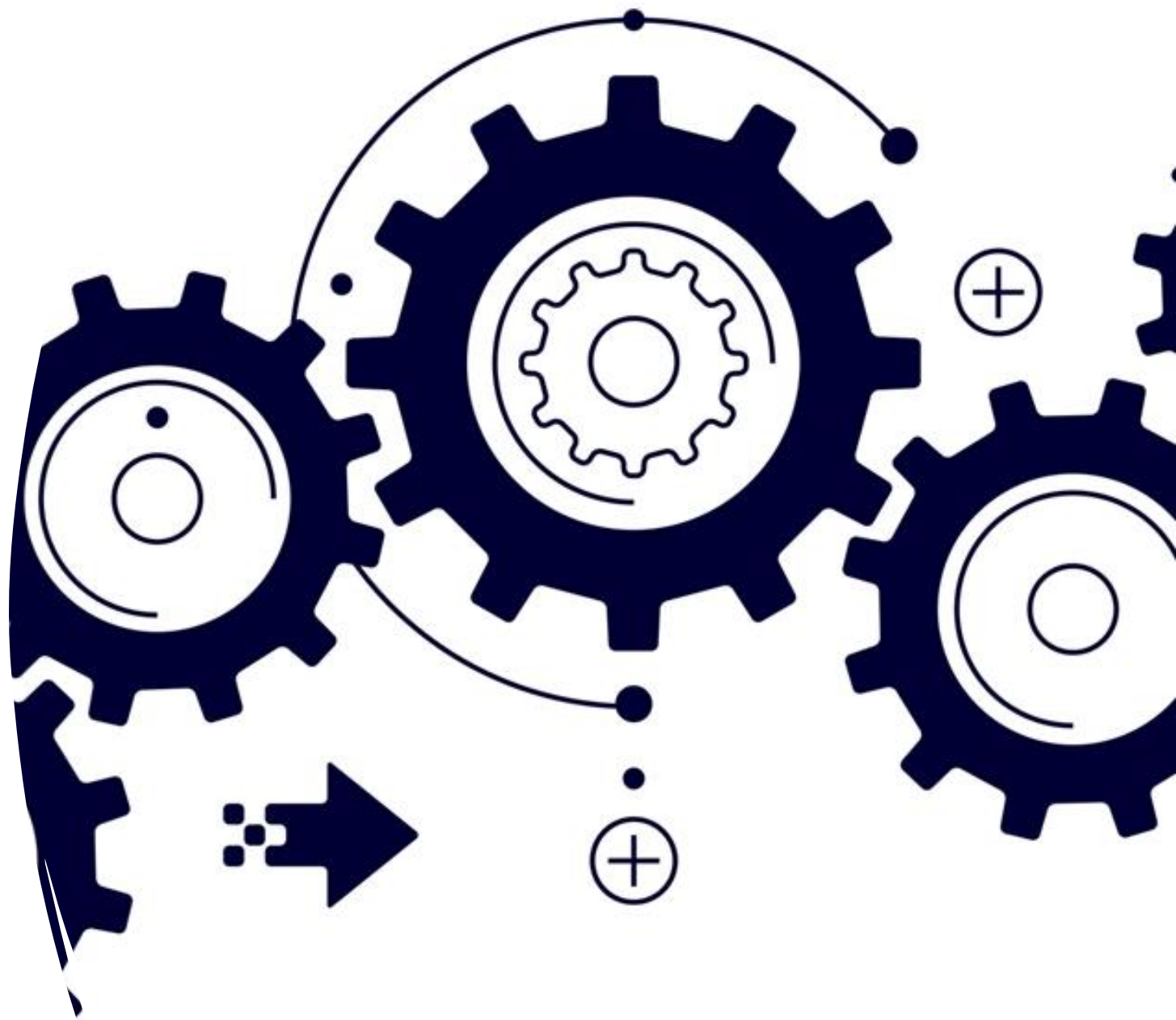


Arquitetura de Microserviços

Arquitetura de Sistemas .Net
FIAP



Sobre

Carlos Fernando Sylverio

- Técnico em Eletrônica
- Bacharel em Sistemas de Informação
- MBA em Gestão de Projetos
- Certificado Scrum Master
- 20 anos atuando com desenvolvimento
- Liderando times de dev desde 2014
- Apaixonado por Arquitetura de Software e Orientação a Objetos



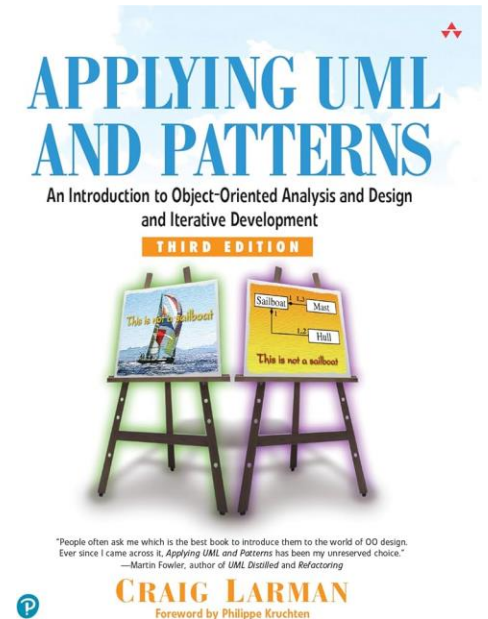
<https://www.linkedin.com/in/csylverio/>



Revisitando conceitos de O. O.

GRASP - General Responsibility Assignment Software Patterns

- Conjunto de padrões
- Guia a atribuição de responsabilidades em objetos
- **Craig Larman.** Applying UML and Patterns



Revisitando conceitos de O. O.

Princípio	Descrição
Creator	Define quem é responsável por criar uma nova instância de uma classe
Information Expert	Atribui a responsabilidade a uma classe que possui a informação necessária para cumprir a responsabilidade
Low Coupling	Minimiza a dependência entre módulos (classes, componentes, etc.)
High Cohesion	Garante que uma classe tenha responsabilidades fortemente relacionadas e focadas
Controller	Objeto que coordena e controla o fluxo de eventos ou operações de um caso de uso

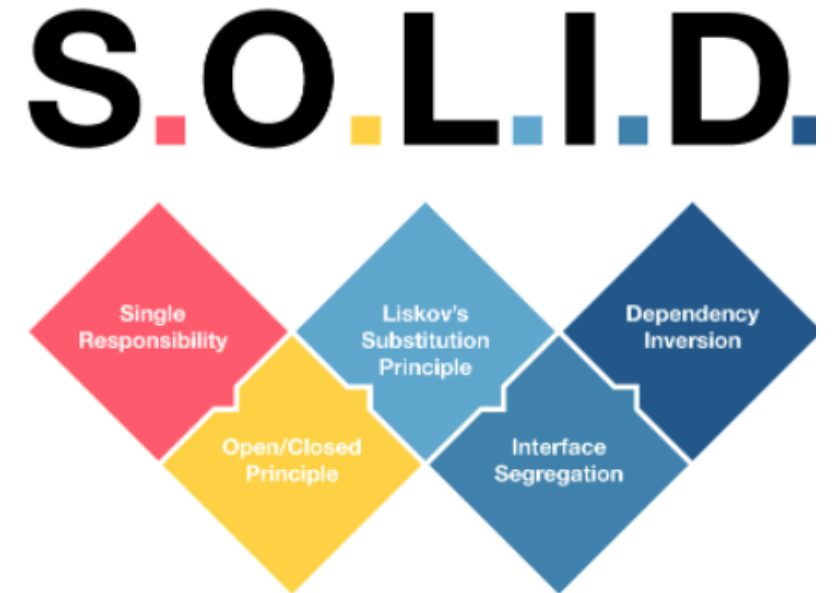
Revisitando conceitos de O. O.

Princípio	Descrição
Polymorphism	Operações similares sejam realizadas de maneiras diferentes (Implements e Extends)
Pure Fabrication	Não representa um conceito do domínio, mas que é criada apenas para atingir baixo acoplamento, alta coesão e/ou reutilização (Ex: Pattern Repository)
Indirection	Intermediário para desacoplar classes ou componentes para evitar dependências diretas (Ex: Pattern Mediator e Observer)
Protected Variations	Protege partes do sistema contra variações em outras partes, utilizando interfaces, abstrações ou outros mecanismos que limitem o impacto de mudanças.

Revisitando conceitos de O. O.

SOLID

- 5 princípios fundamentais de design orientado a objetos
- Criação de software modular e de fácil manutenção
- **Robert C. Martin** (também conhecido como Uncle Bob)



Revisitando conceitos de O. O.

Princípio	Descrição
Single Responsibility Principle	Uma classe deve ter apenas uma razão para mudar, ou seja, deve ter apenas uma responsabilidade ou propósito.
Open/Closed Principle	Entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação.
Liskov Substitution Principle	Objetos de uma classe derivada devem poder substituir objetos de sua classe base sem alterar a corretude do programa.
Interface Segregation Principle	Uma classe não deve ser forçada a depender de métodos que não utiliza. Interfaces devem ser específicas para o cliente e não genéricas
Dependency Inversion Principle	Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações (interfaces ou classes abstratas).

Revisitando conceitos de O. O.

Resumo Comparativo

GRASP foca na atribuição de responsabilidades aos objetos e na manutenção da coesão e do baixo acoplamento.

SOLID fornece diretrizes para a criação de software modular, facilmente extensível e de manutenção simples, com base em princípios de design orientado a objetos.



Ambos os conjuntos de princípios são essenciais para arquitetos e desenvolvedores de software que desejam criar sistemas robustos e flexíveis.

Arquitetura de Software X Design de Software

- **"Arquitetura"** é usado no contexto de algo em um nível mais alto e que independe dos detalhes dos níveis mais baixos.
- **"Design"** parece muitas vezes sugerir as estruturas e decisões de níveis mais baixos.

Arquitetura

- Componentes
- Interações
- Visão sistêmica

Design

- Implementação de componente arquitetônico
- Algoritmos
- Estrutura de dados

Robert C. Martin, Clean Code

Arquitetura de Software X Design de Software



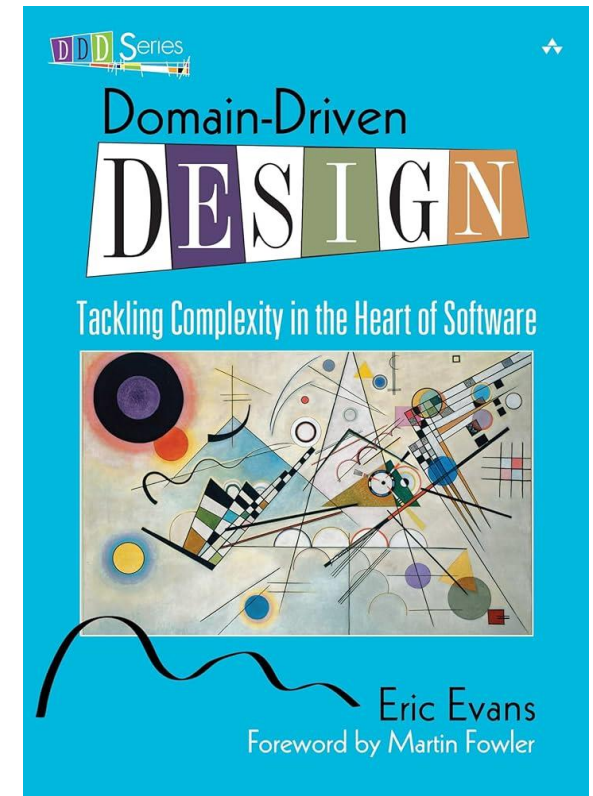
*"Detalhes de baixo nível e estruturas de alto nível
são partes do mesmo todo. Juntos, formam um
tecido contínuo que define e molda o sistema."*

Robert C. Martin, Clean Architecture

Revisitando conceitos de DDD

Domain-Driven Design (DDD)

- Enfatiza a **importância** de alinhar a implementação de software com o domínio de negócio que ele busca solucionar
- Promover um entendimento profundo dos problemas e requisitos do domínio
- Eric Evans , Domain-Driven Design: Tackling Complexity in the Heart of Software

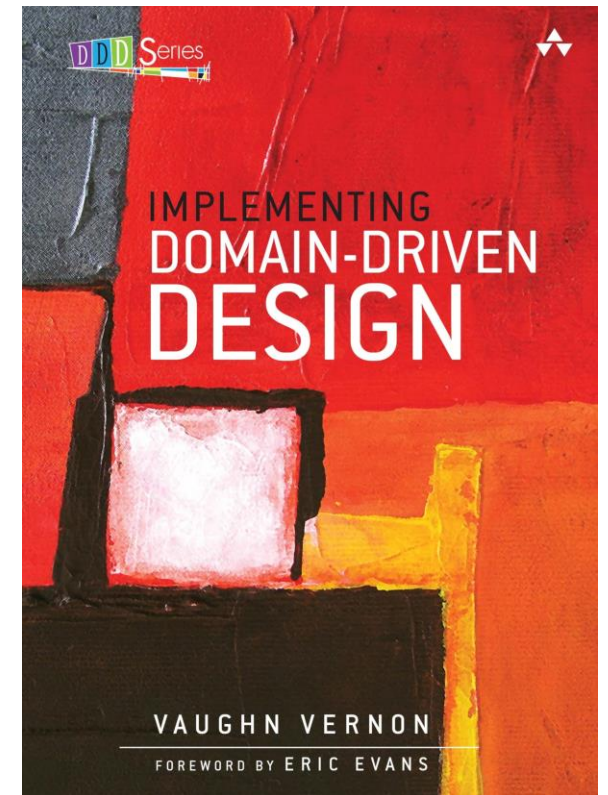


Revisitando conceitos de DDD

O que é um modelo de domínio?

- Modelo de software do domínio de negócio muito específico
- Objetos com dados e comportamentos com significado literal e preciso do negócio
- Modelo único elaborado no centro de um subsistema
- Nunca modela todos os negócios da empresa como um único grande modelo de domínio

Vaughn Vernon



Revisitando conceitos de DDD

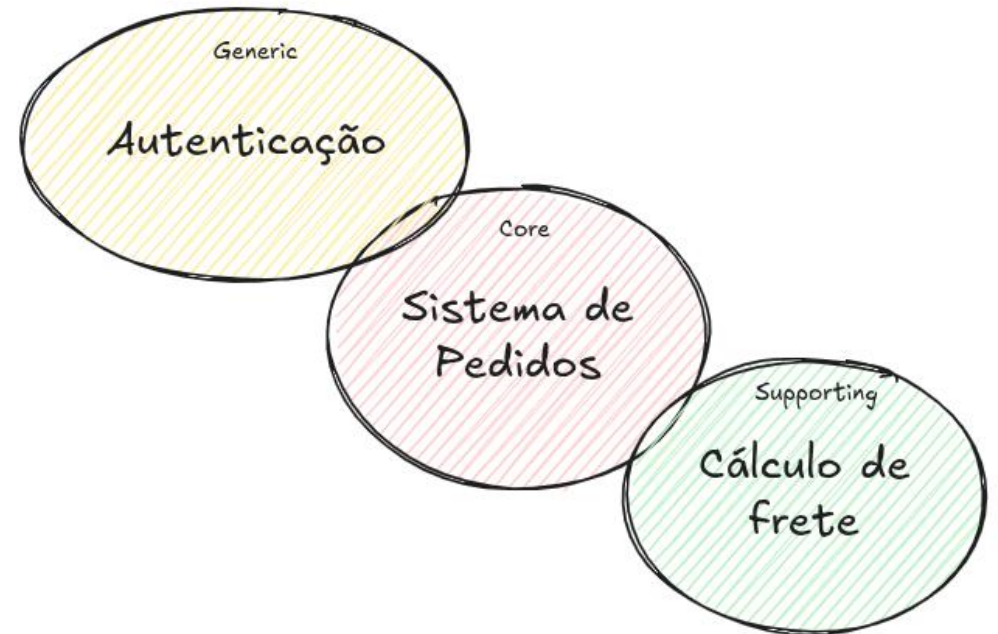
Bounded Contexts

- **Contextos Delimitados (Bounded Contexts)** são uma forma de **definir limites claros dentro dos quais um determinado modelo de domínio é válido**.
- *“DDD é primariamente sobre modelar uma linguagem ubíqua em contexto delimitado”
— Vaughn Vernon*

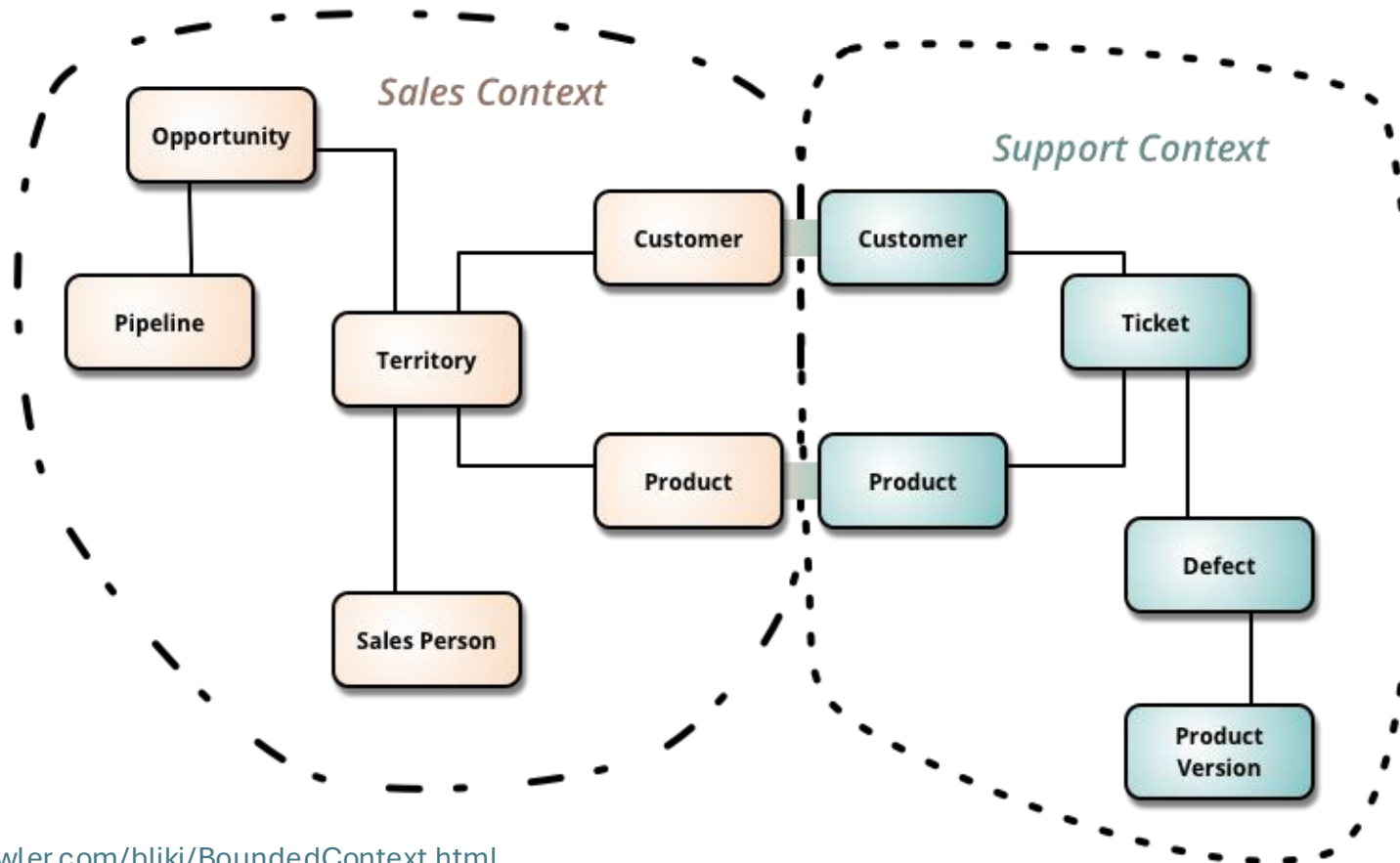
Revisitando conceitos de DDD

Tipos de contexto delimitado

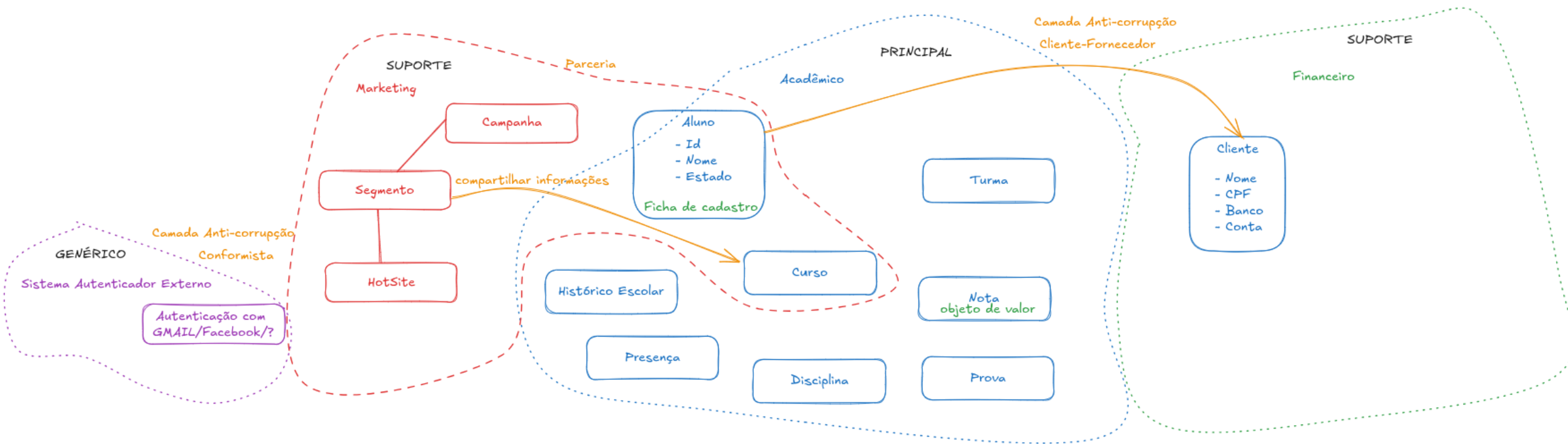
- Core domain
- Supporting Subdomain
- Genéric Subdomain



Revisitando conceitos de DDD



Revisitando conceitos de DDD



Arquitetura de Microserviços

O QUE É?

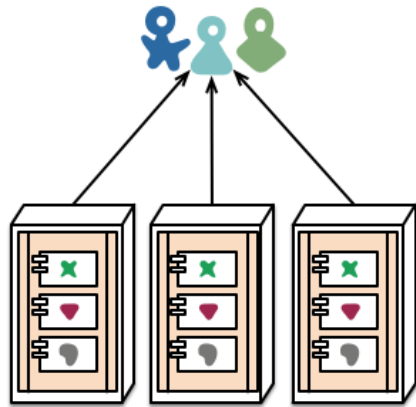


Estilo de arquitetura de software que estrutura uma aplicação como um conjunto de **serviços pequenos, independentes e implementáveis separadamente**

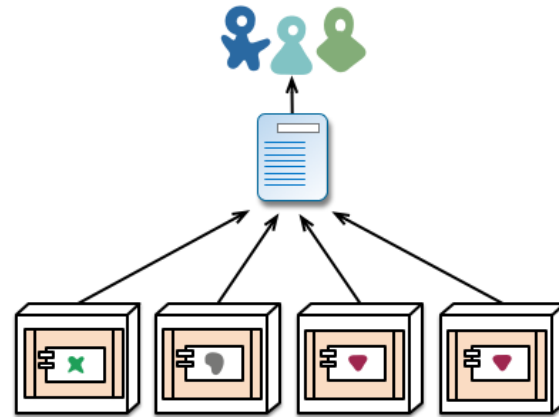
Responsável por uma funcionalidade específica do sistema,
comunicando-se com outros serviços por meio de protocolos leves



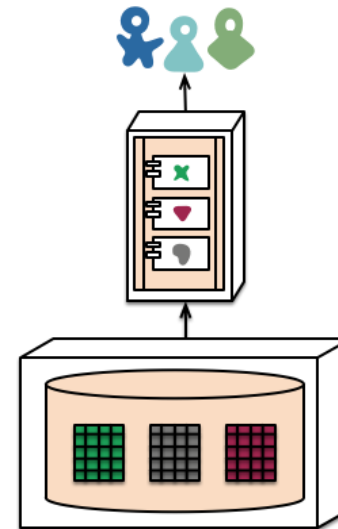
Arquitetura de Microserviços



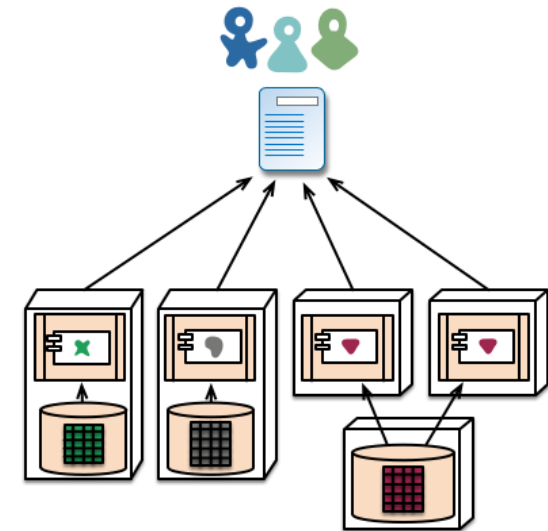
monolith - multiple modules in the same process



microservices - modules running in different processes



monolith - single database

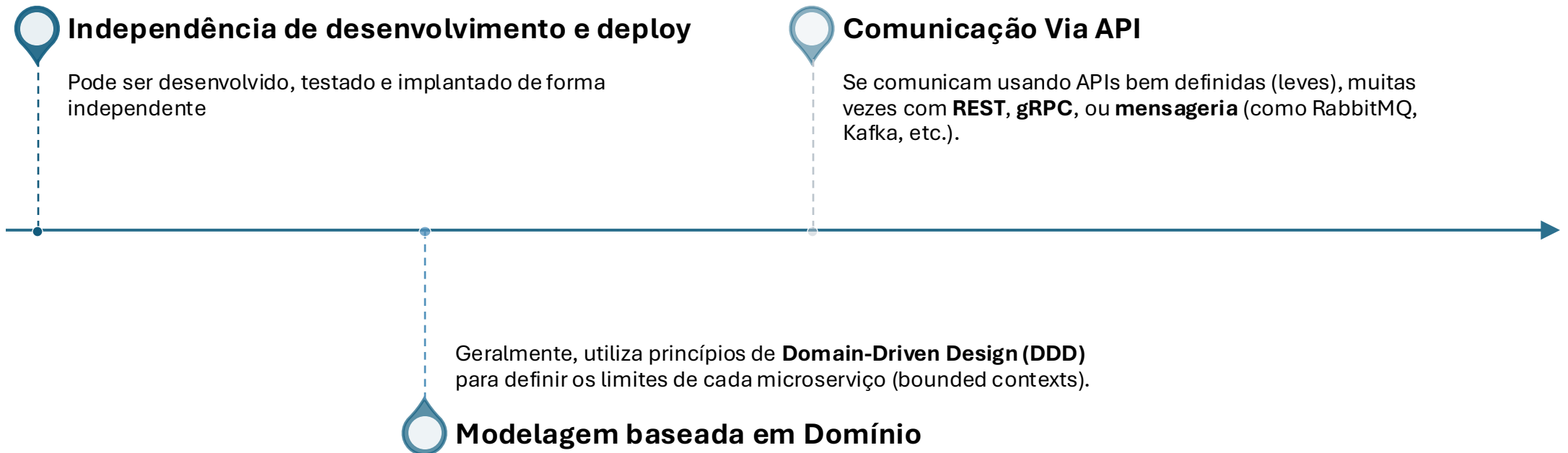


microservices - application databases



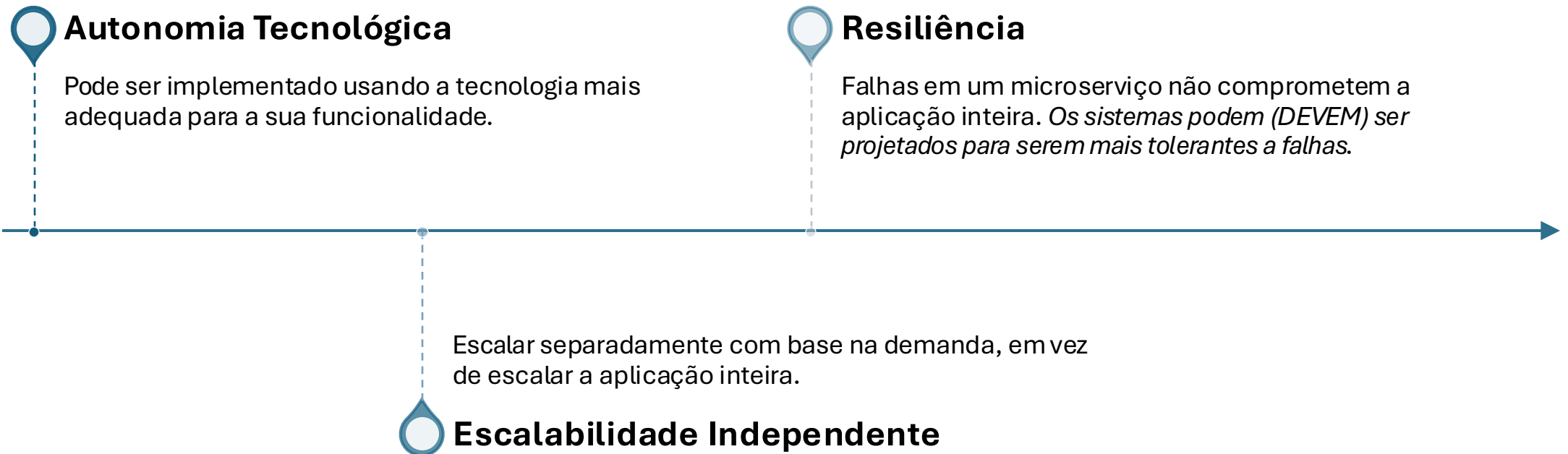
Arquitetura de Microserviços

Características



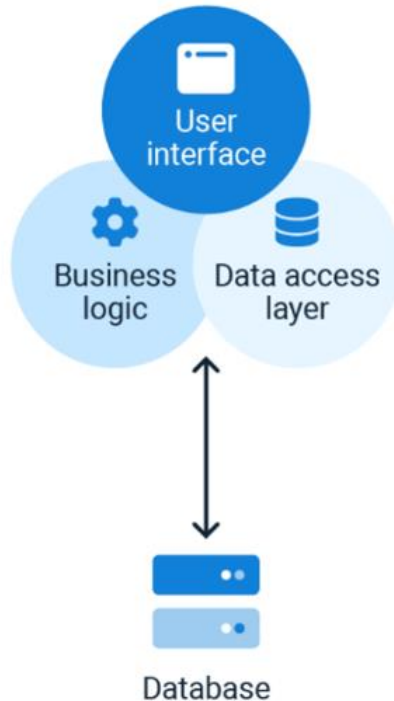
Arquitetura de Microserviços

Características

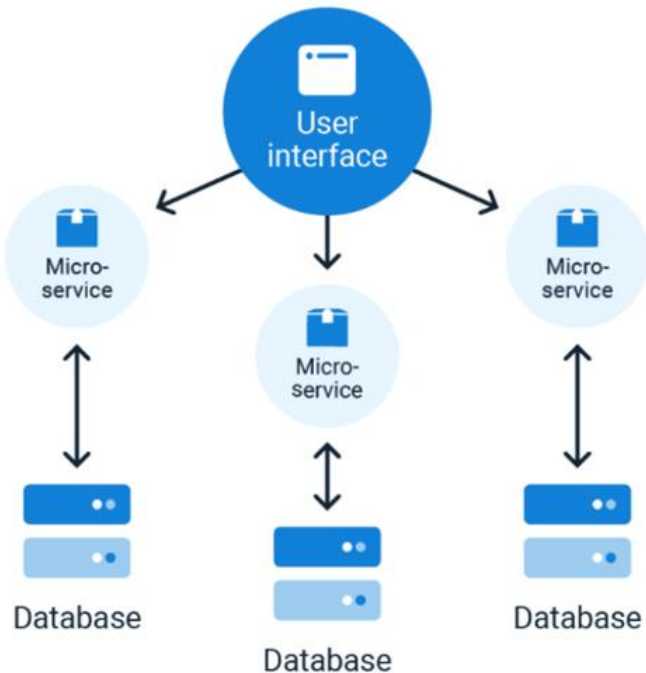


Arquitetura de Microserviços

Monolithic Architecture



Microservice Architecture



Arquitetura de Microserviços

Como criar uma microserviços (Requisito de Negócio)










Site de um sistema de controle de ponto.

- Entrega somente o controle de ponto?
- Todos os módulos são contratados sempre de uma vez?
- Todos os clientes usam o Painel de Risco?

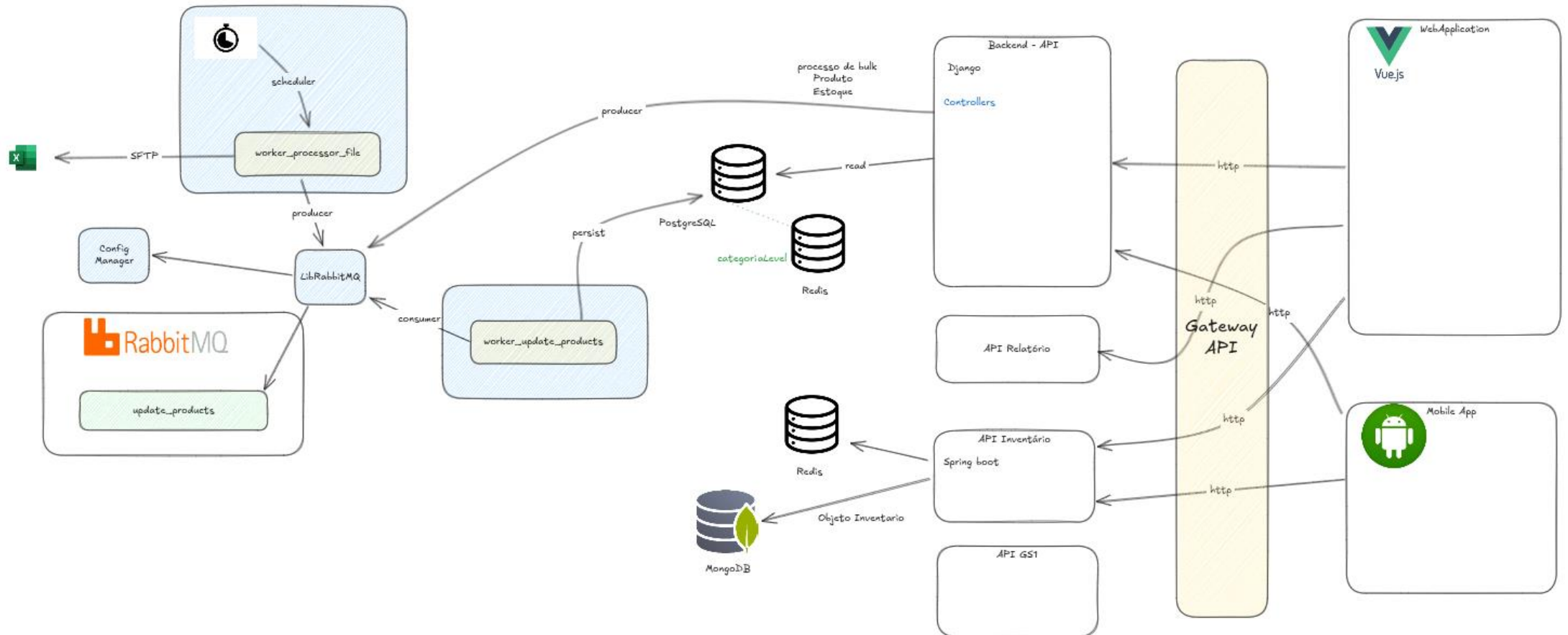
Facilidade é ter tudo para o RH **em um só lugar!**

Conheça as soluções que vão **otimizar a sua rotina de trabalho.**

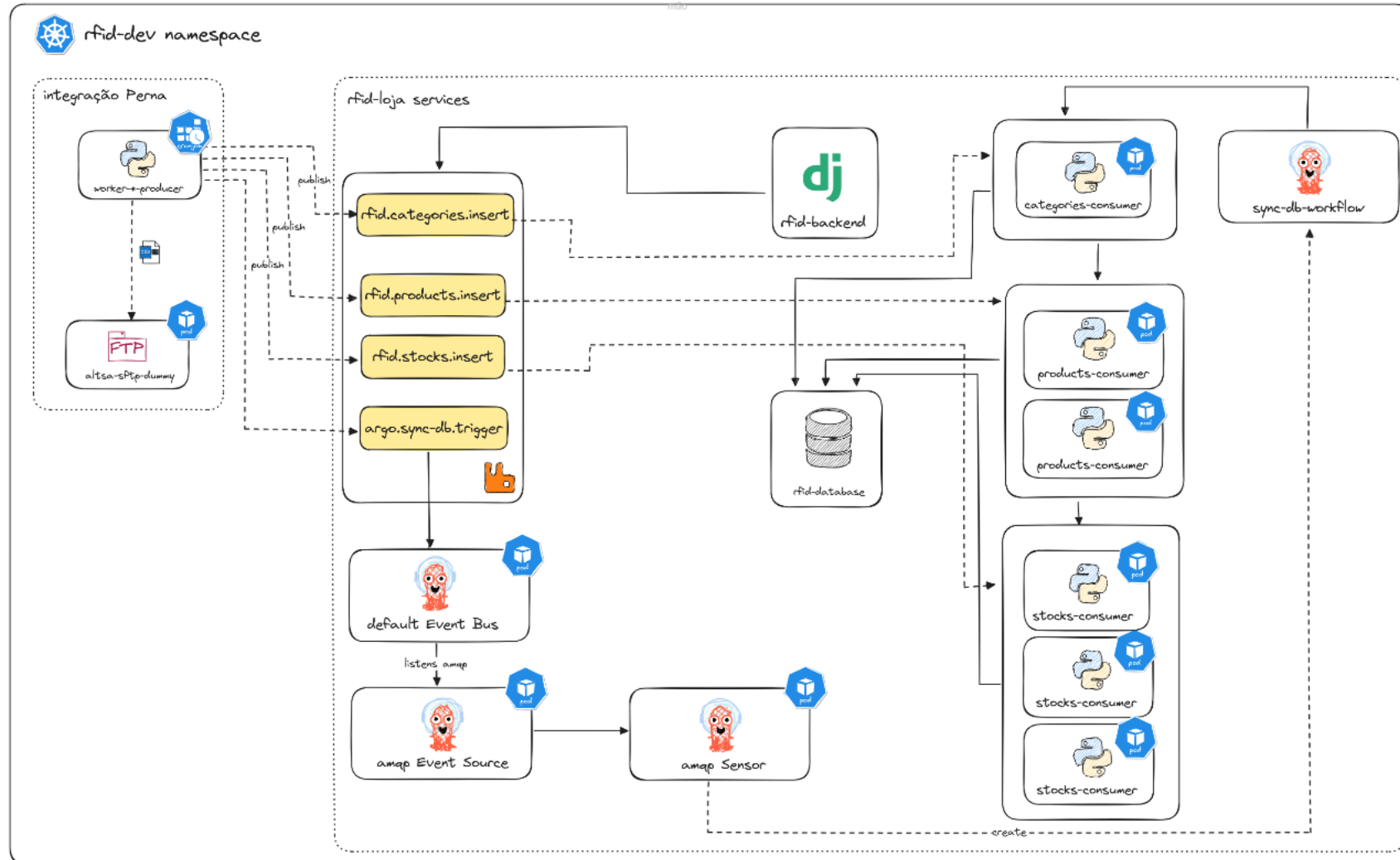
 Controle de ponto	▼
 Férias e folgas	▼
 Gestão de escala	▼
 Admissão de funcionários	▼
 Holerite	▼
 GED Gestão Eletrônica de Documentos	▼
 Painel de Risco	Novidade ▼

Arquitetura de Microserviços

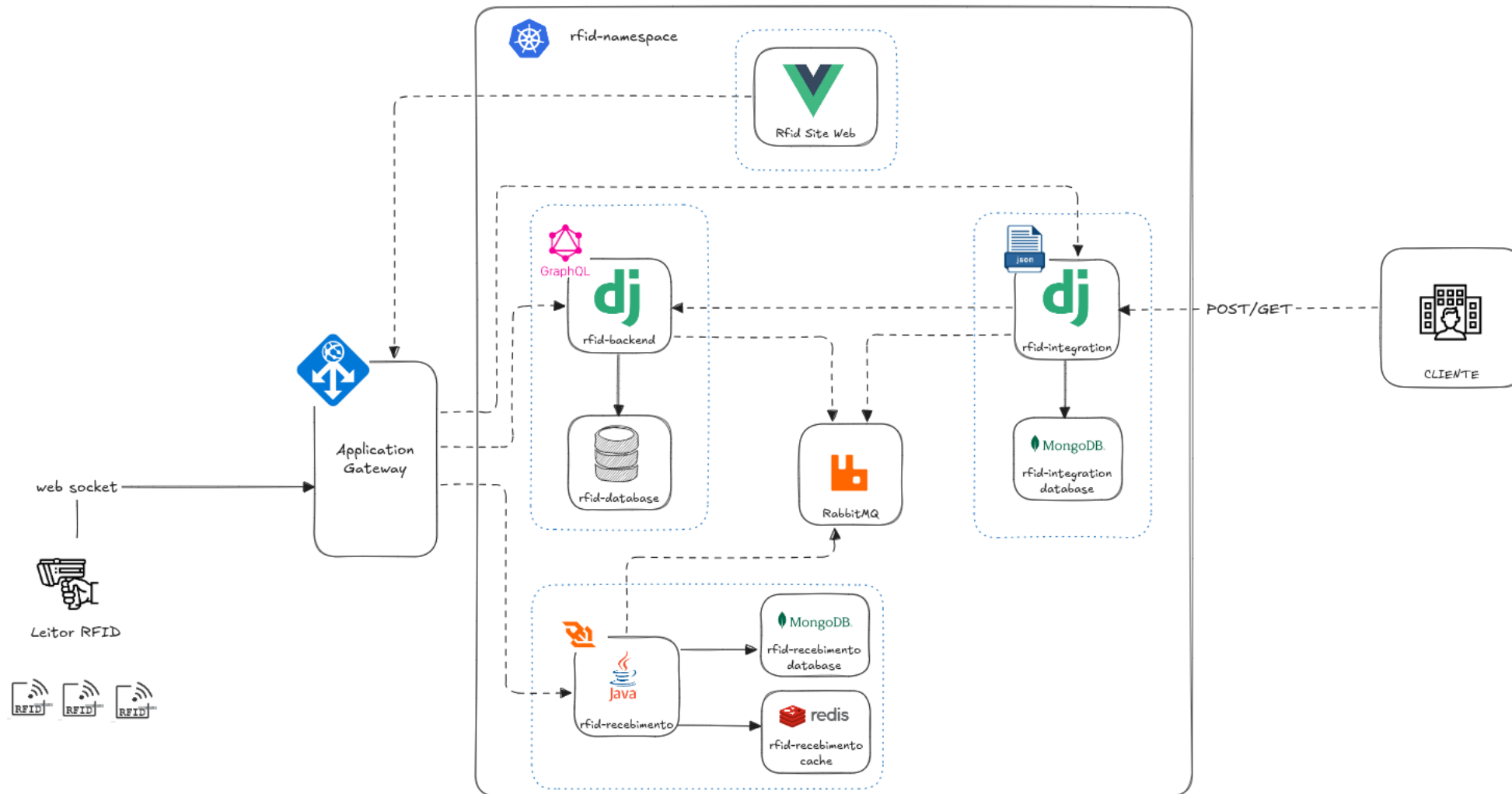
Como criar uma microsserviços (Requisito Técnico)



Arquitetura de Microserviços



Arquitetura de Microsserviços

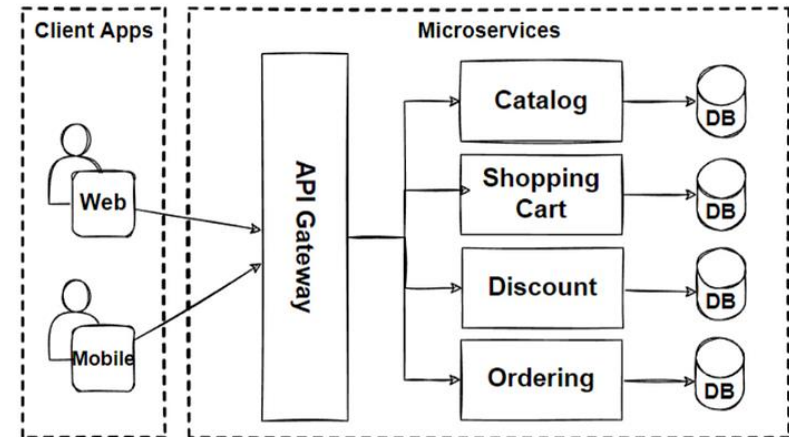


Arquitetura de Microsserviços

Exemplo de código



<https://github.com/csylverio/microservice-example>

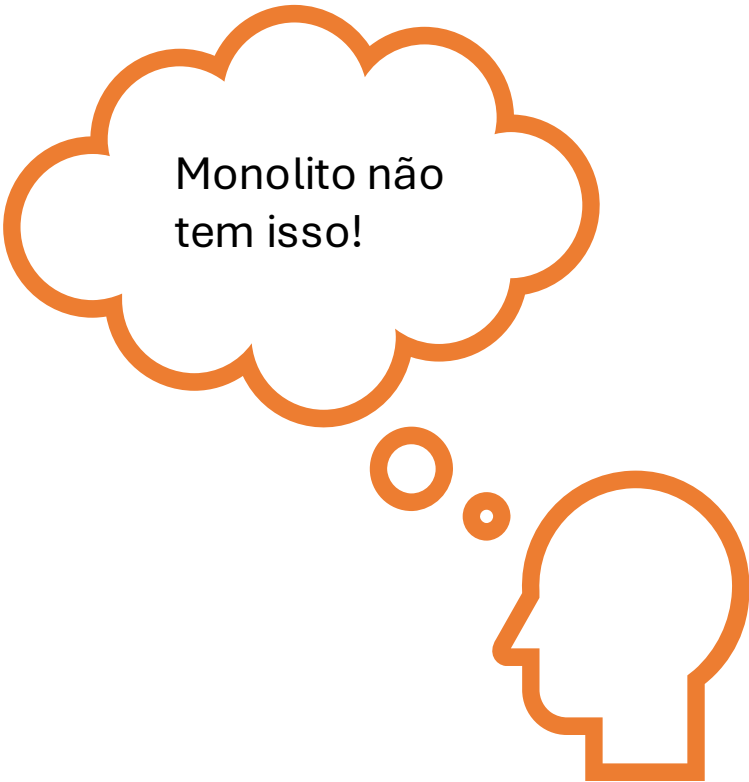


Arquitetura de Microserviços

Analizando o código:



- **Dificuldade de Comunicação**
 1. Envolve redes e protocolos (HTTP, AMQP, gRPC, etc)
 2. Induz latência de rede
 3. Risco de falhas de rede
- **Depuração de código**
 1. Quem vai fazer?
- **Resiliência?**
 1. Dificuldade de dados em um ambiente distribuído
- **Deploy**
 1. Complexidade de DevOps



Monolito não tem isso!

Arquitetura de Microserviços

Analizando o código:



- **Comunicação**
 1. Envolve redes e protocolos (HTTP, AMQP, gRPC, etc)
 2. Induz latência de rede
 3. Risco de falhas de rede
- **Dificuldade de deputação de todo o processo sistêmico**
- **Resiliência?**
 1. Consistência de dados em um ambiente distribuído
- **Orquestração de deploy**
 1. Exige time mais experiente e multidisciplinar



Arquitetura de Microserviços



Como tratar problemas de comunicação

- **Retry Pattern** – Permite a um sistema tentar repetidamente executar uma operação falha antes de considerá-la um erro permanente
- **Linear Breakout** – Adiciona intervalos no tempo das tentativas
- **Jitter** – Adiciona intervalo de tempo aleatório nas tentativas
- **Circuit Break Pattern** - Evitar falhas em cascata, especialmente em sistemas distribuídos
- **Bulkhead Pattern** – Previne que falhas em um componente não causem o colapso do sistema. É também conhecido como arquitetura baseada em células



Arquitetura de Microserviços



Quando não usar!!!

Segundo **Martin Fowler**, a **arquitetura de microserviços** pode trazer muitos benefícios, mas também apresenta desafios significativos, que podem torná-la inadequada em alguns cenários.

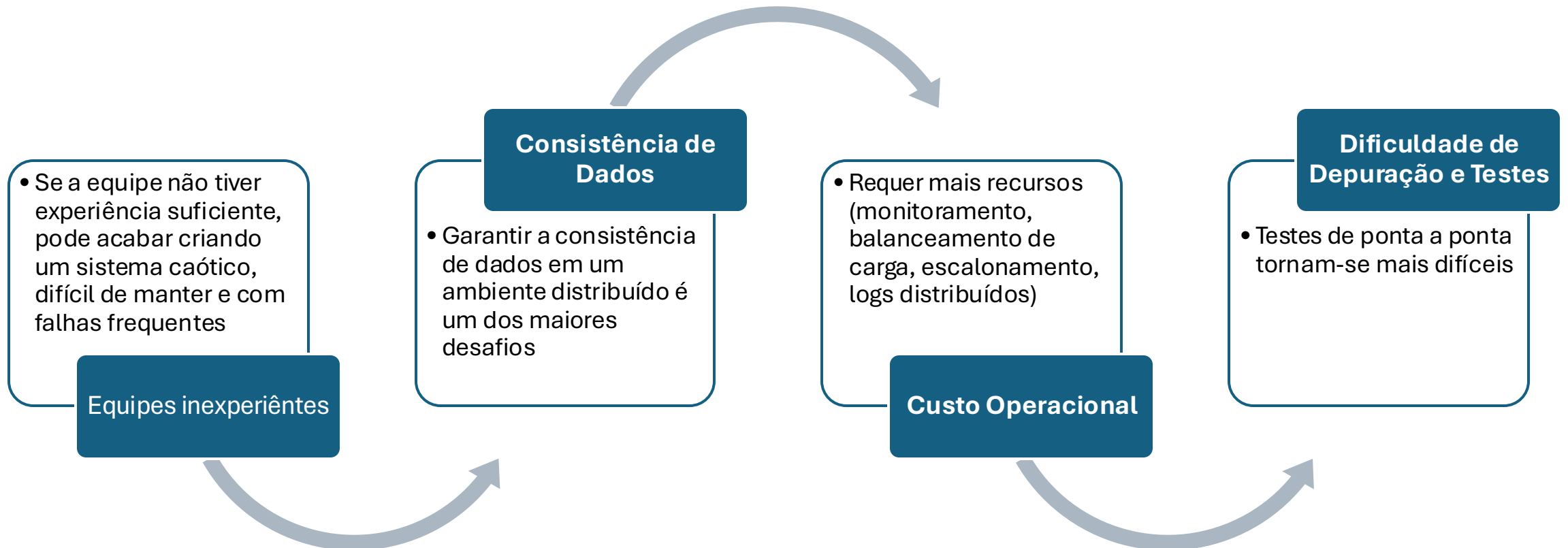
Ele alerta que **não se deve começar um projeto diretamente com microserviços**, especialmente em sistemas simples ou em estágio inicial.

A recomendação é **primeiro construir um monolito modular** e, só depois, avaliar se faz sentido migrar para microserviços.

Arquitetura de Microserviços



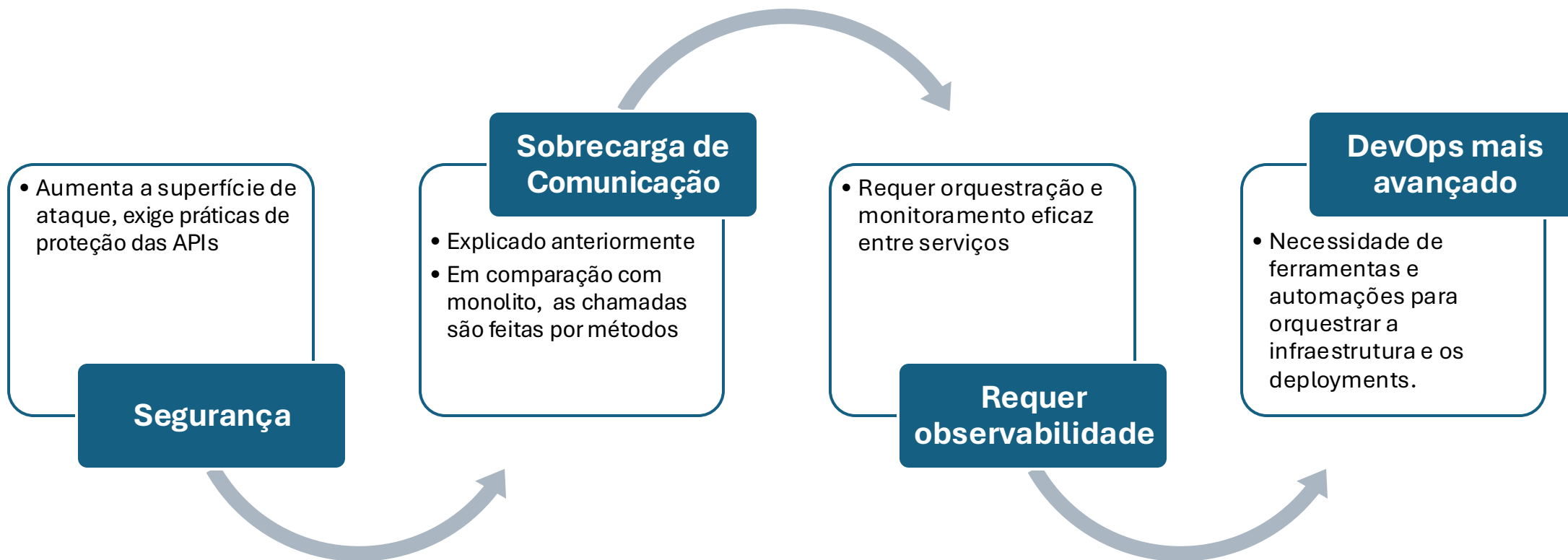
Razões para não utilizar



Arquitetura de Microserviços



Razões para não utilizar



Arquitetura de Microserviços

Dúvidas?

