Libs
Guest OS
VM B
App B
Libs
Guest OS
VM C
App C
Libs
Guest OS
Hypervisor
Infrastructure

A quick recap



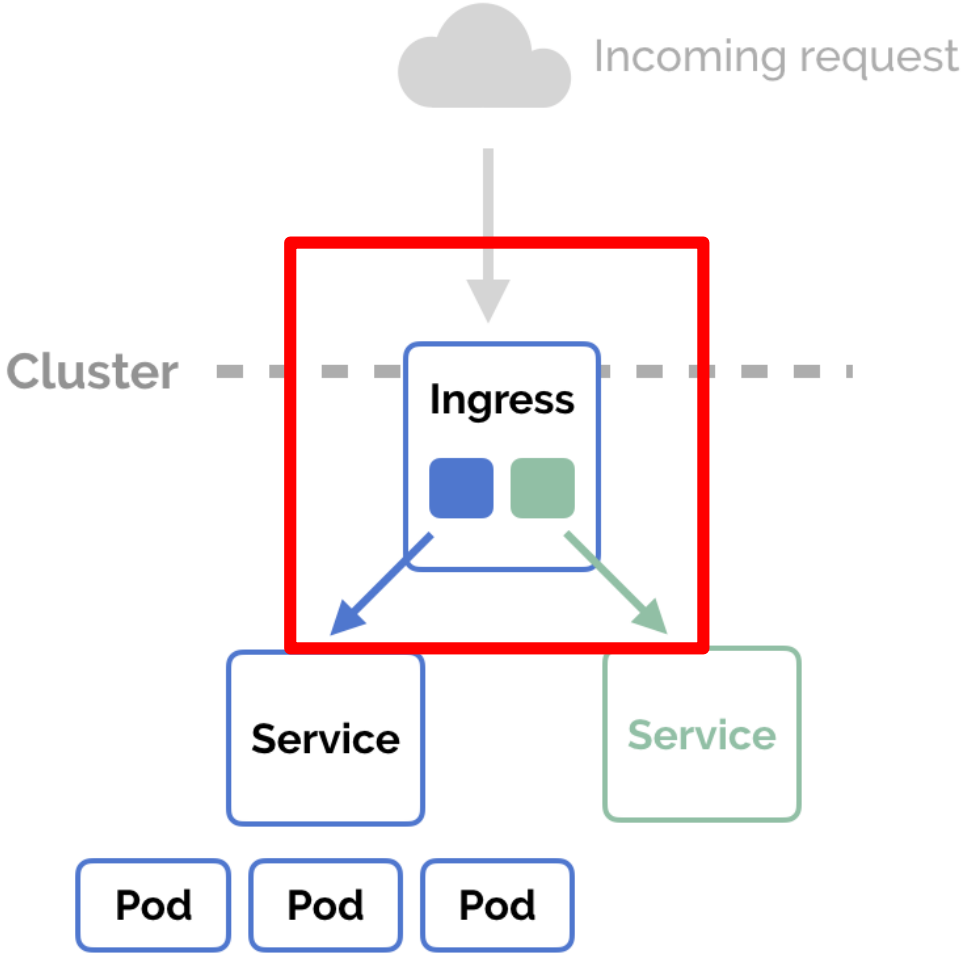
PRINCIPAIS PLAYERS (K8S):

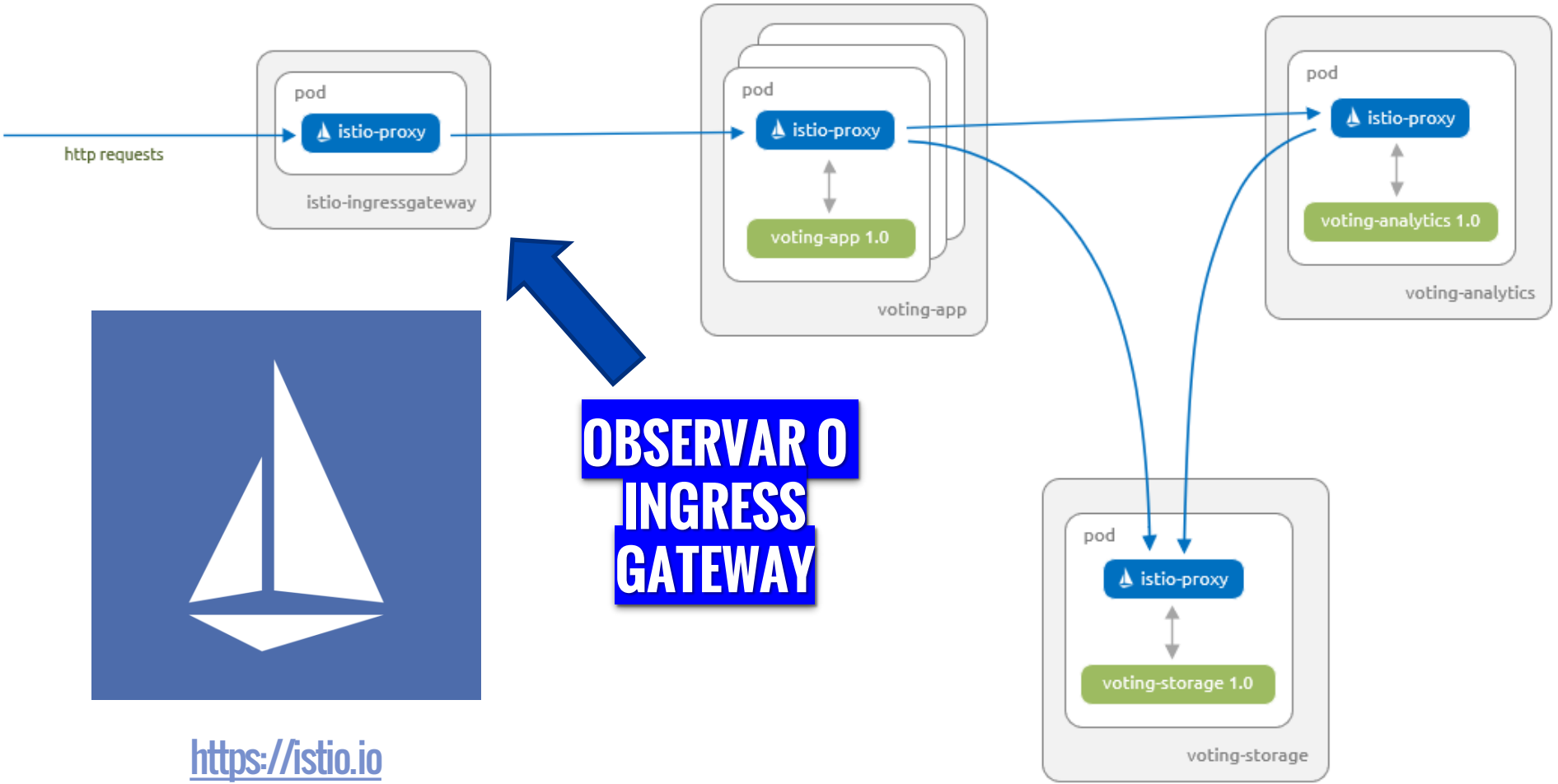
https://cloud.google.com/kubernetes-engine/

https://aws.amazon.com/pt/eks/

https://azure.microsoft.com/pt-br/free/kubernetes-service/

SERVICE MESH COM ISTIO


Incoming request
Cluster
Ingress
Service
Service
Pod
Pod
Pod


http requests
pod
istio-proxy
istio-ingressgateway
pod
istio-proxy
voting-app 1.0
voting-app
pod
istio-proxy
voting-analytics 1.0
voting-analytics
pod
istio-proxy
voting-storage 1.0
voting-storage
OBSERVAR O INGRESS GATEWAY
https://istio.io

REFERÊNCIA:

https://istio.io


Dúvidas?


Thank
you