

# Engenharia de Softwares Escaláveis

Domain-Driven Design (DDD) e Arquitetura de  
Softwares Escaláveis com Java

# Agenda

**Etapas 1:** Aplicando Aggregates e Bounded Contexts.

- Modelos de Computação.
- Modularização de Software.
- Modularidade Arquitetônica.



# Modelos de Computação

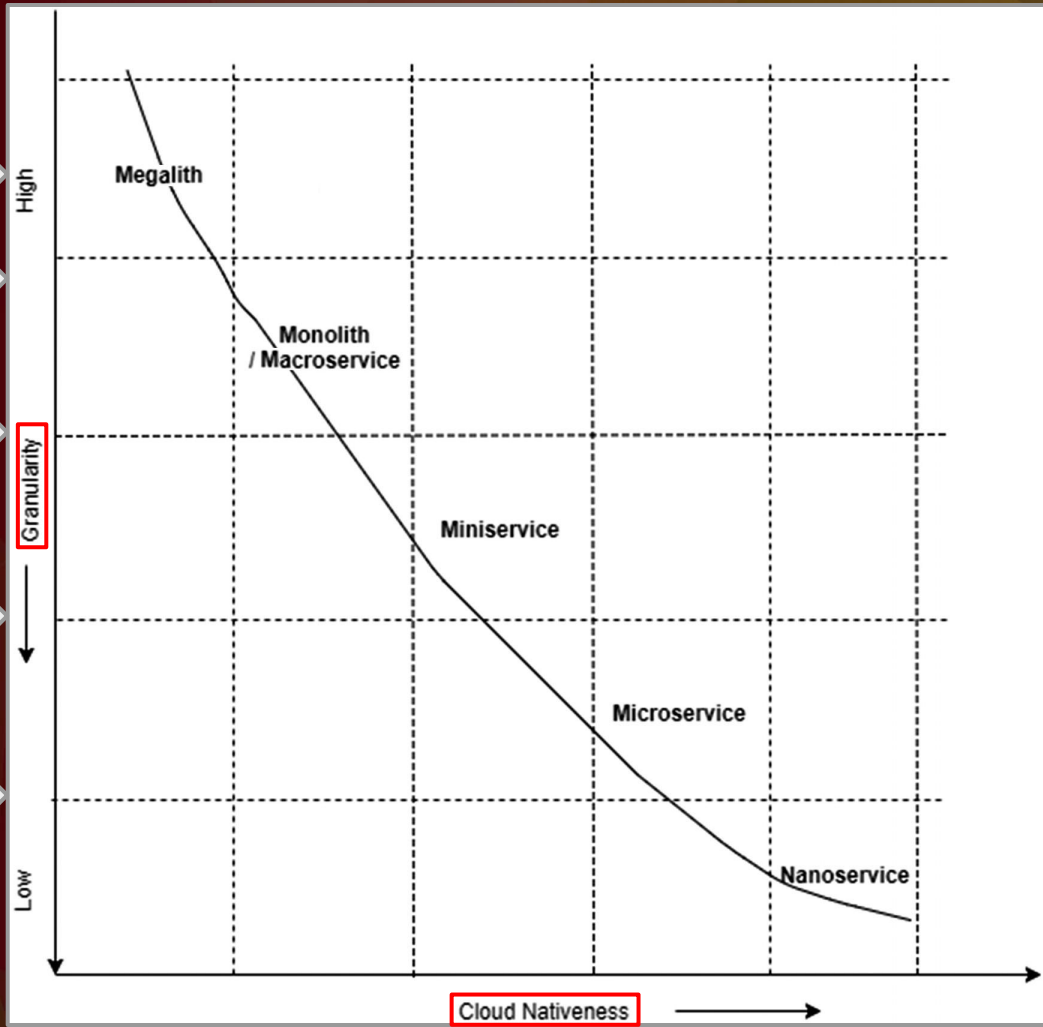
Mainframe

Cliente-Servidor

3-Tier

N-Tier

Serverless



## **Customer-Controlled Services**

Oferecem ao negócio total flexibilidade (dentro da gama de produtos do mercado), mas colocam muito mais ênfase nos processos de design, construção e operação.

Esta é uma das razões para o aumento de soluções convergentes no início da década de 2010, que procuravam simplificar o processo de tomada de decisão.

## **Vendor-Controlled Service**

São totalmente projetadas e construídas pelo fornecedor.

Os clientes têm pouca ou nenhuma participação no design de serviços, estruturas de dados ou modelos de segurança (dependendo da camada de implementação).

O fornecedor determina a velocidade da inovação, o que pode ser uma bênção ou uma maldição.

Vendor-Controlled

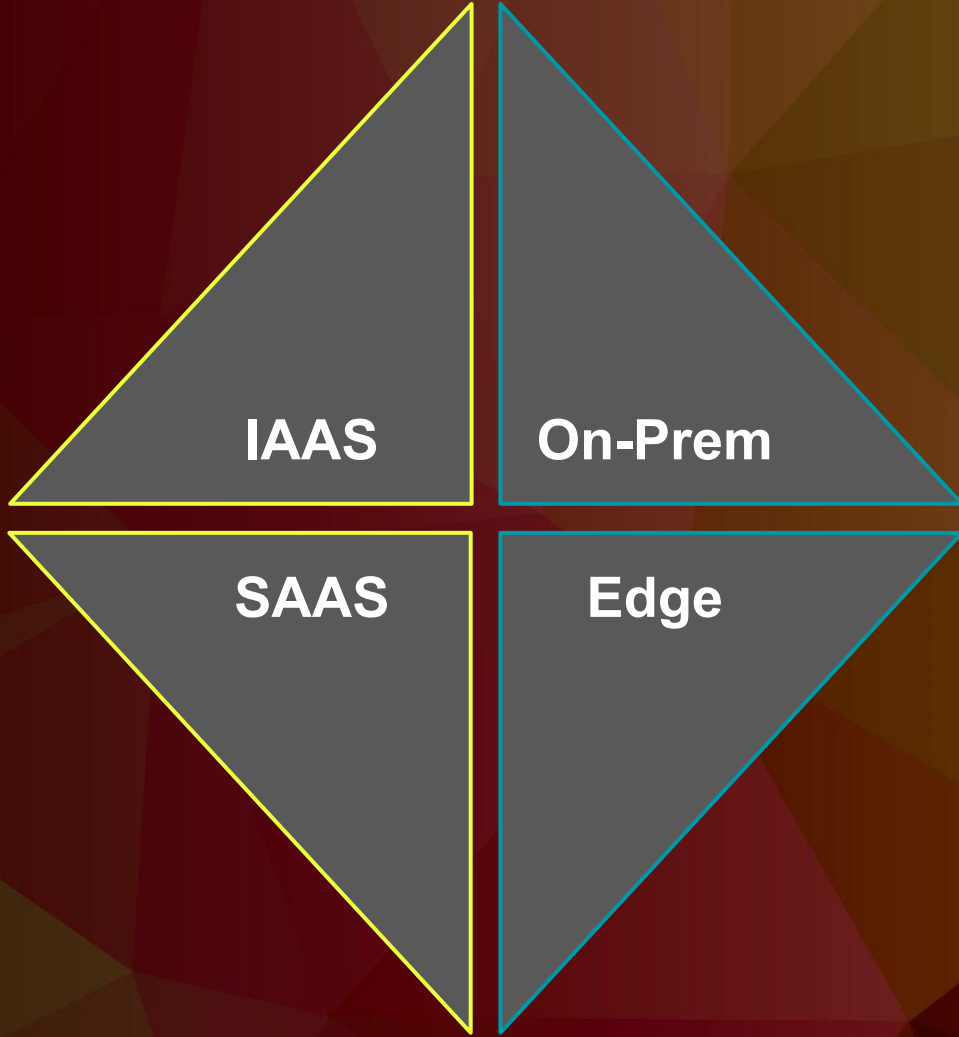
**IAAS**

**SAAS**

**On-Prem**

**Edge**

Customer-Controlled



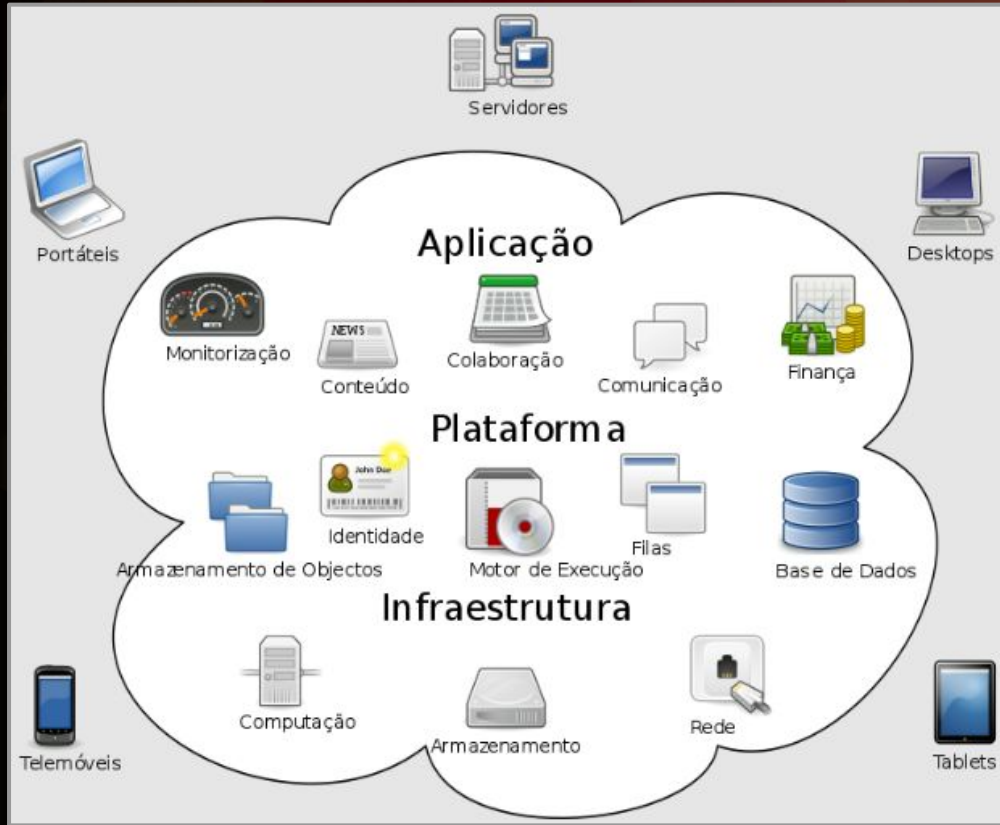
**IaaS** Infraestrutura como serviço são blocos de construção LEGO de armazenamento, computação e rede, que podem ser combinados para criar serviços mais complexos.

**SaaS** Software como serviço é uma variedade de soluções que fornecem funcionalidade padrão de processos de negócios e alguns serviços de infraestrutura interna.

**On Premises** é a infraestrutura autogerenciada construída em data centers alugados ou de propriedade do cliente.

**Edge** são unidades menores de processamento fora de data centers que oferecem serviços de computação locais, incluindo criação de dados.





**Cloud Computing** é um termo para a disponibilidade sob demanda de recursos computacionais, sem o gerenciamento ativo direto do utilizador (**Vendor-Controlled**).

Nuvens em grande escala, geralmente têm funções distribuídas em vários locais dos servidores centrais.

Se a conexão com o utilizador for relativamente próxima, pode ser designado um servidor de borda.



# Modularização de Software

A construção de software, desde sempre, sugere a separação das várias funcionalidades em partes menores para promover a organização.

Essa divisão pode ser feita a partir de vários critérios e esse é o principal desafio da **modularização**.

Por exemplo, em uma aplicação os componentes podem ser **agrupados em domínios**, como Vendas, Estoque e Produção.

Por outro lado, no **modelo *microkernel***, as funcionalidades são particionadas em componentes de plug-in separados, permitindo um escopo de teste e implantação muito menor.

A **modularização** é diretamente afetada pelos vários modelos e estilos de **arquitetura de software**.

A tradução de responsabilidades em classes e métodos é influenciada pela **granularidade da responsabilidade**.

**Grandes responsabilidades** exigem centenas de classes e métodos.

**Pequenas responsabilidades** podem exigir somente um método.

Por exemplo, a responsabilidade de “fornecer acesso a bancos de dados relacionais” pode envolver duzentas classes e milhares de métodos, empacotados em um subsistema.

Por outro lado, a responsabilidade de “criar uma Venda ” pode envolver apenas um método em uma classe.

**Uma responsabilidade não é a mesma coisa que um método** – é uma abstração – mas os métodos cumprem responsabilidades.

**Acoplamento** é uma medida de **quão fortemente um elemento está conectado**, tem conhecimento ou depende de outros elementos.

Se houver **acoplamento ou dependência**, então, quando o elemento dependente for alterado, o dependente poderá ser afetado.

Por exemplo, uma subclasse está fortemente acoplada a uma superclasse.

Um objeto A que chama as operações do objeto B possui acoplamento aos serviços de B.

**Coesão** mede informalmente **o quão funcionalmente relacionadas** estão as operações de um elemento de software e também mede quanto trabalho um elemento de software está realizando.

Por exemplo, um objeto **Grande** com 100 métodos e 2.000 linhas de código-fonte está fazendo muito mais do que um objeto **Pequeno** com 10 métodos e 200 linhas de código-fonte.

E se os 100 métodos do **Grande** cobrem muitas responsabilidades diferentes (como acesso ao banco de dados e geração de números aleatórios), então ele tem menos **coesão funcional** do que o **Pequeno**.

A **quantidade de código** e o **relacionamento do código** são indicadores da coesão de um objeto.

## Baixo Acoplamento e Alta Coesão

P → Como implementar o **Baixo Acoplamento**?

R → Atribua responsabilidades para que o acoplamento (desnecessário) permaneça baixo. Use este princípio para avaliar alternativas.

P → Como implementar a **Alta Coesão**?

R → Atribua responsabilidades para que a coesão permaneça elevada. Use isso para avaliar alternativas.



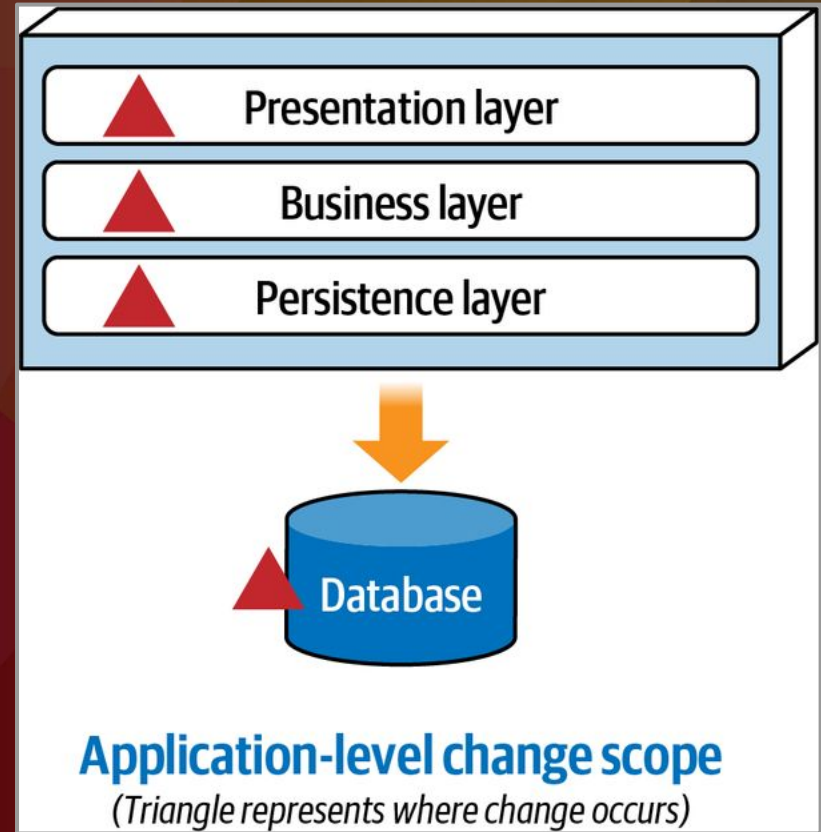


# Modularidade Arquitetônica

**Aplicações Monolíticas** podem ser bem-sucedidas, porém as pessoas começarão a ficar frustradas.





Ciclos de mudanças começam a ficar amarrados – uma pequena alteração feita em uma parte do software faz com que toda a aplicação necessite ser republicada.

Com o passar do tempo ficará cada vez mais difícil manter uma estrutura modular, sendo difícil separar as mudanças que deveriam afetar somente um módulo.

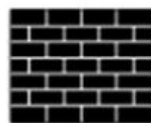


# Aplicação Monolítica

## Legend

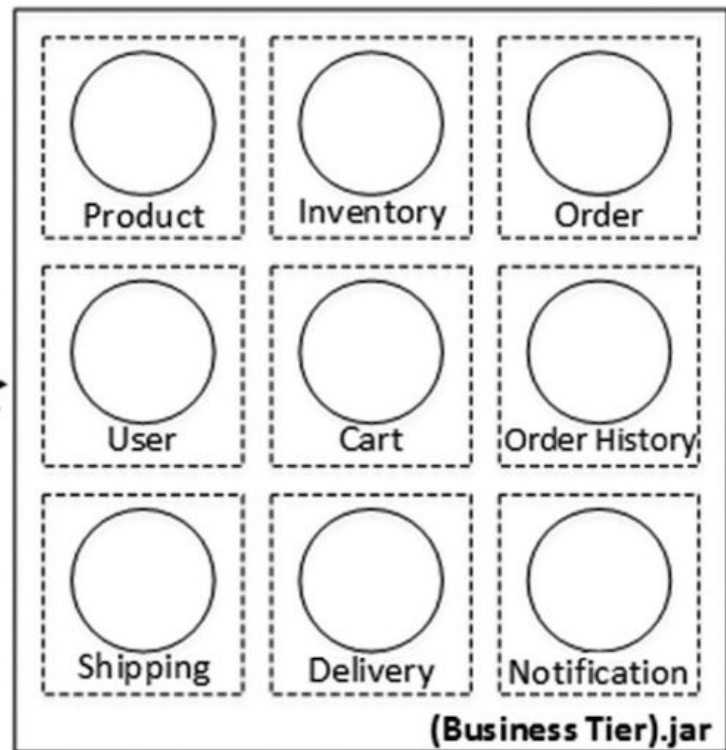
-  Service
-  Module / Functional Boundary
-  Application / System Boundary
-  Request Routes

Request in



Firewall

Route  
Request



(Application).ear

### Legend

- Service
- Module / Functional Boundary
- ▭ Application / System Boundary
- Request Routes

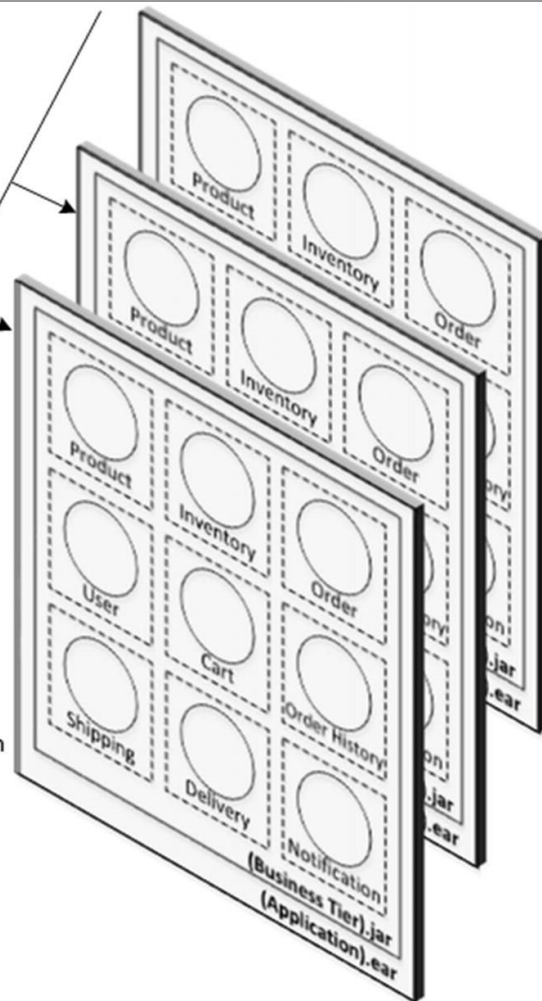
Request in



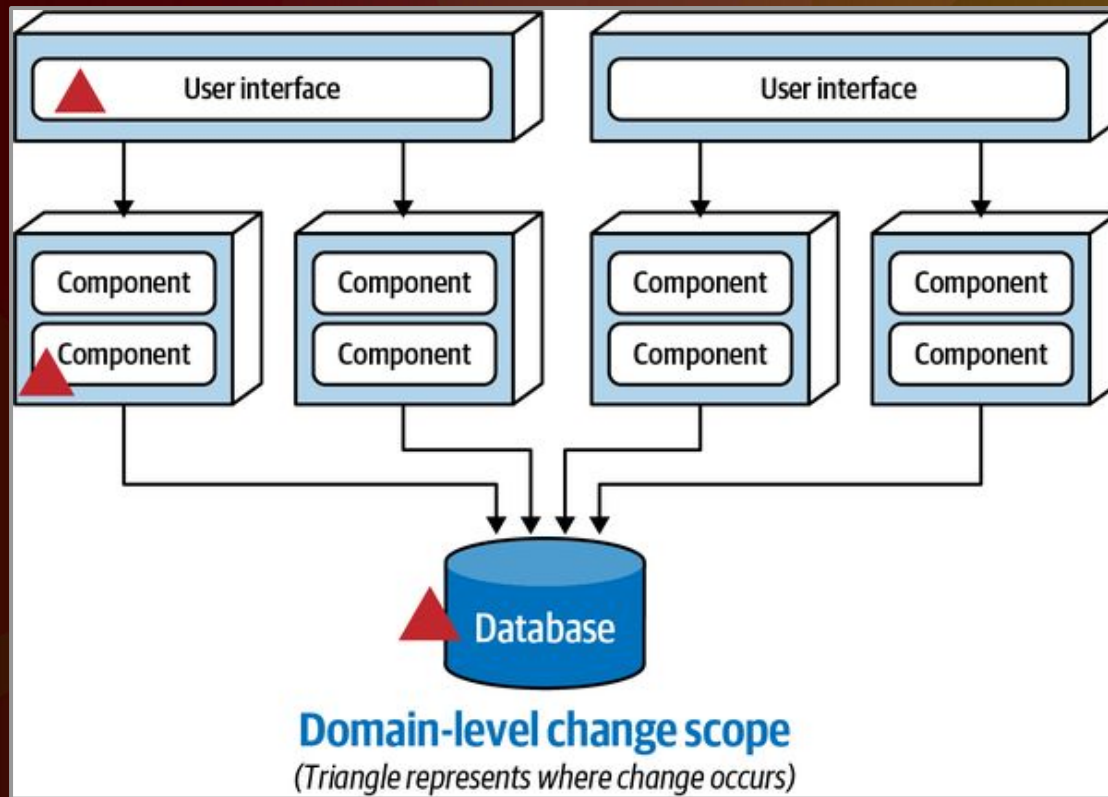
Firewall



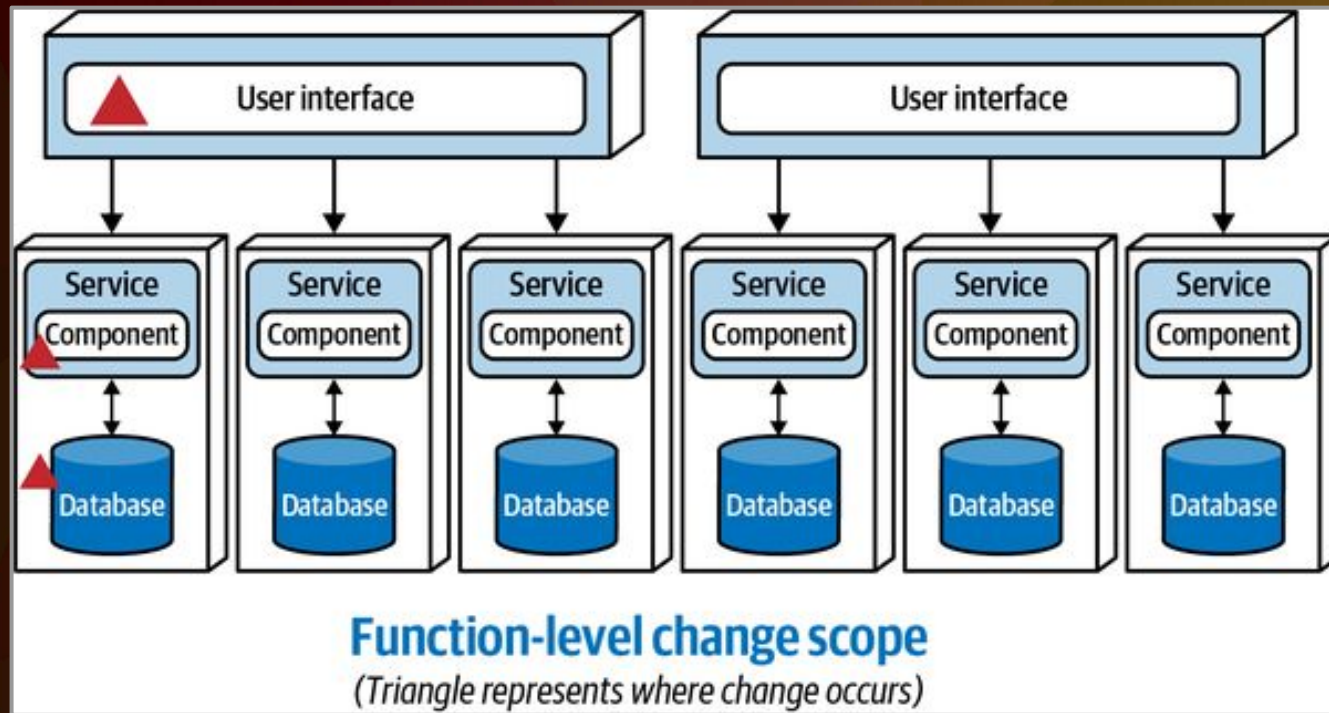
App  
Server Farm



Escalabilidade  
Horizontal



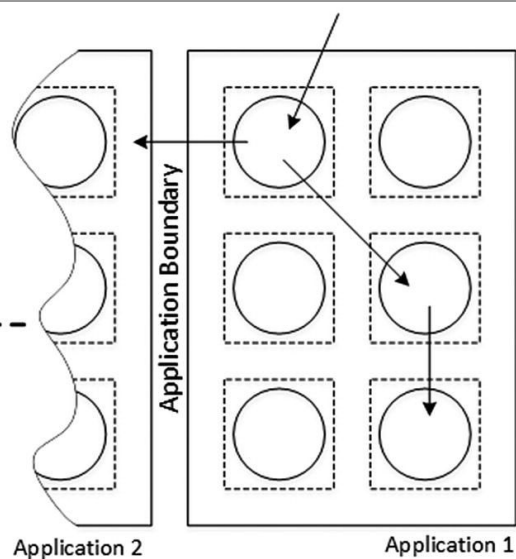
**Arquiteturas Modulares** particionam domínios e subdomínios em unidades de software menores, implantadas separadamente, facilitando assim a modificação de um domínio ou subdomínio.



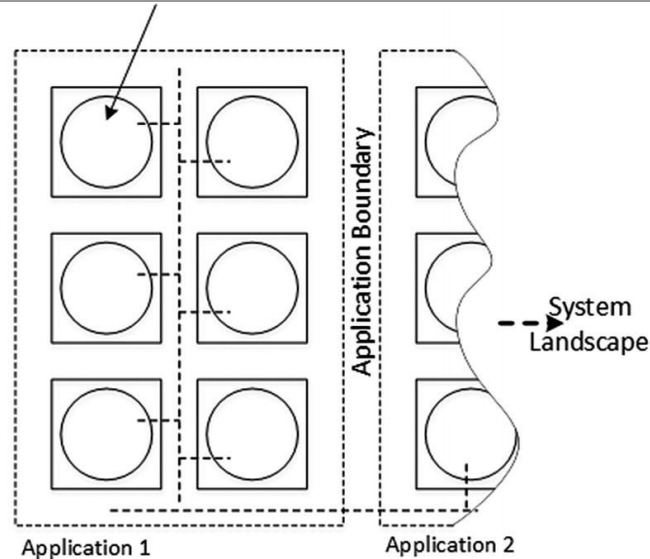
**Arquitetura de Microsserviços** coloca o novo requisito em um escopo de mudança em nível de função, isolando a mudança em um serviço específico, onde se espera **alta coesão** funcional (cada componente dispões de um conjunto bem definido de responsabilidades).



System  
Landscape








Monolith Architecture



Microservices Architecture

Legend

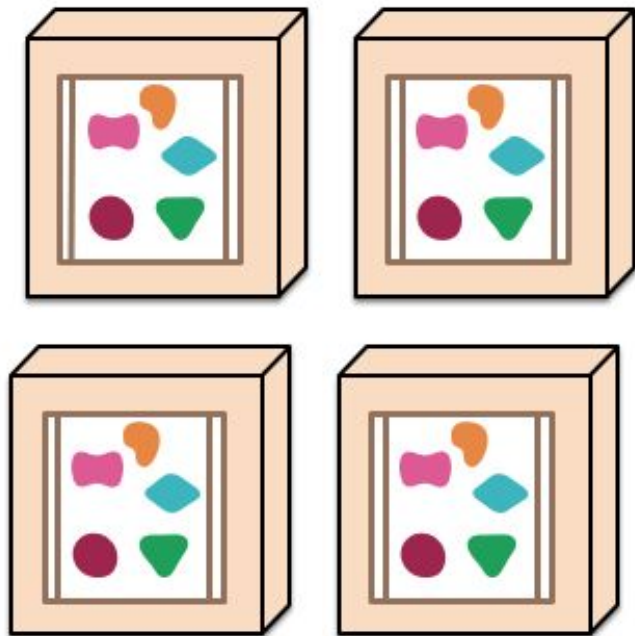
-  Service
-  Module / Functional Boundary
-  Application / System Boundary
-  (Service to) Service Interactions
-  Messaging Backbone

Migrando  
Monólitos para  
Microserviços

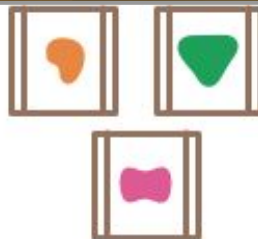
Uma aplicação monolítica coloca toda sua funcionalidade em um único processo...



... e escala replicando a aplicação monolítica em vários servidores



Uma arquitetura em microsserviços põe cada elemento de uma funcionalidade em um serviço separado ...



... e escala distribuindo estes serviços entre os servidores, replicando quando necessário.

