01 - Introdução

- O que é Arquitetura de Software?
    - Muitas funções
    - Nem sempre claro definir
    - Contém muitas ideia ultrapassadas
    - Contexto muda o que é viável
- Definição
    - Blueprint?
    - Roadmap? - Quatro componentes da Arquitetura

    1. Estrutura
        - Estilo geral (e.g. microsserviços)
    2. Características
        - Critérios de sucesso (e.g. escalabilidade, testabilidade, etc)
    3. Decisões
        - Regras do sistema (e.g. quais camadas fazem o quê)
        - Pode ter variâncias (exceções)
    4. Princípios
        - Diretriz
            - não é uma regra (e.g. usar mensageria assíncrona)
- Oito Expectativas

    1. Tomar decisões de arquitetura
        - Orientar, não especificar, quando der
    2. Analisar continuamente a arquitetura
        - Verificar se decisões antigas seguem boas
    3. Assegurar conformidade com decisões
        - Inspecionar desenvolvimento
    4. Exposição a experiências diversificadas
        - Amplitude > Profundidade
    5. Ter conhecimento sobre o domínio do negócio
    6. Ter habilidade interpessoal
        - Tão importante quanto técnica
    7. Entender e lidar bem com questões políticas
        - Toda decisão será questionada
- Práticas de engenharia
    - Escala Elástica - gearar mais instâncias de recursos sob demanda
    - Arquitetura != Desenvolvimento (e.g. Agile) - por tradição
    - [XP - Extreme programming --> Continuos Delivery]
- Impossível antecipar incógnita desconhecidas
- "Arquitetura Evolutiva" - fácil de alterar e adaptar
- Avaliação objetiva das características da arquitetura
- Devops
    - Associação óbvia com arquitetura
    - 1990 a 200 - terceirizavam operação - arquitetura que limitava operadores
- Métricas, testes, etc
- Processo
    - Processo (como é criado) é independente da arquitetura (estrutura)
    - Mas pode não ser completamente
        - E.g. considerar ciclo mais rápido de feedback
        - Migração de arquitetura -> Agile
- Dados
    - DB-Admin trabalham junto de arquiteto
    - Simbiose
- 1ª LEI - Tudo é uma concessão (trade-off)
- 2ª LEI - O porquê vale mais que o como

03 - Modularidade

- Difícil definir modularidade
- Princípio organizacional
- Agrupamento lógico de código (não físico necessariamente)
- Importantes implicações na arquitetura
- Anos 70 - Programação Modular
- **MÉTRICAS**
- Coesão -
    - Quanto que as partes precisam estar juntas
    - Mede interdependência
    - Vários tipos:
        - Funcional
        - Sequencial
        - Comunicacional
        - Procedural
        - Temporal
        - Lógica
        - Coincidental (ruim - junto mas não devia)
    - Como agrupar código - trade-off

Forma 1	**Forma 2**
Customer Maintenance • add customer • update customer • get customer • notify customer • get customer orders • cancel customer orders	Customer Maintenance • add customer • update customer • get customer • notify customer
	Order Maintenance • get customer orders • cancel customer orders

- Métrica disso é mais subjetiva
- Métrica Chindambor e Kemer (CK) - > para OOP
    - LCOM -
        - Mede falta de coesão em métodos
        - Ajuda a achar métodos que deveriam estar separados
        - Tem falhas - Acoplamento -
- Análise por grafos - mais objetivo
- (Aferente - entrada; Eferente - saída)
- **ABSTRAÇÃO:**
    - Proporção de elementos abstratos para elementos concretos

      Equation 3-3. Abstractness

      $$A = \frac{\sum m^a}{\sum m^c}$$
- **INSTABILIDADE:**
    - Determina volatilidade do código
    - Quanto maior a volatilidade, maior a tendência de quebrar quando mudar
    - Classe que chama muitas outras classes para delegar tarefa é mais instável

      Equation 3-4. Instability

      $$I = \frac{C^e}{C^e + C^a}$$
- **DISTÂNCIA DA SEQUÊNCIA PRINCIPAL**
    - Relaciona abstração e instabilidade

Equation 3-5. Distance from the main sequence

$$D = |A + I - 1|$$

- Quanto menor D, mais equilibrada
- Zona de Inutilidade:
  - Abstrato demais
- Zona de sofrimento:
  - Sem abstração e difícil de manter

- Conascência:
  - Quanto que uma mudança em um componente requer mudanças em outros
  - Estática (a nível de código fonte)
    - De Nome
    - De Tipo
    - De Significado, convenção
    - De posição
    - De algoritmo
  - Dinâmica (a nível de execução)
    - De execução
    - De tempo
    - De valores
    - De identidade
  - Propriedades da conascência:
    - **Força**
      - Facilidade de refatorar
      - Conascência de nome apenas é o ideal
      - Conascência estática é melhor que dinâmica

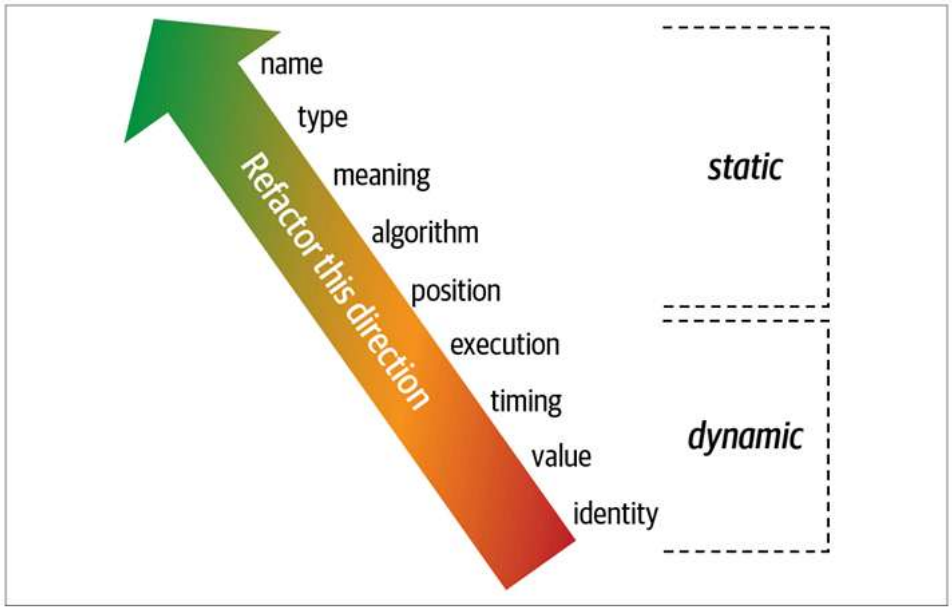


Figure 3-5. The strength on connascence provides a good refactoring guide

    - **Localização**
      - Proximidade lógica (e.g. mesmo módulo) entre códigos
      - Conascência em código próximo é preferível
    - **Grau**
      - Impacto da conascência
      - Quantas classes são afetadas
  - Diretrizes:
    - Encapsular partes do sistema
    - Minimizar conascência entre partes
    - Maximizar conascência intrapartes
  - Regra do Grau:
    - Converter conascência forte em fraca
  - Regra da localização:
    - Quanto maior a distância, mais fraca deve ser a forma de conascência - Acoplamento é similar a conascência - overlap

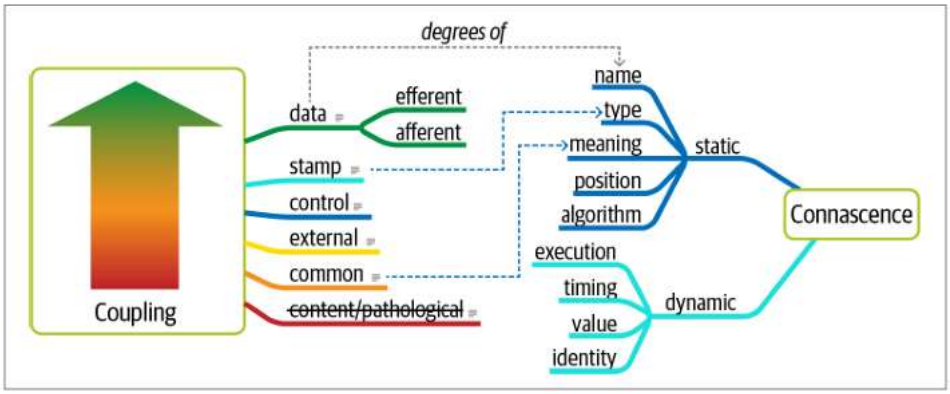


Figure 3-6. Unifying coupling and connascence

- Métrica veem detalhes do código mas não estrutura da arquitetura
- Conascência não é tudo
  - Não responde decisões fundamentais a serem tomadas

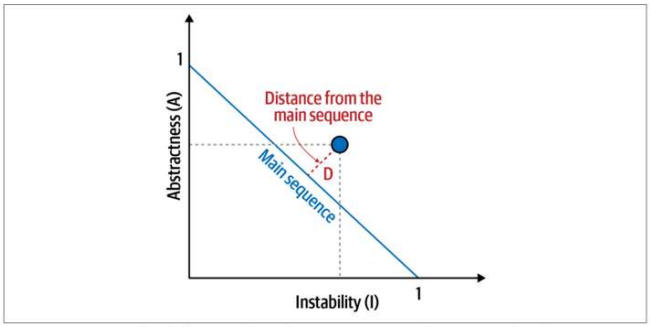


Figure 3-3. Normalized distance from the main sequence for a particular class

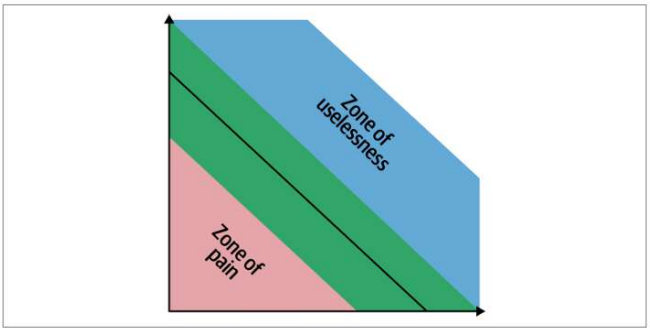


Figure 3-4. Zones of Uselessness and Pain

02 - Pensamento Arquitetônico

- Pensamento arquitetônico != pensar na arquitetura
- Arquitetura != Design

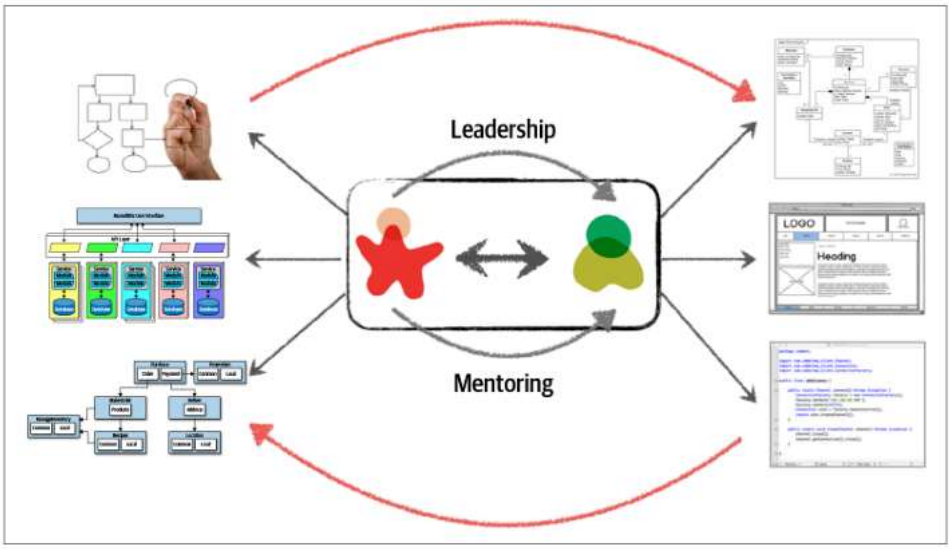


Figure 2-2. Making architecture work through collaboration

- Arquiteto deve estar em contato com devs
- Não é unidirecional - mentoria & liderança constante
- Iterativo
- **CONHECIMENTO**
    - Desenvolvedor:
        - Profundidade
    - Arquiteto
        - Amplitude
    - Difícil transição
    - Riscos de:
        - Especializações demais e rasas
        - Especializações obsoletas - dominar algo velho sem sabê-lo
            - "Frozen Caveman Antipattern"
- Tudo é um trade-off
- "Arquitetura é o que você não consegue pesquisar no Google"
- Tudo "Depende"
    - Não há certo ou errado
- Exemplo do Leilão:

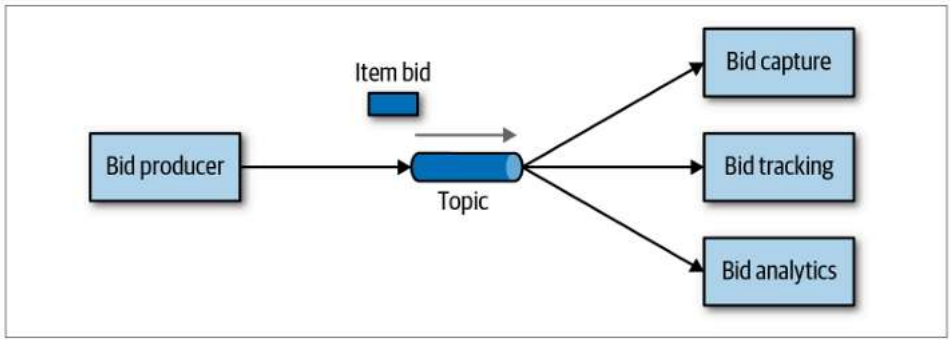


Figure 2-8. Use of a topic for communication between services

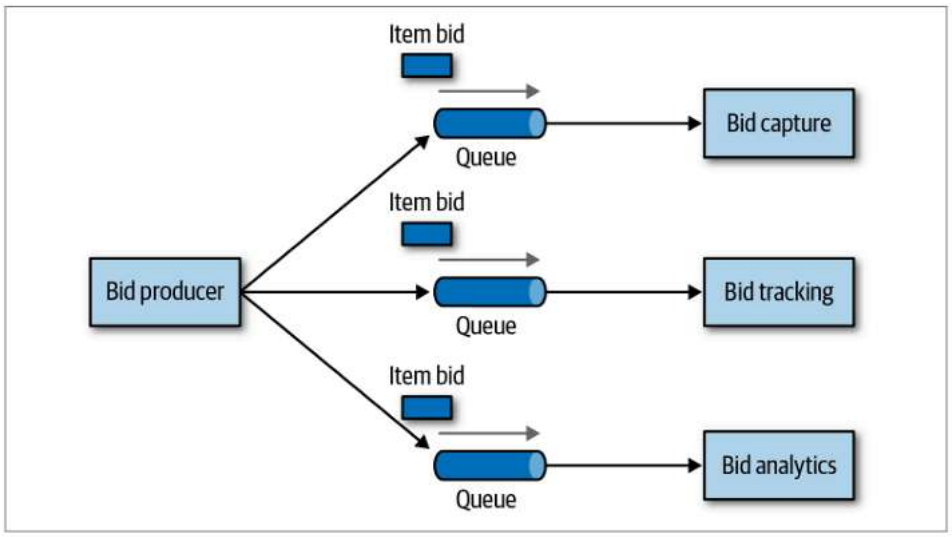


Figure 2-9. Use of queues for communication between services

-
- Qual escolher?:
  - 1 tópico é mais fácil de estender mais uma funcionalidade
  - 3 filas é mais seguro e separável, melhor load balancing

    Table 2-1. Trade-offs for topics

Topic advantages	Topic disadvantages
Architectural extensibility	Data access and data security concerns
Service decoupling	No heterogeneous contracts
	Monitoring and programmatic scalability
  -
- Equilibrar código com arquitetura nas suas funções como arquiteto
  - Arquiteto tem muitas funções, não pode se dedicar tanto ao desenvolvimento:
    - Delegar o caminho crítico a um dev
    - Codificar só coisas importante no futuro, não o gargalo atual
    - Fazer algo próximo do que devs fazem
    - POC - a melhor possível, já que código se tornará referência
    - Resolver déficit técnicos
    - Consertar erros
    - Criar ferramentas auxiliares - e.g. validadores
    - Code Review

04 - Definição das Características de Arquitetura

- Vários critérios - requisitos

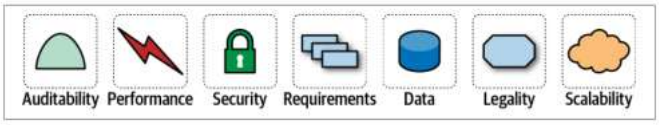


Figure 4-1. A software solution consists of both domain requirements and architectural characteristics

- - Características de arquitetura

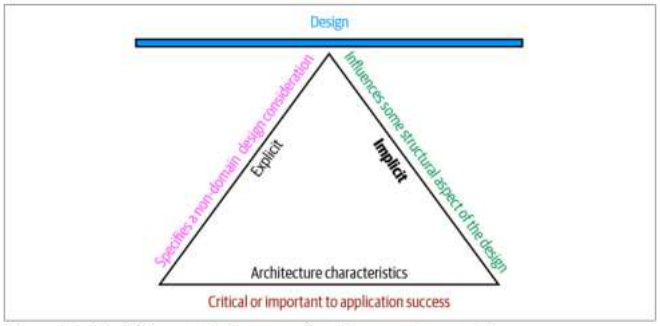


Figure 4-2. The differentiating features of architecture characteristics

- - Arquiteto deve escolher o mínimo de características possível
  - Cada uma adiciona complexidade
- Características:
  - Implícitas
    - Confiabilidade, segurança, está pressuposto em qualquer aplicação
  - Explícitas - Listas (incompletas) de características:
  - **Características Operacionais** (próximo a DevOps)

Table 4-1. Common operational architecture characteristics

Term	Definition
Availability	How long the system will need to be available (if 24/7, steps need to be in place to allow the system to be up and running quickly in case of any failure).
Continuity	Disaster recovery capability.
Performance	Includes stress testing, peak analysis, analysis of the frequency of functions used, capacity required, and response times. Performance acceptance sometimes requires an exercise of its own, taking months to complete.
Recoverability	Business continuity requirements (e.g., in case of a disaster, how quickly is the system required to be on-line again?). This will affect the backup strategy and requirements for duplicated hardware.
Reliability/ safety	Assess if the system needs to be fail-safe, or if it is mission critical in a way that affects lives. If it fails, will it cost the company large sums of money?
Robustness	Ability to handle error and boundary conditions while running if the internet connection goes down or if there's a power outage or hardware failure.
Scalability	Ability for the system to perform and operate as the number of users or requests increases.

  - **Características Estruturais** (estrutura do código)

Table 4-2. Structural architecture characteristics

Term	Definition
Configurability	Ability for the end users to easily change aspects of the software's configuration (through usable interfaces).
Extensibility	How important it is to plug new pieces of functionality in.
Installability	Ease of system installation on all necessary platforms.
Leverageability/ reuse	Ability to leverage common components across multiple products.
Localization	Support for multiple languages on entry/query screens in data fields; on reports, multibyte character requirements and units of measure or currencies.
Maintainability	How easy it is to apply changes and enhance the system?
Portability	Does the system need to run on more than one platform? (For example, does the frontend need to run against Oracle as well as SAP DB?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Upgradeability	Ability to easily/quickly upgrade from a previous version of this application/solution to a newer version on servers and clients.

  - **Características Transversais** (outros)

Table 4-3. Cross-cutting architecture characteristics

Term	Definition
Accessibility	Access to all your users, including those with disabilities like colorblindness or hearing loss.
Archivability	Will the data need to be archived or deleted after a period of time? (For example, customer accounts are to be deleted after three months or marked as obsolete and archived to a secondary database for future access.)
Authentication	Security requirements to ensure users are who they say they are.
Authorization	Security requirements to ensure users can access only certain functions within the application (by use case, subsystem, webpage, business rule, field level, etc.).
Legal	What legislative constraints is the system operating in (data protection, Sarbanes Oxley, GDPR, etc.)? What reservation rights does the company require? Any regulations regarding the way the application is to be built or deployed?
Privacy	Ability to hide transactions from internal company employees (encrypted transactions so even DBAs and network architects cannot see them).
Security	Does the data need to be encrypted in the database? Encrypted for network communication between internal systems? What type of authentication needs to be in place for remote user access?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Usability/ achievability	Level of training required for users to achieve their goals with the application/solution. Usability requirements need to be treated as seriously as any other architectural issue.

- Definições nem sempre são 100% objetivas
    - Algumas se sobrepõem
- ISO (international Standards Organization) tem uma lista própria
    - Não é unanimidade
- Não se pode otimizar todas as características ao mesmo tempo
- Trade-offs
    - E.g. desempenho VS segurança
- Procurar a arquitetura menos pior
- Projetar a arquitetura iterativamente é bom

05 - Identificando as Características de Arquitetura

- Determinar o que realmente importa
  - Depende do domínio e stakeholders
- Entender principais objetivos
- Ter a menor lista possível
- Caso do barco Vasa
  - Mega barco sueco que tinha tudo e afundou
- Forma de selecionar:
  - Cada stakeholder selecionar três características (e não escolher livremente)
    - Gera consenso e revela prioridades
- Deve haver uma tradução linguagem técnica <--> linguagem business

Table 5-1. Translation of domain concerns to architecture characteristics

Domain concern	Architecture characteristics
Mergers and acquisitions	Interoperability, scalability, adaptability, extensibility
Time to market	Agility, testability, deployability
User satisfaction	Performance, availability, fault tolerance, testability, deployability, agility, security
Competitive advantage	Agility, testability, deployability, scalability, availability, fault tolerance
Time and budget	Simplicity, feasibility

  - - Importante atentar aos pacotes de características:
  - Agilidade sem testabilidade costuma ser um erro
  - Nem todas as características ideias são óbvias do pedido do stakeholder
- Conhecimento do domínio ajuda a definir características
  - Nem sempre estão escritas
- Exemplo do Sanduíche
- Primeiro passo:
  - Separar explícito do implícito
  - Explíticas:
    - Número de usuário é uma variável importante
    - Características presentes e futuras
  - Implícitas:
    - E.g. disponibilidade
    - Segurança é default, mas quão crítico?
    - Risco de ter especificações demais
- Elasticidade (crescimento cíclico) != Escalabilidade (crescimento futuro)
- Características interagem entre si
- "Não existe resposta errada na arquitetura, só as caras"
- O que vai em design e o que vai em arquitetura?
  - Trade-off
  - E.g. customização fica em qual nível? Sistema ou aplicação?
- Exercício:
  - Tentar eliminar uma característica para detectar menos importante

06 - Medindo e Controlando as Características de Arquitetura

- Nem toda característica é concreta (e.g. "implementabilidade")
- Mesma característica pode ter várias definições
- Podem ser muito amplas (e.g. agilidade)
    - Definições devem ser OBJETIVAS! - mensurável

Operacionais

- Várias formas:
    - E.g. tempo de resposta:
        - Tempo médio?
        - Tempo máximo?
- Base deve ser baseada em estatística, não arbitrário
- Podem evoluir ao longo do tempo

Estrutura

- Não há métrica completas de qualidade do código
    - Complexidade (Complexidade Ciclomática)
        - Baseado em fragos
        - Mede caminhos que existem no código
        - Complexidade só deve existir para problemas inerentemente complexos
        - Referência: CC < 10 ou então CC < 5
- TDD tende a evitar complexidade

Processo

- Testabilidade e.g. cobertura de testes - cruza com desenvolvimento
- Implementabilidade - taxa de erros na implementação
- Dependem da equipe para saber o que importa
- Agilidade pode informar outras características como modularidade, por exemplo.

Governança e Funções de Aptidão

- Assegurar qualidade de software faz parte da governança
- Orientação
- "Building Evolutionary Architecture"
- Função de aptidão:
    - Função para avaliar proximidade do alvo
    - Várias formas
    - Como um score da arquitetura
        - Servem para muitas coisas
    - E.g. criar funções de aptidão para verificar ciclos entre dependência
    - Limites para distância da sequência principal
    - Teste se camadas estão sendo respeitadas numa arquitetura por camadas
    - E.g. Macaco do Caos - programa verificador constante de governança

07 - Escopo das Características da Arquitetura

- Os componentes externos também são objeto de atenção e métricas
    - Mas difícil de medir
- Quantum de Arquitetura
    - Artefato independente com alta coesão funcional e conascência síncrona
        - Mínimo conjunto funcional
        - Uma unidade da arquitetura
- Conascência de Comunicação
    - DDD - Domain Driven Design - contexto delimitado por domínio
    - Cada domínio faz o seu e depois se resolve na integração, mesmo que haja ideias repetidas.
- Exemplo do Leilão
    - Escala nacional - número de usuários
        - Escalabilidade e elasticidade
        - Desempenho é central
    - Segurança deve ser uma característica especial?
    - Índice de reputação requer conhecimento do domínio
        - Fora do escopo da arquitetura
    - M&As constantes podem definir escolhas por soluções menos específicas e mais interoperáveis
    - Características da arquitetura não existem como um todo mas para as partes do sistema
        - Disponibilidade do leiloeiro é mais importante do que dos usuários.
    - Escopo granular da arquitetura
    - Cracterísticas deve ser por quantum, não o todo
    - 3 Escopos:
        - Feedback do proponente
        - Leiloeiro
        - Proponente

08 - Pensamento Baseado em Componentes

- Componente - como um empacotamento
- Agrupar artefato
    - Library - conjunto de classes/funções
    - Subsistema
    - Serviço - comunica via TCP/IP por exemplo

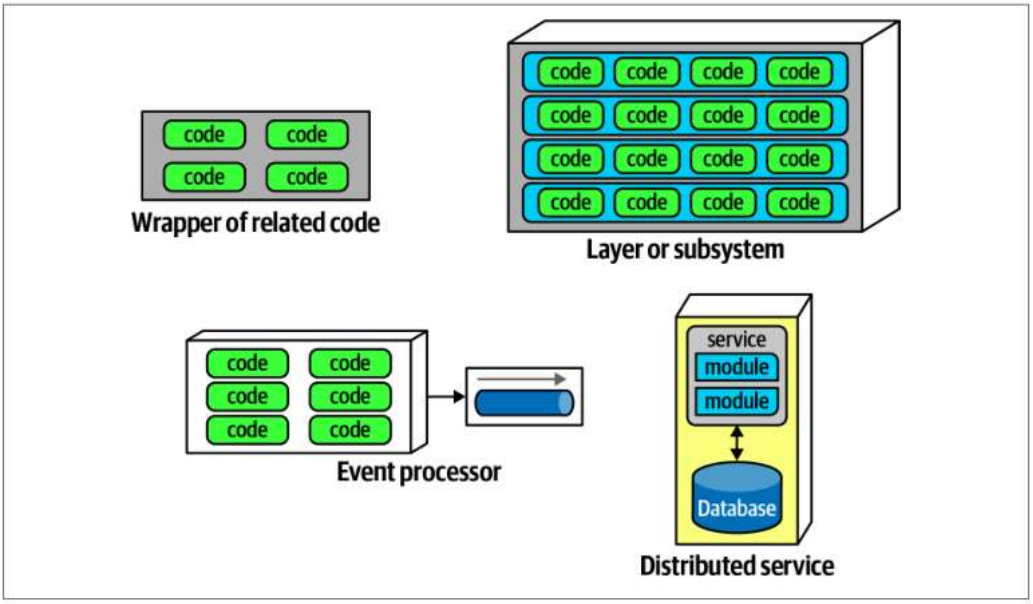
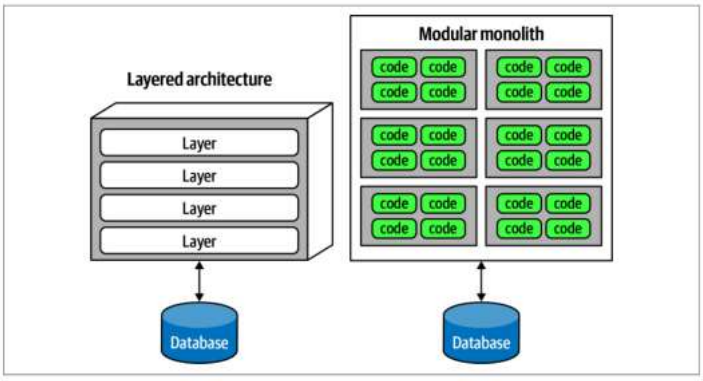


Figure 8-1. Different varieties of components

- Componente pode ser extremamente simples
- Arquitetura em geral independe do desenvolvimento
- Componente é o nível mais básico que o arquiteto lida
    - (exceto para tirar métricas de código, em que lida com código)
- A forma de juntar esses componentes é um trade-off
    - E.g. monolítico em camadas
    - Monolítico modular, etc.



- *Figure 8-3. Two types of top-level architecture partitioning: layered and modular*
- Cada componente pode ter outros componentes
- Particionamento a nível:
    - Domínio?
        - Organizado mas pode ter repetições
    - Técnico?

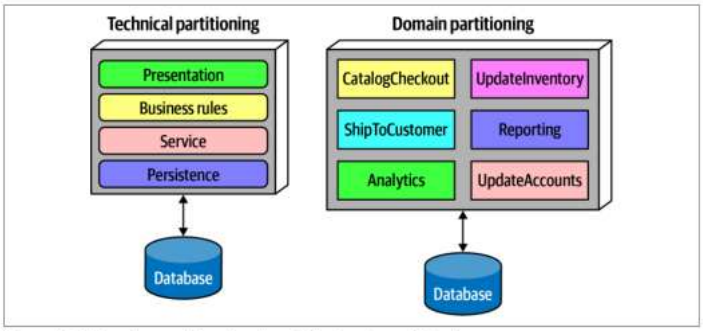


Figure 8-4. Two types of top-level partitioning in architecture

  - Pode ter operações que aparecem em todas as camadas - Arquitetura em camadas é bem comum

- **Lei de Conway -** Arquitetura imita estrutura de comunicação da própria empresa
- Por domínio parece ser tendência, mas é um trade-off

Estudo de Caso: Silicon Sandwiches

- Duas opções, técnicamente e por domínio:

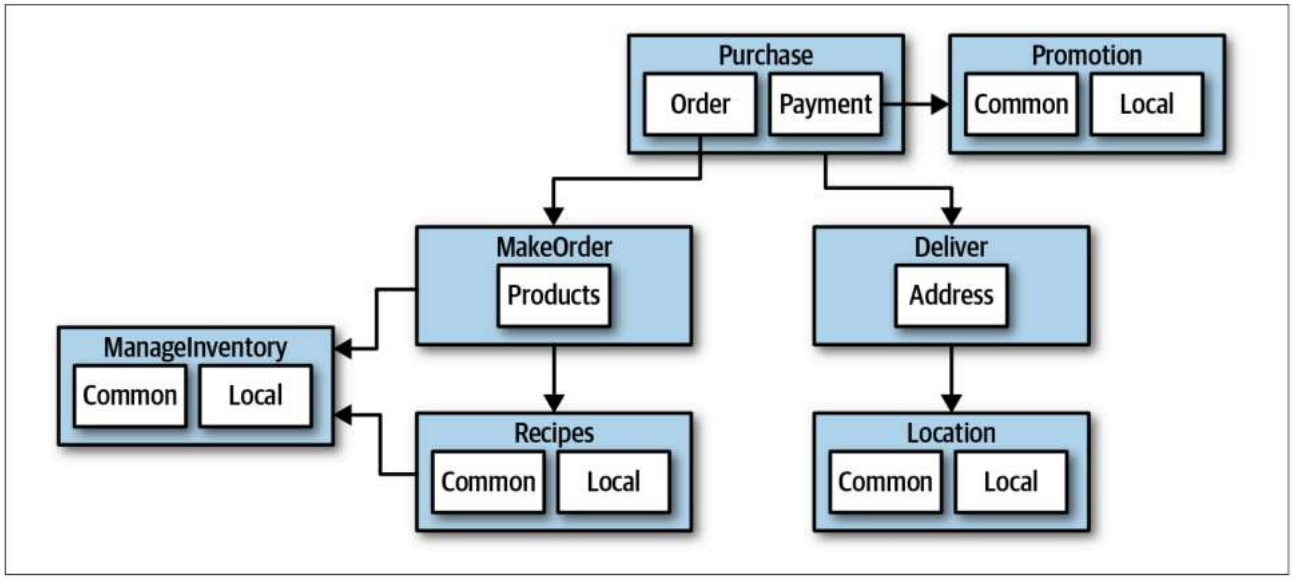


Figure 8-6. A domain-partitioned design for Silicon Sandwiches

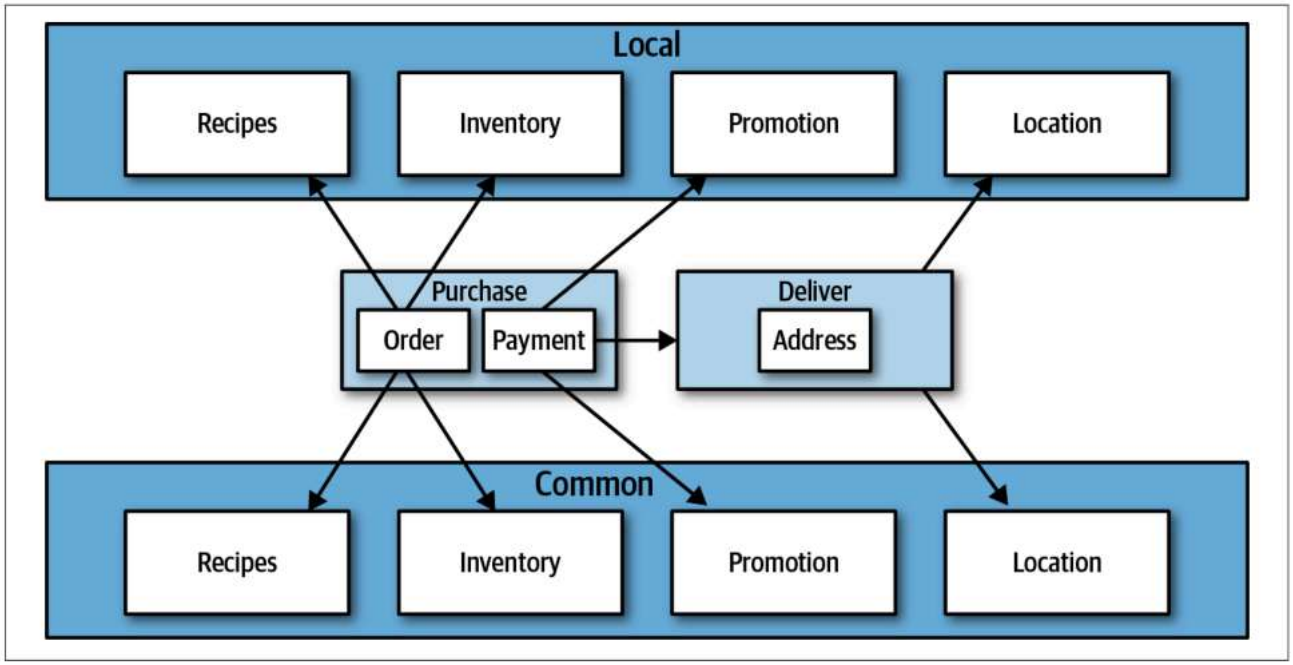


Figure 8-7. A technically partitioned design for Silicon Sandwiches

- Particionamento por domínio:
  - Vantagens:
    - Representa melhor funcionamento do negócio
    - Facilita criar equipes multifuncionais
    - Combina bem com arquitetura monolítica modular e de microsserviços
    - Facilita migrar dados e componentes para a arquitetura distribuída
  - Desvantagens:
    - Mas código pode ficar repetido em vários domínios
- Particionamento técnico:
  - Vantagens:
    - Separa bem o código
    - Combina bem com arquitetura em camada
  - Desvantagens:
    - Alto grau de acoplamento, toda alteração afeta tudo.
    - Pode haver duplicação de "local" e "common" mesmo assim
    - Alto acoplamento de dados, difícil migração.

- Desenvolvedores dividem cada componente definido pelo arquiteto em classes e funções.
  - Essa divisão é responsabilidade de ambos arquitetos e desenvolvedores.

    - Devem interagir e iterar sobre o design projetado.
    - Três etapas em Loop:
        - Analisar funções e responsabilidades
        - Analisar as características da arquitetura
            - Cada componente tem uma especificação
                - E.g. se dois componentes lidam com login de usuários mas em quantidades diferentes, suas especificações devem ser diferentes
        - Reestruturar os componentes
            - Incorporar feedback dos desenvolvedores

- Dificilmente o design inicial será bom

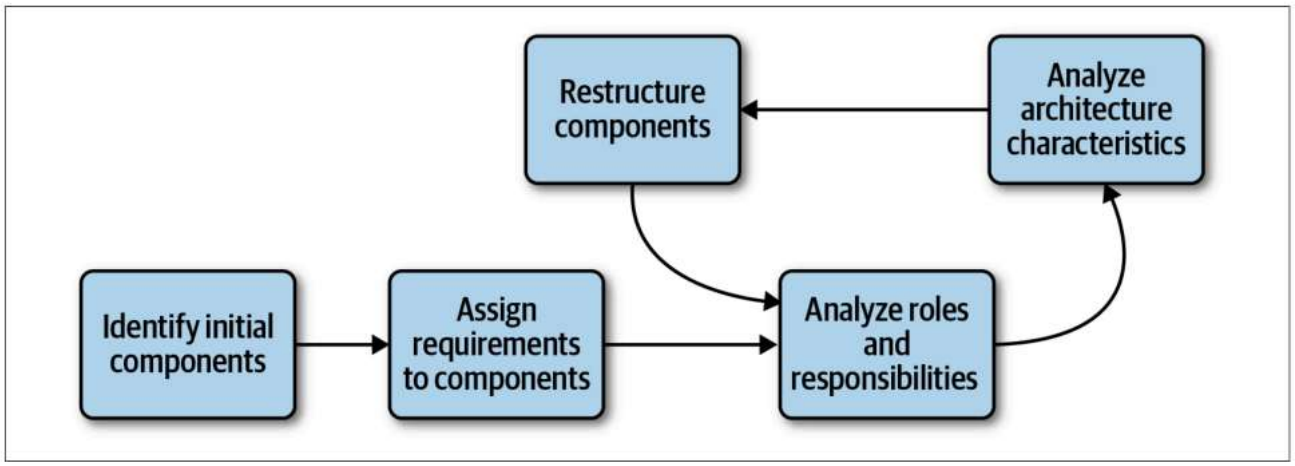


Figure 8-8. Component identification cycle

-
-

- Definir a granularidade ideal do componente é importante e difícil.
    - Pouco granular --> Alto acoplamento interno, difícil implementar e testar
    - Muito granular --> Muita comunicação entre os componentes

- Como definir os componentes?
    - **Armadilha da entidade**:
        - Antipadrão. Confundir arquitetura com entidades de um banco de dados relacional.
        - Confundir relações do banco de dados como fluxo de trabalho na aplicação.
            - (Naked Objects - família de frameworks para criar CRUDs simples)
    - Abordagem *ator/ações*:
        - Identificar os quem e os quês que são feitos.
    - Event Storming:
        - Pressupõe-se o uso de mensagems/eventos.
        - Tenta-se descobrir quais eventos ocorrem e cria componentes em torno disso.
    - Abordagem do fluxo de trabalho:
        - Como um event storming genérico.
        - Imaginam-se os fluxos de trabalho e criam-se componentes baseados nisso.

- Exemplo: Caso do Leilão
    - Abordam ator/ações é genérica e funciona bem
    - Ações:
        - Proponente (bidder)
            - Exibe vídeo ao vivo, faz um lance, etc.
        - Leiloeiro
            - Insere os lances, recebe os lances etc
        - Sistema
            - Inicia o leilão, faz o pagamento, etc.
    - Com isso, podemos imaginar a seguinte arquitetura com os componentes:

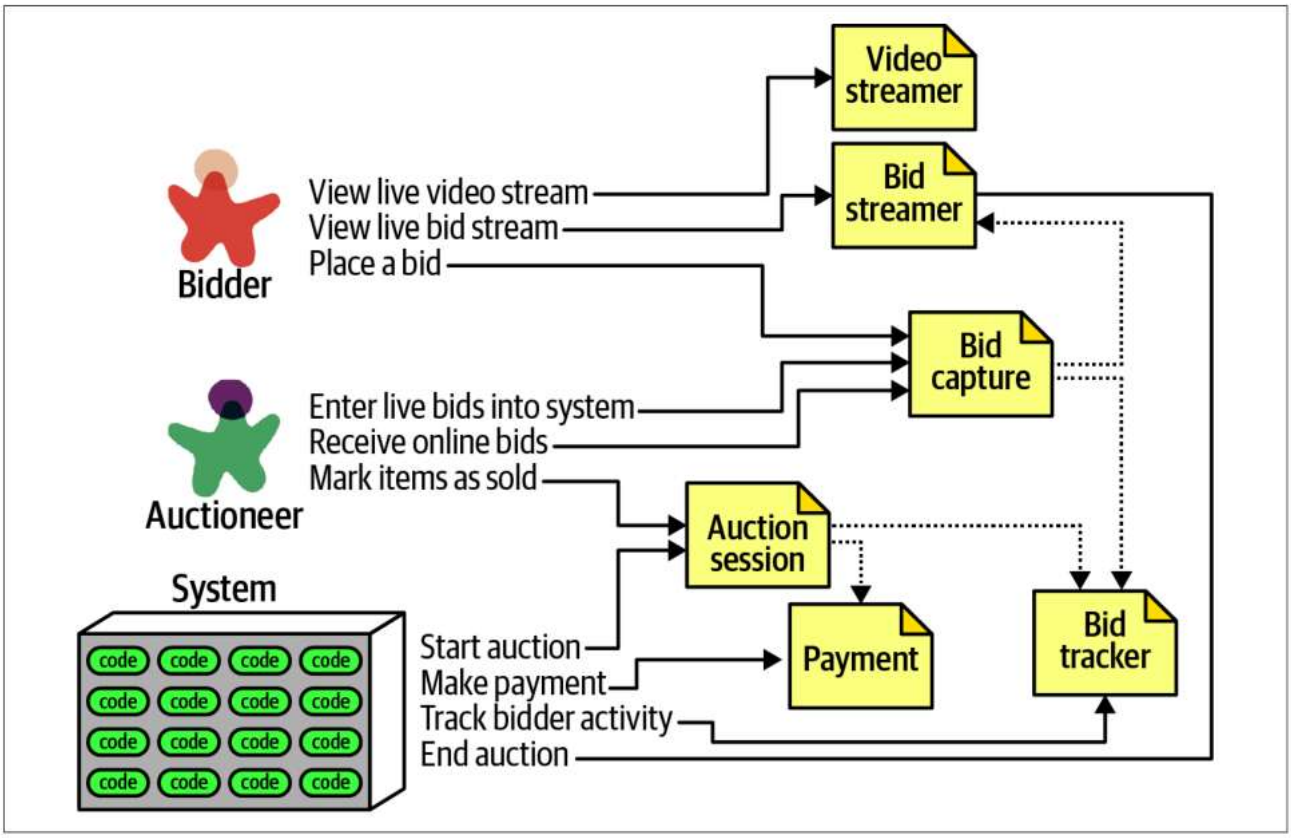


Figure 8-10. Initial set of components for Going, Going, Gone

- A partir disso podemos pensar em como essa estrutura funcionaria, analisando as características.
- Elasticidade do sistema para leiloreiros precisa ser muito menor do que para proponentes.
- Confiabilidade dos sistemas para o leiloeiro é mais importante do que dos proponentes.
    - Têm características diferentes.
    - Sendo assim, podemos dividir o item "Bid Capture" em um componente para o leiloeiro e um para o proponente.
    - O componente "Bid Tracker" agregará ambos.

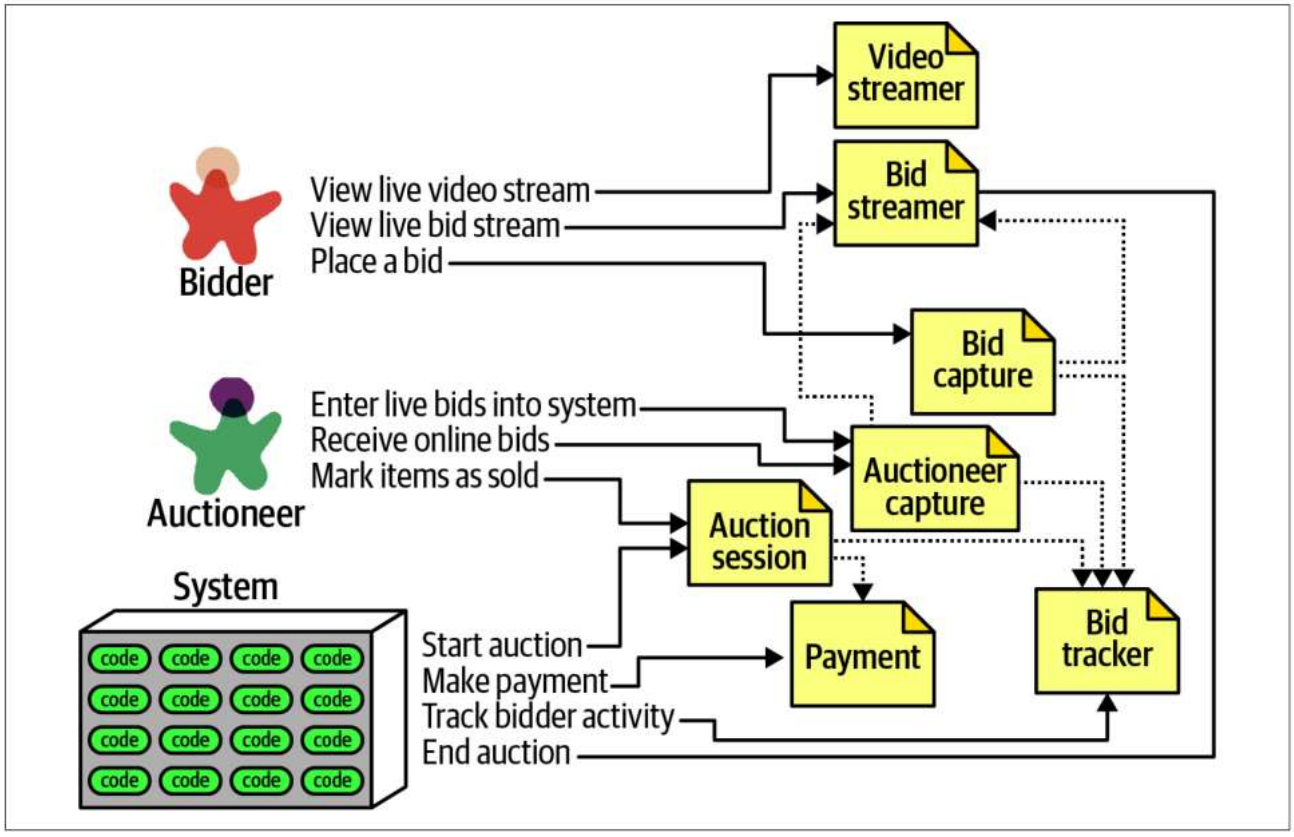


Figure 8-11. Incorporating architecture characteristics into GGG component design

- Existem muitas formas de resolver este problema, nunca há uma solução única.
- A arquitetura deve ser monolítica ou distribuída?
  - Monolítica:
    - Uma única unidades de implementação
    - Um único banco de dados, em geral
  - Distribuída:
    - Vários serviços comunicando-se.
    - Mais independentes.
  - Um critério importante pra escolher é a quantidade de "Quantum de arquitetura" (i.e. conjunto de critérios de arquitetura).
    - Se um quantum bastar, estrutura monolítica costuma ser uma boa escolha.
    - Se requerer muitas quantums, estrutura distribuída pode fazer mais sentido.
    - No exemplo do leilão:
      - Características diferentes de componentes de *Capture* apontam para arquitetura distribuída.
      - Os dois streamers com requisitos diferentes (read-only intenso vs read/write constante) também apontam para distribuído.

09 - Fundamentos

2025-12-12 10:12 am

Padrões podem ter subpadrões contidos em padrões de arquiteturas maiores.

Padrões Fundamentais

Grande Bola de Lama

- Antipadrão
- Sem organização e hierarquia
- Crescem sem ordem, às vezes começa assim como pequeno.
- Deve ser evitado a todo custo.
- Difícil de prever efeito de mudanças

Arquitetura Unitária

- Software e hardware eram uma coisa só no início
- Não havia rede de computadores para distribuir tarefas.
- Raros hoje em dia, exceto em sistemas embarcados e afins.

Cient/Servidor

- Separação em dois níveis: cliente e servidor, backend e front-end
- Tem muitas variações:
  - **Desktop + servidor de banco de dados**
    - Apresentação no desktop, computação intensiva no servidor
  - **Browser + Web Server**
    - Separação de responsabilidades também
  -

Três Níveis

- Popular nos anos 90.
- Database, Aplicação e Frontend
- Protocolos CORBA e DCOM facilitaram arquiteturas distribuidas (análogo a TCP/IP)
- Motivou requisito de serialização de objetos em Java, inexistente em C++, para facilitar transferência na rede. Hoje ninguém usa isso e é um fardo de retrocompatibilidade.

Arquiteturas Monolíticas Vs Distribuídas

- Dois grandes tipos.
- Código em um único deploy ou espalhado.
Monolíticos	**Distribuídos**
Em Camadas	Baseado em Serviços
Pipeline	Event-driven
Microkernel	Baseado em espaço
	Orientada a serviços
	Microsserviços
- Distribuídos costumam ser mais escaláveis e performáticos mas há desvantagens.
- **Falácia #1 - A Rede é Confiável**
  - Rede melhorou com o tempo mas ainda é um ponto de falha
  - Comunicação entre dois serviços perfeitamente saudáveis pode falhar, requer cuidados.
- **Falácia #2 - Latência é Zero**
  - Conexão pela rede é sempre mais lenta do que comunicação entre componentes locais.
  - Qual a latência de uma requisição ida/volta para um RESTful? Necessário saber.
  - Latência pode ir se somando a cada chamada, se tornando alta.
- **Falácia #3 - A Banda é Infinita**
  - Tamanho das requisições podem se somar até ser massivas.
  - *Stamp coupling* - obrigatoriedade pelo sistema de enviar muitos dados quando na verdade só se quer um pedacinho.
    - Stamp Coupling pode ser evitado selecionando campos, usando GraphQL para desacoplar dados ou ter endpoints específicos

- **Falácia #4 - A Rede é Segura**
    - Sistemas distribuídos requerem mais cuidados com segurança.
    - Cada endpoint é um risco, inclusive entre serviços.
- **Falácia #5 - A Topologia Nunca Muda**
    - Rede vai mudando com o tempo.
    - Upgrades no software de rede e ajustes podem desencadear problemas e requerer mudanças.
- **Falácia #6 - Só há um administrador**
    - Sempre há várias pessoas gerindo várias partes do projeto.
        - e.g. administrador de redes.
    - Requer coordenação que aplicações monolíticas não precisam.
- **Falácia #7 - Custo de Transporte é Zero**
    - Arquitetura distribuída custa mais que monolítica em geral.
    - Requer mais elementos de rede e hardware.
- **Falácia #8 - A Rede é Homogênea**
    - Hardwares de rede são diferentes e podem ter especificações diferentes dentro de si, com equipamento de marcas diferentes.
- Arquitetura distribuída também dificulta o logging e descobrir a causa de problemas.
    - Há ferramentas que consolidam diversas fontes, mas não resolvem tudo.
- Consistência de dados é mais difícil em sistemas distribuídos, mais difícil garantir que os mesmos dados estão presentes a todo momento em todas as partes do sistema (*eventual consistency*)
    - ACID vs BASE
- Manutenção de contratos entre cliente e serviço também é um ponto de dor em sistemas distribuídos. Também complexidade em organizar deprecação de versões.

10 - Arquitetura em Camadas

2025-12-20 19:20 pm

- Um dos estilos mais comuns
  - Simples e barato
  - Segue a lei de Conway, copia a comunicação da empresa
  - Às vezes acaba ocorrendo por acidente (anti-pattern)

Topologia

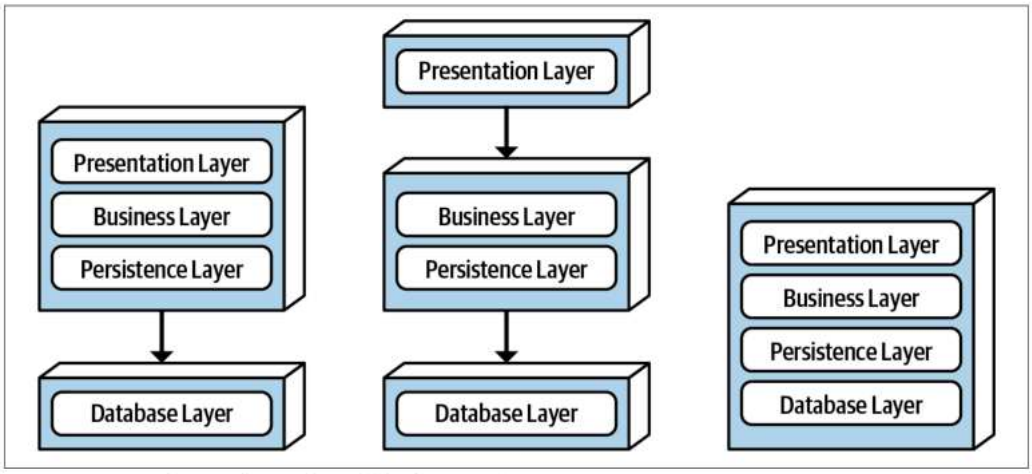


Figure 10-2. Physical topology (deployment) variants

- Organizado em camadas lógicas horizontais, cada camada tem uma responsabilidade.
- Geralmente quatro camadas padrão:
  - **Apresentação**
    - e.g. lidar com interação do usuário
  - **Negócio**
  - **Persistência**
    - Pode estar fundida à de negócio
  - **Database**
- Tem uma série de variações, combinando ou não as diferentes camadas.
- Produtos *on-prem* costumam combinar todas.
- Cada camada não se preocupa com as demais
  - e.g. camada de negócio não se preocupa com o formato de visualização
- Arquitetura particionada por técnica (e não pode domínio)
  - agrupamento de elementos por papel técnico
  - Ou seja, domínio se espalha por toda a arquitetura
    - e.g. todas as camadas têm que ter informações a respeito de o que é um Customer.
    - Incompatível com domain-driven design
- Difícil de ser alterada com agilidade.

Camadas de Isolamento

- Camada pode ser aberta ou fechada.
  - **Fechada**:
    - Se comunica só com as camadas adjacentes na hierarquia
  - **Aberta:**
    - Se comunica com qualquer camada sem passar pelas que não precisam
    - e.g. Camada de apresentação talvez precise acessar DB diretamente.
- Interdependência entre camadas causa acoplamento.
- Camadas fechadas facilitam isolamento e diminuem número de mudanças necessárias.

Adicionando Camadas

- Às vezes faz sentido ter camadas abertas.
  - E.g. quando um componente é compartilhado pela camada de apresentação e de negócio.
  - Seria ruim se a camada de apresentação demandasse um componente (e.g. dateutil) que pertence à camada de negócio)

Outras Considerações

- Costuma ser um bom começo de aplicação.
  - Depois é alterada para algo mais robusto.
- Manter uma hierarquia de objetos rasa e reuso de objetos ajuda migração futura.

- Respeitar a arquitetura em camadas só por respeitar mas elas não realizam nenhuma função, só a formalidade da hierarquia, é um *anti-pattern* - *architetural sinkhole*.
    - Um pouquinho disso acaba sendo difícil de evitar, mas muito é um problema.
    - Solução de abrir todas as camadas para evitar este *anti-pattern* gera outras problemas

Por que Usar este Estilo de Arquitetura

- Bom para pequenas aplicações simples ou websites, ou como ponto de partida.
- Bom quando há incerteza sobre a melhor arquitetura.
- Deve ser evitado para qualquer aplicação um pouco maior.

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★
Elasticity	★
Evolutionary	★
Fault tolerance	★
Modularity	★
Overall cost	★★★★★
Performance	★★
Reliability	★★★
Scalability	★
Simplicity	★★★★★
Testability	★★

- Principal benefício é custo e simplicidade.
- Toda mudança requer muito cuidado no código e é difícil de testar.

11 - Arquitetura de Pipeline

2025-12-20 19:51 pm

- Pipeline ou *pipe and filters*
- Princípio usado em bash ou zsh
- Parece programação funcional - análogo ao modelo MapReduce

Topologia

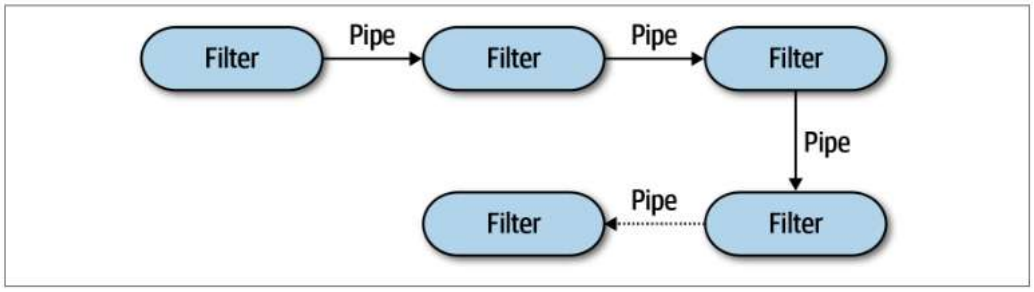


Figure 11-1. Basic topology for pipeline architecture

- **Pipe**
  - Canal de comunicação entre os filtros
  - Unidirecional e de ponta a ponta.
  - Preferível que o input seja pequeno
- **Filters**
  - São autocontidos e independentes dos outros filtros
  - Stateless
  - Um filtro deve executar uma única tarefa
  - Quatro tipos de filtros:
    - **Producer:**
      - Source, ponto de partida
    - **Transformer:**
      - Transforma um input e o passa ao destino. Um **map**
    - **Tester:**
      - Aceita input, testa algum(ns) critérios e gera ou não um output. Um **reduce**
    - **Consumer:**
      - Ponto de destino. Pode persistir resultado em um database e/ou exibir resultado.
  - Simplicidade dos componentes favorece reuso.

Exemplo

- ETL costumam usar esta arquitetura

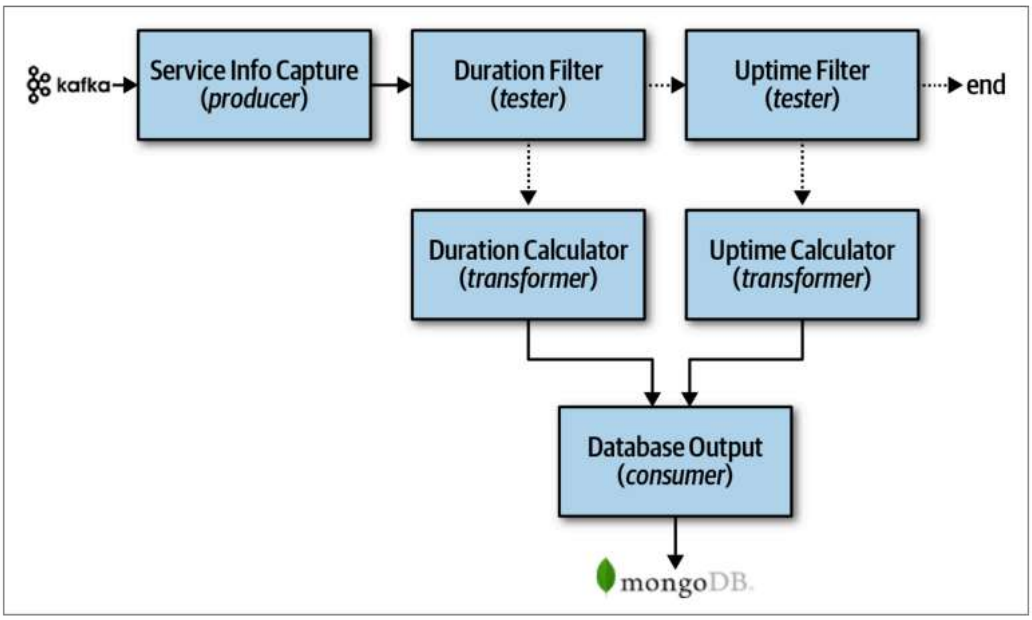


Figure 11-2. Pipeline architecture example

  - Neste caso, um novo tester poderia ser facilmente inserido após o uptime filter.

Classificação

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★★
Elasticity	★
Evolutionary	★★★
Fault tolerance	★
Modularity	★★★
Overall cost	★★★★★
Performance	★★
Reliability	★★★
Scalability	★
Simplicity	★★★★★
Testability	★★★

- Estilo tecnicamente particionado.
- Costuma ser um deploy monolítico, um só quantum de arquitetura.
- Pontos fortes são simplicidade, custo e modularidade (mas ainda é um monolito)
- Testes requerem teste de todo o monolito.
- Pouco escalável, desacoplar desempenho das peças é muito complexo.
- Erro de um pedaço causa erro em tudo.

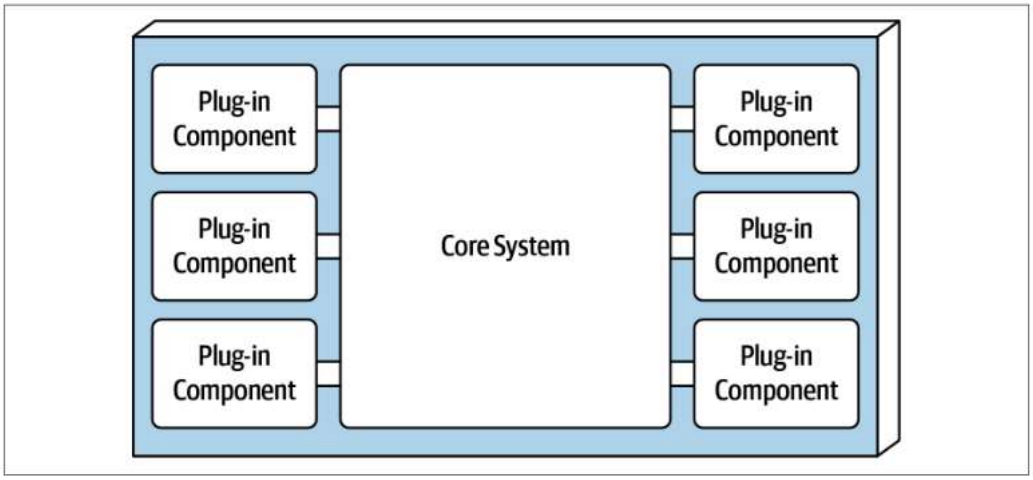
12 - Arquitetura de Microkernel

2025-12-20 20:10 pm

- Também chamado de arquitetura de *plug-in*
- Combina com software como produto.

Topologia

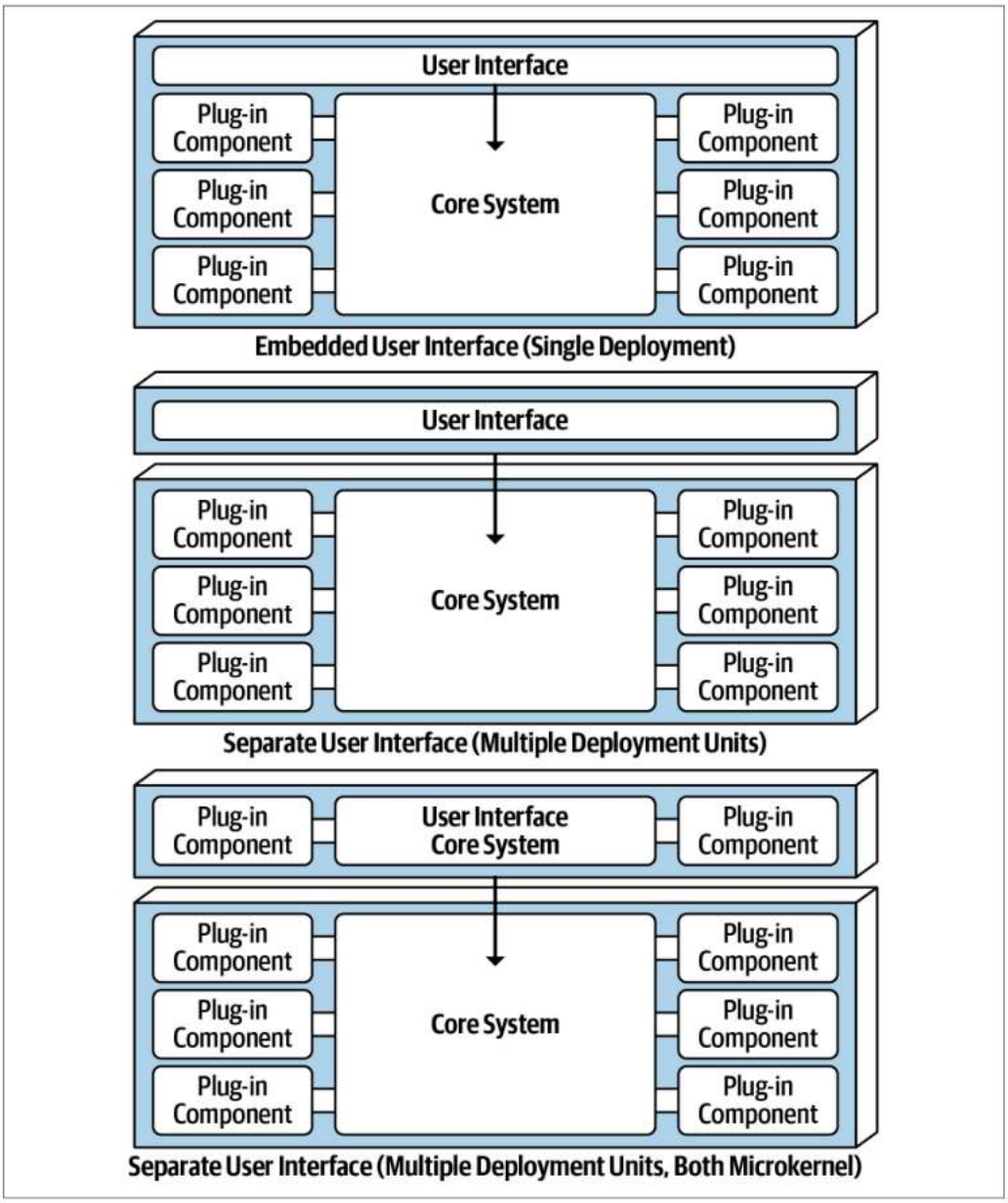
- Monolítico
- Relativamente simples
- Dois componentes:
  - **Core**
  - **Plug-in**s



- *Figure 12-1. Basic components of the microkernel architecture style*

Sistema Core

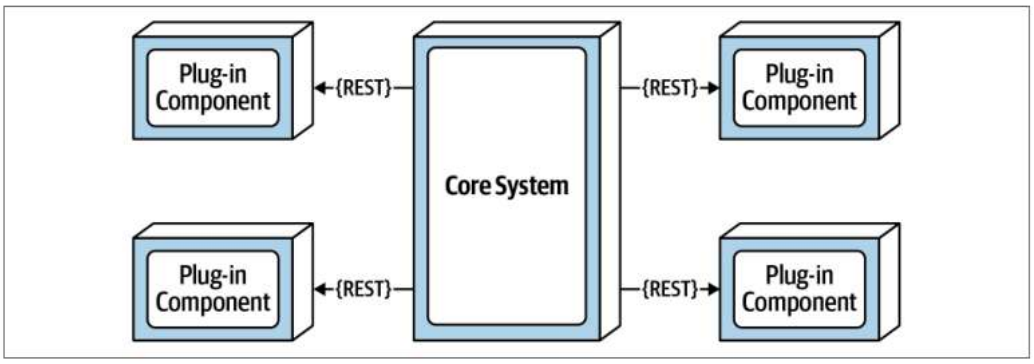
- Funcionalidade mínima para rodar o sistema. Ou então o menor *caminho feliz*.
  - e.g. numa IDE, o core pode ser um simples editor de texto, os plug-ins o complementam.
- Delega funcionalidades específicas aos plug-ins.
  - Plug-ins são independentes.
- Sistema core pode ter arquitetura em camadas ou um monolito modular. Ou mesmo distribuído.
- Core pode ser particionado tecnicamente ou por domínio.
- Pode interagir ou outros componentes eles próprios estruturados em microkernel.



- *Figure 12-3. User interface variants*

Componentes Plug-in

- Componentes independentes que contêm processamento especializado que estende o core system.
- Isola pedaços do código.
  - `app.plugin.<domain>.<context>`
- Plug ins runtime based vs compile based - exige ou não redeploy da aplicação
- Plug in pode ser um serviço avulso que comunica via REST com o Sistema Core.



  - *Figure 12-6. Remote plug-in access using REST*
  - Mais desacoplado e dessincronizado.
  - Mais complexo

- Plug-in não costumam se comunicar com banco de dados central, sistema core gere isso.
    - Mas cada plug-in pode ter seu pequeno datastore particular que só ele acessa.

Registry

- Sistema core precisa saber quais plugins existem e como pegá-los.
    - Registry é uma forma de fazer isso
- Registry tem informações de nome, contrato de dados e endereço do plug-in
- Pode ser simples ou complexo.

```
Map<String, String> registry = new HashMap<String, String>();
static {
  //point-to-point access example
  registry.put("iPhone6s", "Iphone6sPlugin");

  //messaging example
  registry.put("iPhone6s", "iphone6s.queue");

  //restful example
  registry.put("iPhone6s", "https://atlas:443/assess/iphone6s");
}
```

Contratos

- Forma de transferir dados entre plugin e sistema core.
- Costuma ser padrão para um domínio de componentes de plug-in.
- Pode requerer um adapter entre um contrato de plugin não padrão e o contrato de plugin padrão.

Exemplos e Casos de Uso

- Jura, Eclipse IDE, Chrome, Firefox usam essa arquitetura.
- Quando usar para software não-produto?
    - Grande variações de regras de caso para caso pode ser bom ter plugin
    - E.g. sistema de seguros que têm regras diferentes por estado.
        - Permite manipular regras particulares com facilidade.
    - E.g. sistema de imposto que linka formulários diferentes.
        - Facilita mudanças na lei fiscal se cada regra é responsabilidade de um plugin isolado.

Características

Architecture characteristic	Star rating
Partitioning type	Domain and technical
Number of quanta	1
Deployability	
Elasticity	
Evolutionary	
Fault tolerance	
Modularity	
Overall cost	
Performance	
Reliability	
Scalability	
Simplicity	
Testability	

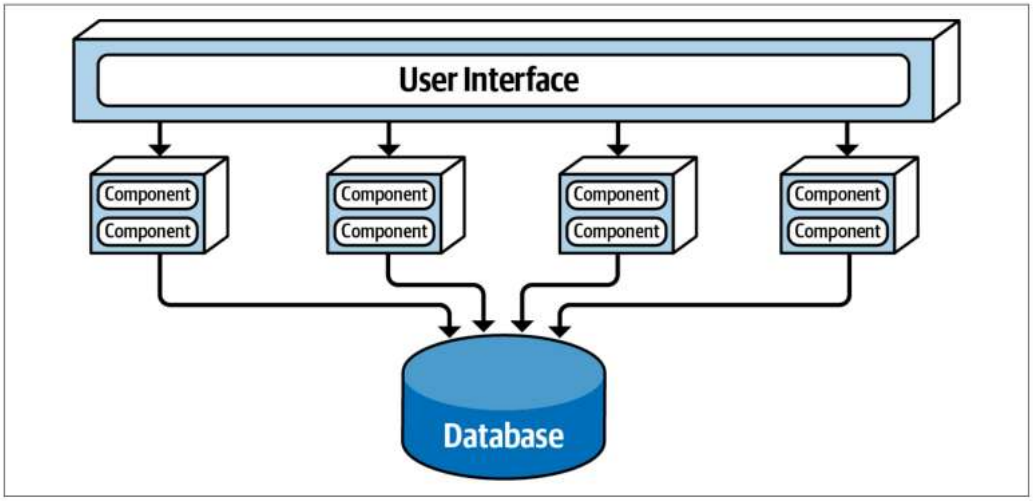
- Simples e barato.
- Monolítico, então pouco escalável e mais difícil de estender e deployar.
- Pode ser tanto particionado por técnica quanto por domínio.
    - Embora geralmente particionado por técnica.
- Altamente customizável.

13 - Arquitetura Baseada em Serviços

2025-12-21 10:23 am

- Híbrido de arquitetura de microsserviços.
- Bem flexível.
- Distribuído mas sem muita complexidade de outros sistema distribuídos.

Topology



- *Figure 13-1. Basic topology of the service-based architecture style*
- Interface separada e serviços individuais não muito granulares, com database único.
  - Serviços de domínio.
- Cada serviço é deployado independentemente.
  - Costuma ter de 4 a 12 serviços.
- Tipicamente um serviço por domínio, mas requisitos podem mudar isso.
- Geralmente REST faz a comunicação entre interface e serviços
  - Mas pode ser RPC ou SOAP.
  - usa *Service Locator Pattern*
- Database único é importante.
  - Mudanças no database podem causar transtornos.

Variantes de Topologia

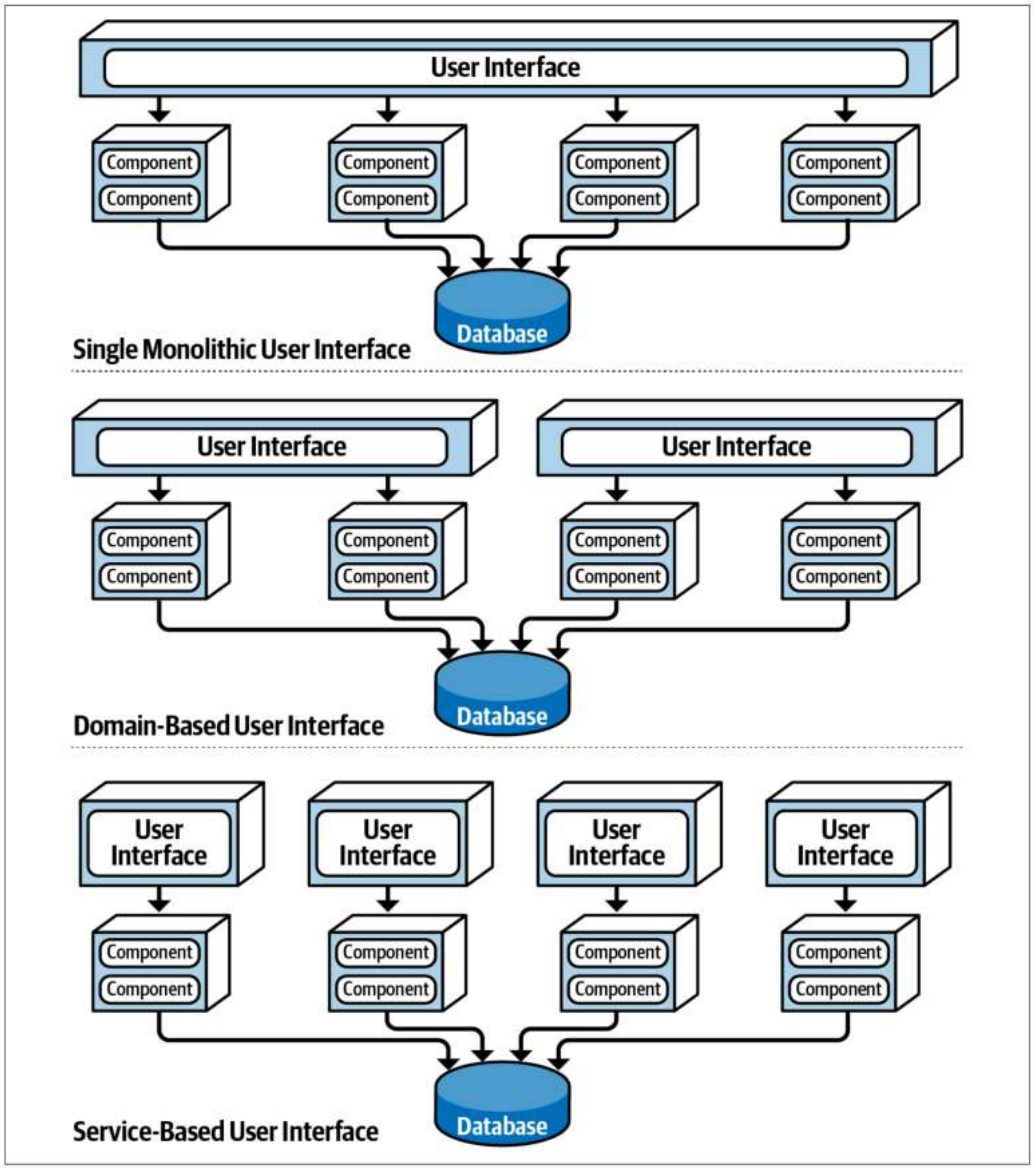


Figure 13-2. User interface variants

- Muitas variações, da forma de separar a interface do usuário.
- Também pode haver mais de um database, separados entre alguns serviços ou para cada serviço individual.
    - Requer garantir que um serviço não precisa dos dados do outro
- Também pode haver uma camada de API como um proxy reverso entre a interface e os serviços.

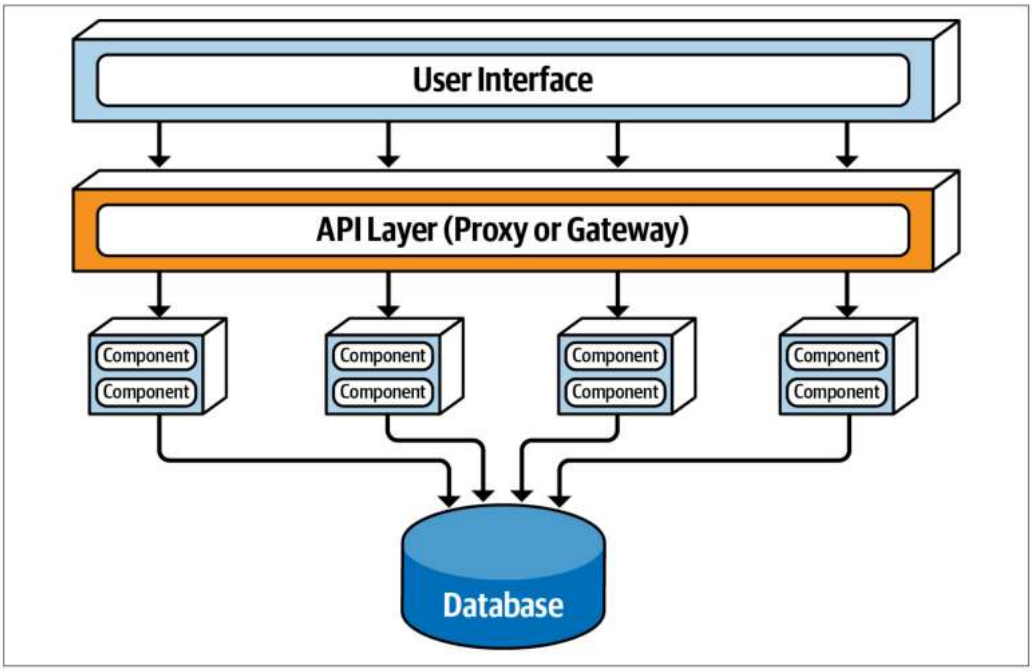


Figure 13-4. Adding an API layer between the user interface and domain services

- **
- Boa prática para expor serviço externamente ou para separar responsabilidades (e.g. monitoramento)

Projeto do Serviço e Granularidade

- Serviços têm escopo mais ou menos largo.
- Cada serviço pode ter arquitetura em camadas.
- Também pode ser internamente separado por domínio.

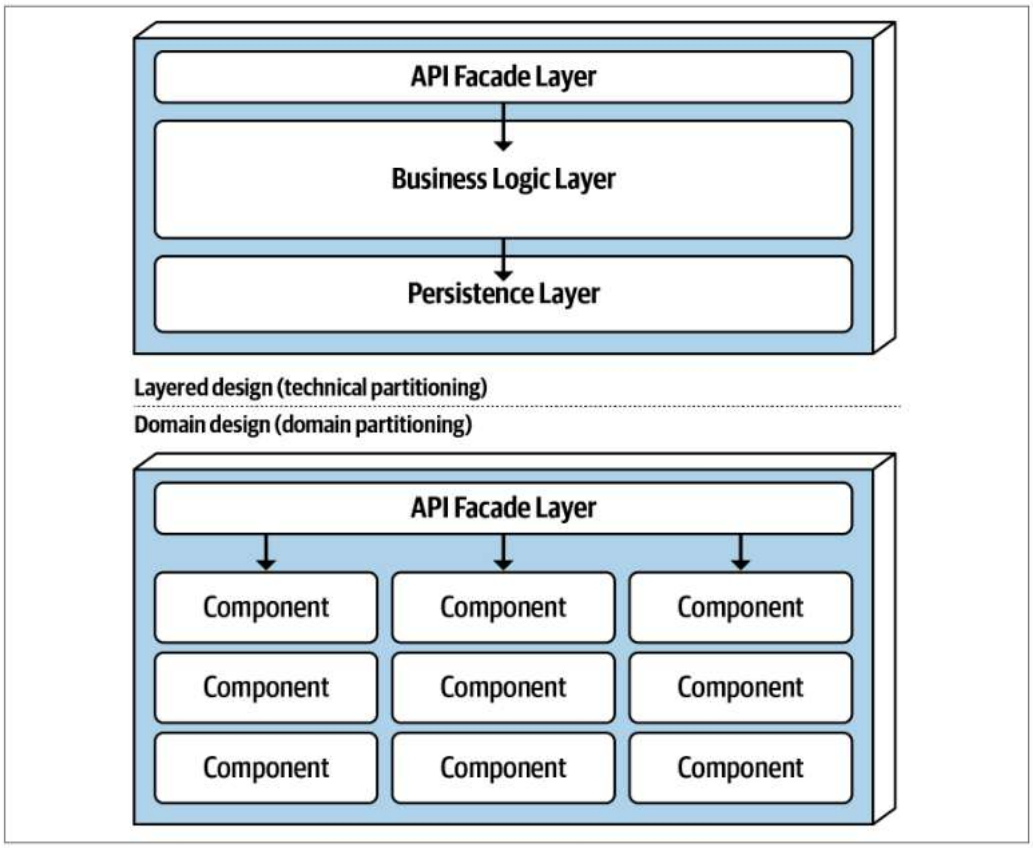


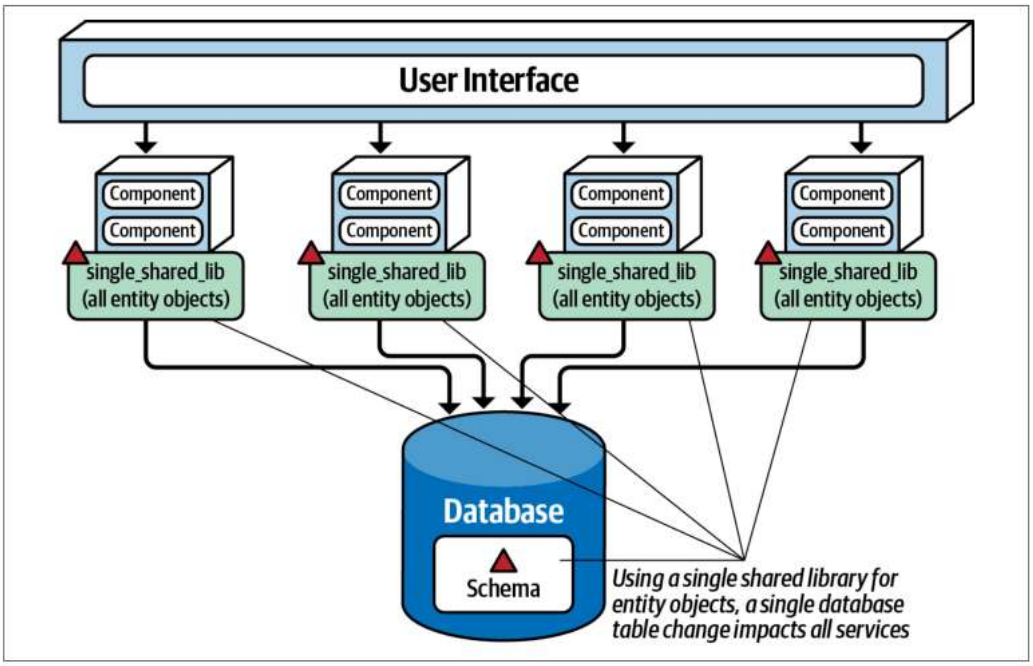
Figure 13-5. Domain service design variants

- Diferença com microsserviços é a granularidade dos serviços.
- Consegue garantir transações ACID.
  - Microsserviços costumam usar BASE (basic availability, soft state, consistência em algum momento)
- Mais fácil de dar rollback dentro de um mesmo serviço.
  - Serviços por domínio facilitam integridade dos dados.

- Mas dificultam mudanças, já que uma mudança pequena dentro de um serviço requer o teste do serviço inteiro.

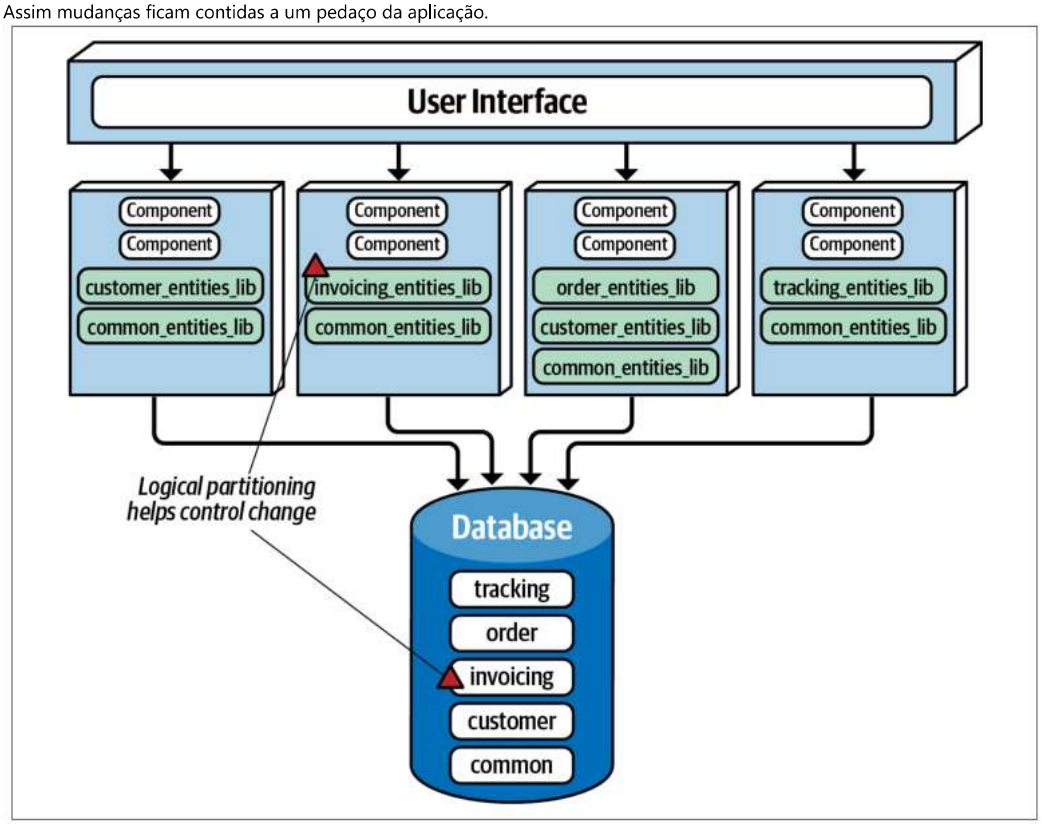
Partição de Banco de Dados

- Database único não é necessário mas é usual.
- Mudança de schema pode alterar todos os serviços.
- Ter um conjunto de componentes compartilhados por todos os serviços que representam o banco de dados é uma alternativa, mas não costuma ser bom, muita complexidade e acoplamento.



  - *Figure 13-6. Using a single shared library for database entity objects*

- Melhor é ter pedaços de entidades compartilhadas, separadas logicamente.
  - Assim mudanças ficam contidas a um pedaço da aplicação.



  - *Figure 13-7. Using multiple shared libraries for database entity objects*
  - Mantendo os domínios bem definidos, quanto mais particionado, melhor.
  - Ainda assim pode ser necessário ter componentes comum a todos (e.g. `common_entities_lib` no exemplo)
    - Mudança nela é bom que seja restrita ao time de database.

Exemplo

- Reciclagem de eletrônicos.
  - Várias etapas entre orçamento, recepção do objeto, checagem de estado, etc.
- Cada etapa (domínio) pode ser um serviço diferente, deployado independentemente.
- Separação entre databases de operações internas e o de informações do usuário, comunicação entre eles unilateral.

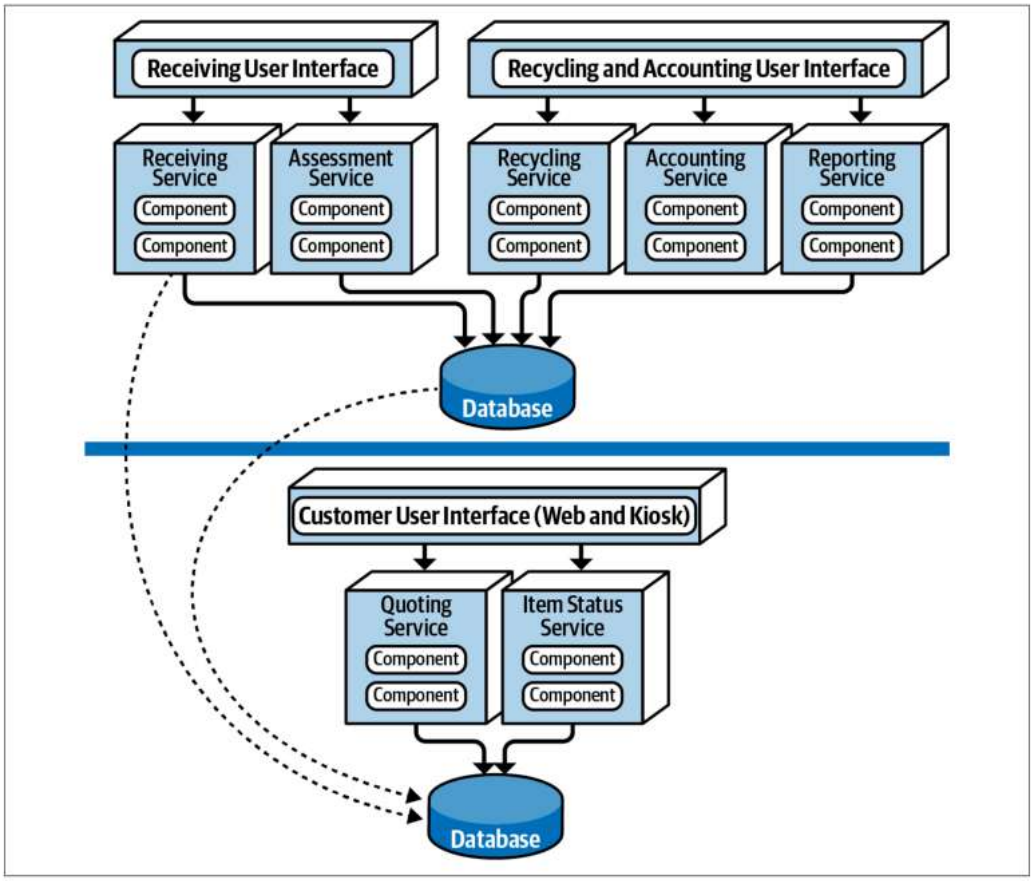


Figure 13-8. Electronics recycling example using service-based architecture

- Escalável, tolerante a falhas, segurança granular.
- Dois quanta de arquitetura (parte interna e a parte externa, cada um com um database)

Características

Architecture characteristic	Star rating
Partitioning type	Domain
Number of quanta	1 to many
Deployability	
Elasticity	
Evolutionary	
Fault tolerance	
Modularity	
Overall cost	
Performance	
Reliability	
Scalability	
Simplicity	
Testability	

- Particionada por domínio.
- Permite vários quanta de arquitetura.
- Bastante boa arquitetura em geral, sem ser excelente em nenhum atributo.
- Serviços são autocontidos e testáveis
- Escalabilidade não é ótima, já que serviços são grosseiros.
- Simples e custo-efetivo.
- Requer pouca banda e comunicação interna, com poucos pontos de falha

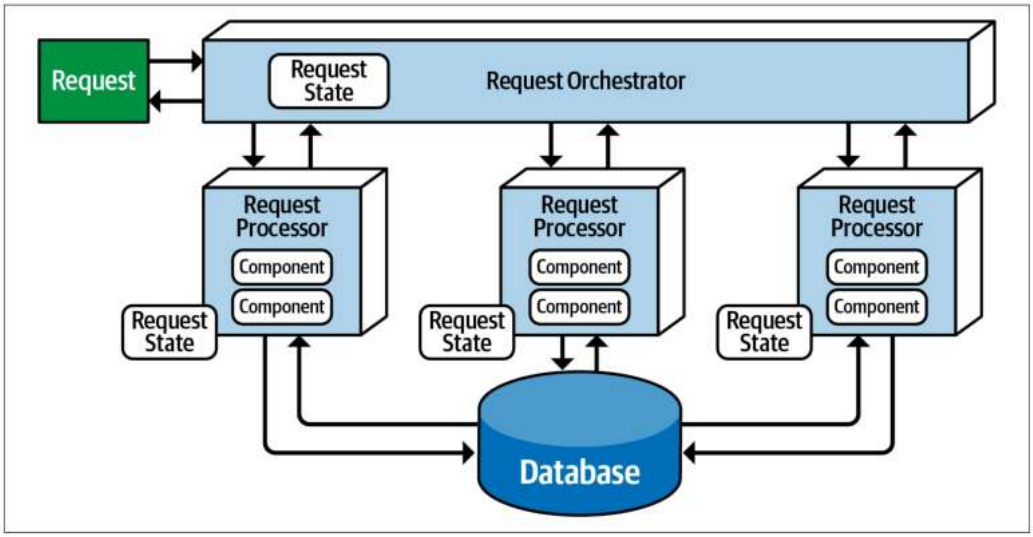
Quando Usar este Estilo de Arquitetura

- Quando requisitos de desempenho não são muito altos.
- Para Domain-Driver Design é um bom candidato.
- Quando preservar transações ACID é importante.
- Razoavelmente modular sem ser granular demais.
- Não requer tanta coordenação de muitos componentes. Exige alguma orquestração e coreografia.
    - **Orquestração:**
        - Coordenação de vários serviços por meio de um serviço mediador separado
    - **Coreografia:**
        - Coordenação de vários serviços pela forma com que os serviços se comunicam, sem um mediador.

14 - Arquitetura Orientada a Eventos

2025-12-21 11:14 am

- Popular para escalabilidade e alto desempenho
- Processamento assíncrono de eventos, desacoplado.
- Geralmente modelo *request-based:*

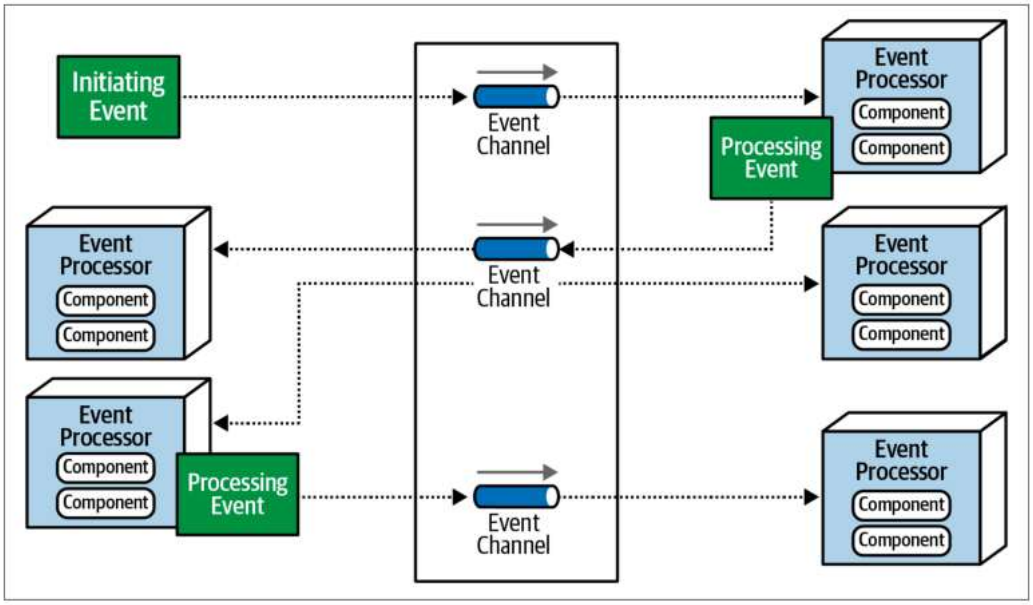


  - *Figure 14-1. Request-based model*
  - Usa um orquestrador de requisições.
    - Direciona as requisições cada uma a um processador de requisições.
    - Processadores lidam com a requisição em si.
- Modelo *event-based:*
  - Reage baseado em algum evento.
  - Não é um requisição em si, mas algo que ocorreu.

Topologia

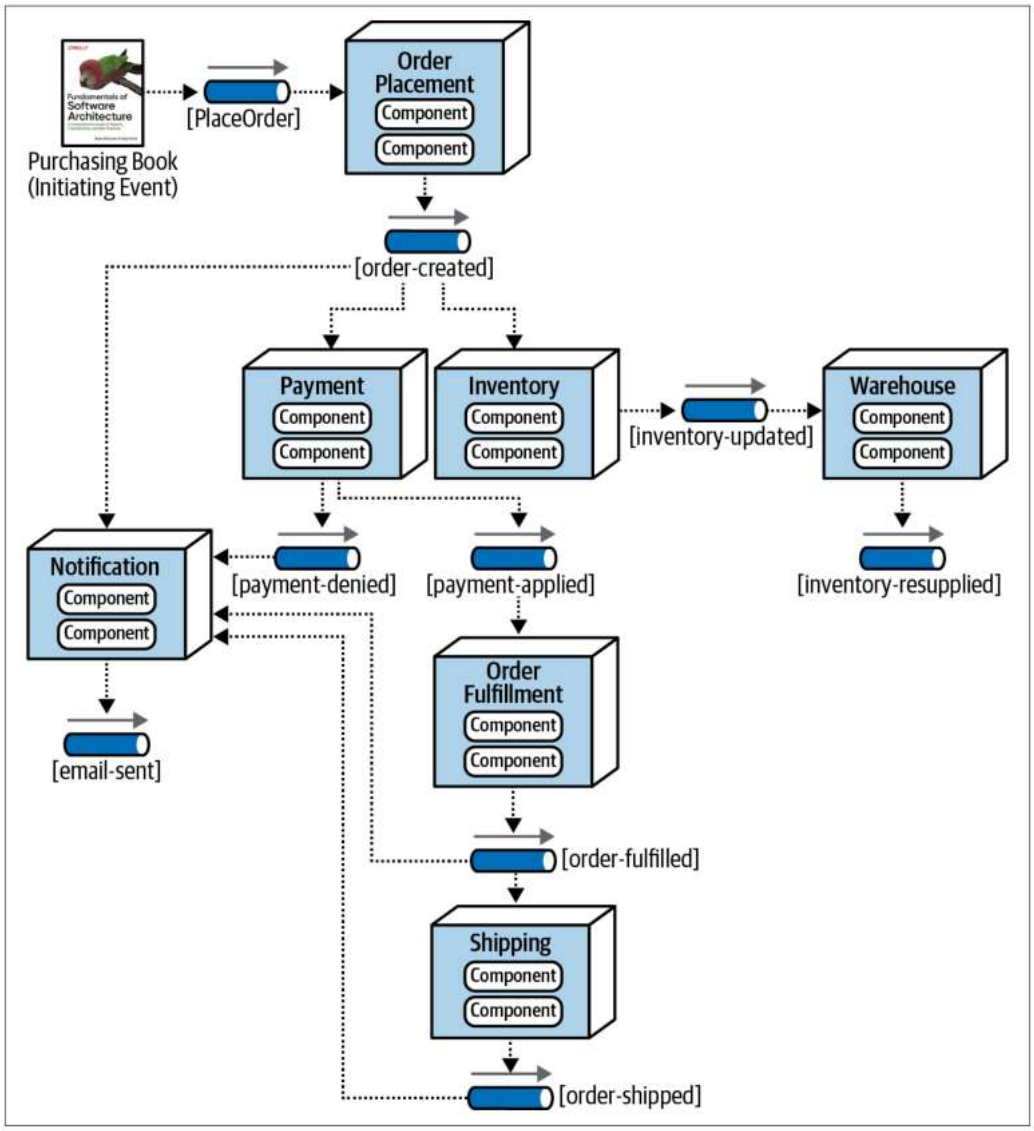
- Dois tipos:
  - **Mediator Topology**
    - Quando precisa de controle sobre o workflow do processamento
  - **Broker Topology**
    - Quando precisa de responsividade e controle dinâmico sobre o processamento de eventos.

Broker Topology



- *Figure 14-2. Broker topology*
- Não existe um mediador central.

- Eventos fluem como um rio entre componentes processadores.
- Bom quando processamento é relativamente simples.
- Há quatro componentes principais:
  - **Evento iniciador:**
    - Dá partida ao fluxo, um evento qualquer.
  - **Broker de Eventos:**
    - Tem o canal de eventos
  - **Processador de Evento:**
    - Tem os componentes que processam o evento
  - **Evento de Processamento:**
    - Enviado ao broker para mais processamento se necessário.
- Continua até o evento final não propagar mais nada que desencadeie um evento.
- Broker de eventos costuma ser federado.
- Tópicos usado no modelo *publish-subscribe*
- É boa prática que cada processador publique para todo o sistema o que foi feito, não só para as filas imediatamente relevantes.
  - Facilita extensibilidade caso sejam criados novos processadores.
- Exemplo:



  - *Figure 14-4. Example of the broker topology*
- Eventos disparam em cadeia.
- Processadores podem escalar independentemente uns dos outros.
- Problemas:
  - Não há controle geral do fluxo.
  - Lidar com erros é difícil.
  - Um erro pode travar tudo, ou pode dar erro apenas num parte e a outra parte continuar como se não houvesse erro.

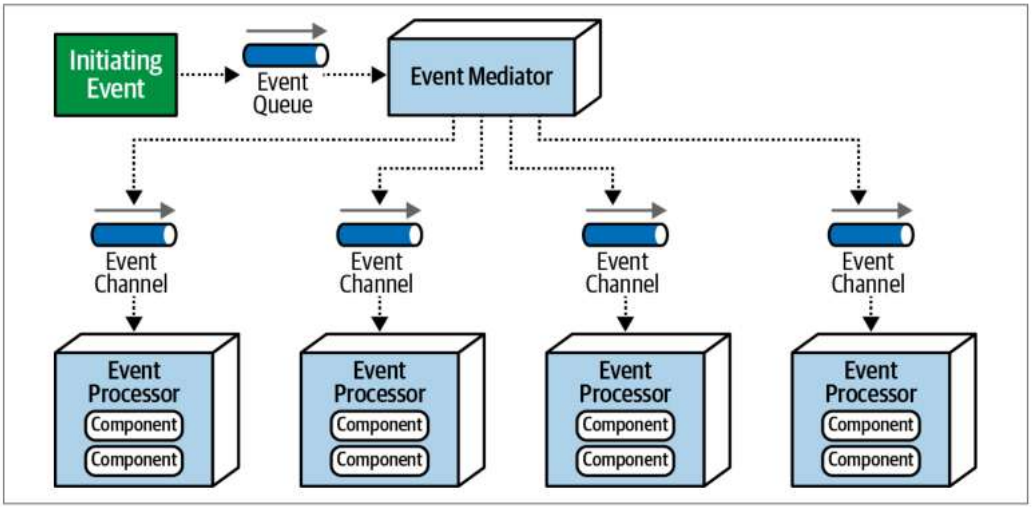
Table 14-1. Trade-offs of the broker topology

Advantages	Disadvantages
Highly decoupled event processors	Workflow control
High scalability	Error handling
High responsiveness	Recoverability
High performance	Restart capabilities
High fault tolerance	Data inconsistency

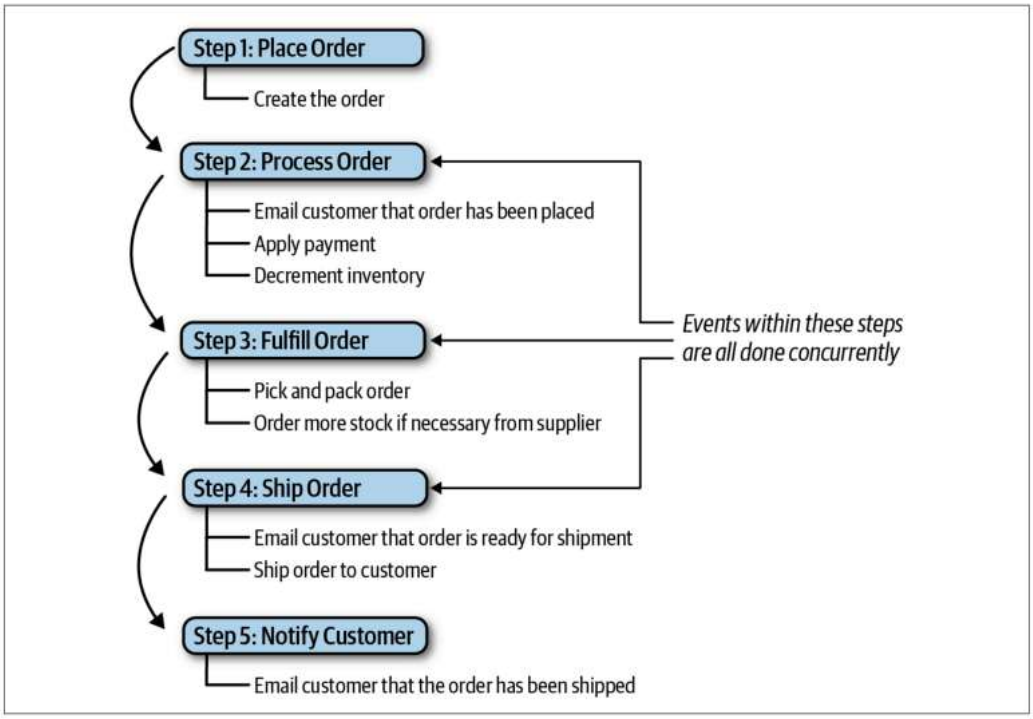
-

Mediator Topology

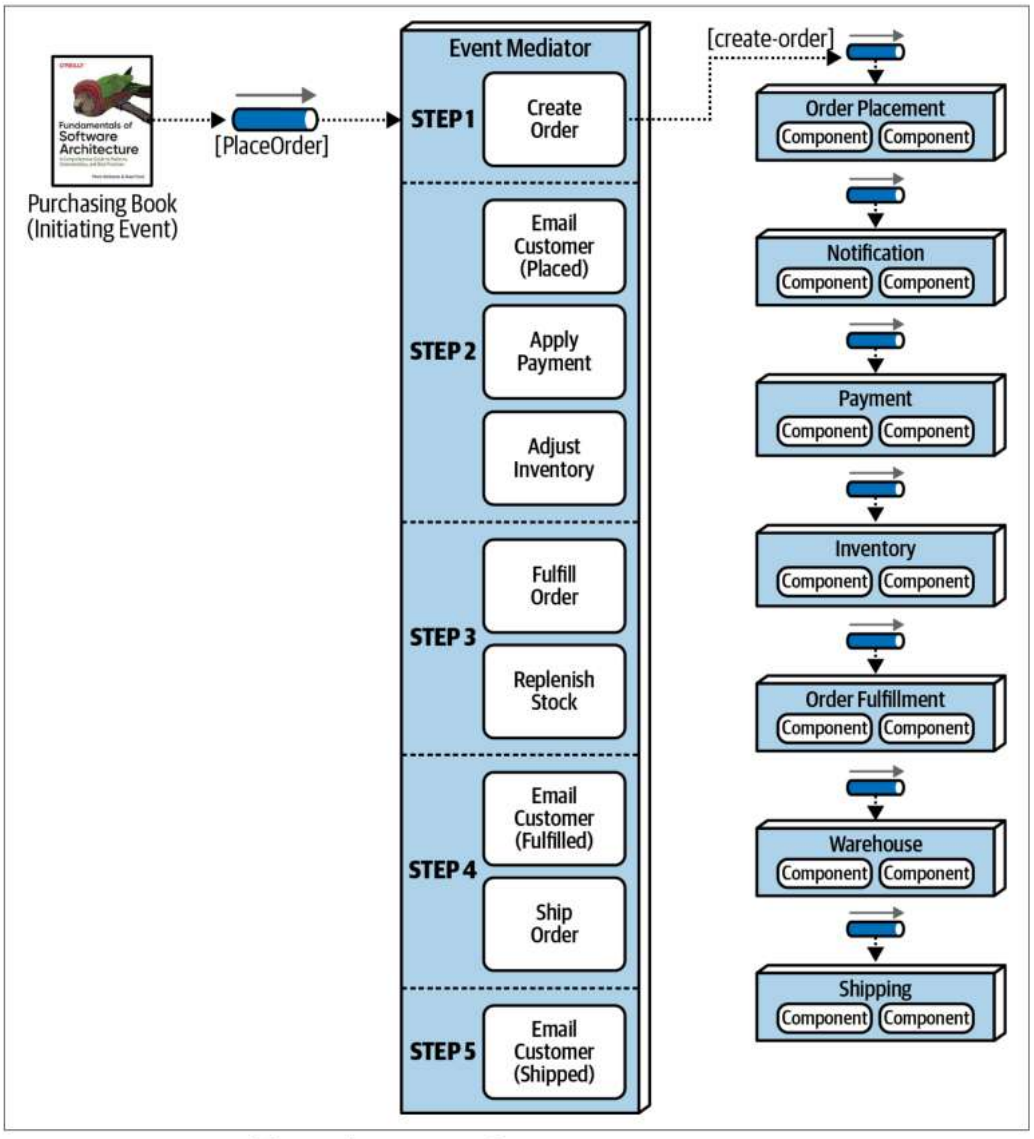
- Tenta corrigir problemas da Broker topology.
- Tem um mediador central



- *Figure 14-5. Mediator topology*
- Inicia com um evento iniciador
    - Mas difere da broker topology, o evento é mandado a uma fila de iniciação de eventos, aceita por um mediador de eventos.
- Mediador direciona cada evento aos canais (geralmente filas) correspondentes.
- Processador escuta a fila, processa e responde ao mediador.
    - Processador não publica pra todo o sistema o que foi feito.
- Geralmente há vários mediadores, agrupando eventos ou por domínio.
    - e.g. um mediador para eventos de clientes.
- Apache Camel, Mule ESB ou Spring Integration são mediadores comuns e simples.
- Se for muito complexo e cheio de condicionais, Apache ODE ou Oracle BPEL podem ser melhores.
    - Baseados em BPEL (Business Process Execution Language), XML-like que descreve os passos.
        - Completo mas complexo, geralmente tem um GUI para ajudar.
- Eventos podem ter human-in-the-loop, nem todo sistema tem suporte a isso
- Pode haver vários mediadores diferentes, que classificam complexidade de evento e redirecionam para outro mediador mais adequado.
- Na topologia de mediador, mediador sabe os passos necessários até o fim:



- *Figure 14-7. Mediator steps for placing an order*



- *Figure 14-8. Step 1 of the mediator example*

- Mediador espera o sucesso de todos os processadores de um dado passo pra partir para o próximo.
- Mantém estado do evento.
- Pode parar fluxo até que alguma condição ocorra.
- Lida com *comandos* (futuro) mais do que eventos (*passado*)
- É mais complexo e tem pior desempenho que broker topology
  - Mediador em si pode ser um gargalo.
- Acoplamento é maior.

Table 14-2. Trade-offs of the mediator topology

Advantages	Disadvantages
Workflow control	More coupling of event processors
Error handling	Lower scalability
Recoverability	Lower performance
Restart capabilities	Lower fault tolerance
Better data consistency	Modeling complex workflows

Capacidade Assíncrona

- Toda comunicação é assíncrona.
- Responsividade vs Desempenho
  - Responsividade é sobre UX e dar mensagem ao usuário
  - Desempenho é a velocidade do processo em si
  - Usuário precisa esperar o processamento inteiro ser confirmado ou só receber uma mensagem de acknowledgement imediata?
- Maior problema é *error handling

Error Handling

- *Workflow event pattern*
- Delega o erro para um processador de workflows e passa pra próxima mensagem imediatamente

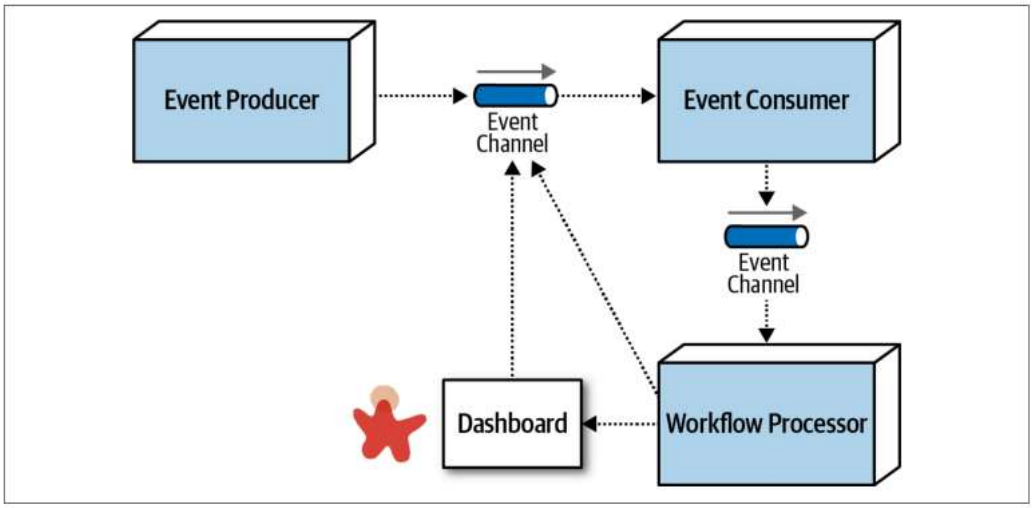


Figure 14-14. Workflow event pattern of reactive architecture

- Workflow processor tenta descobrir e tratar o erro.
  - Manda o resultado de volta pra fila para ser reprocessado, agora sem erro.
  - Se ainda não der certo, pode mandar o erro para um componente (*dashboard*) que aceita intervenção manual, para então ser tratado manualmente e redirecionado à fila.
- Exemplo:

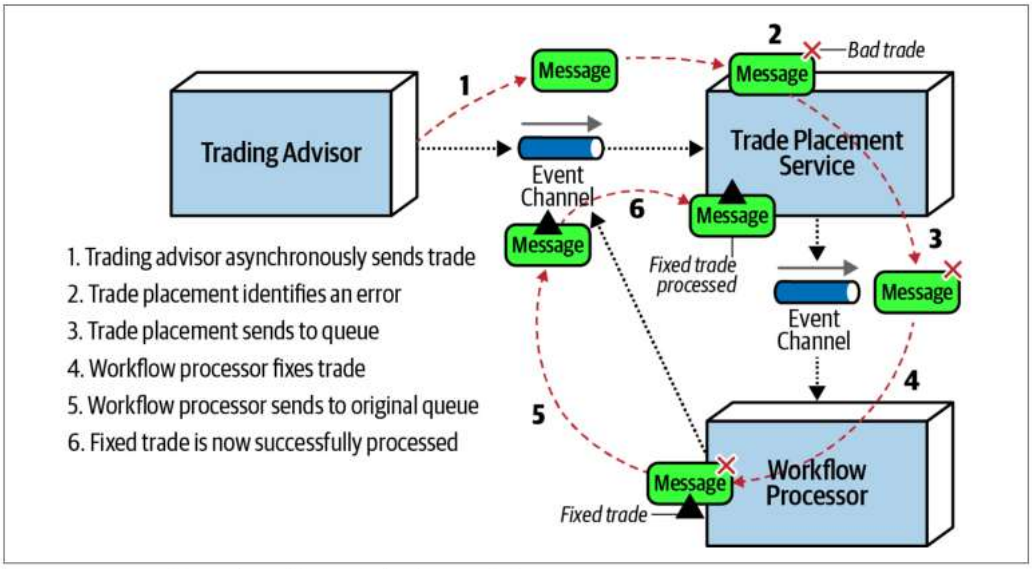


Figure 14-15. Error handling with the workflow event pattern

- Não é impossível manter a ordem original da fila, mas requer uma complexidade a mais de guardar as mensagens que devem aguardar em uma outra fila para manter a ordem até que o erro seja arrumado.

Prevenindo Perda de Dados

- Mensagens assíncronas podem se perder no caminho
- Erros podem ter origem em:
  1. Mensagem nunca chega à fila ou o broker falha
  2. Processador retira uma mensagem da fila mas falha durante processamento da mensagem.
  3. Persistência da mensagem falha.

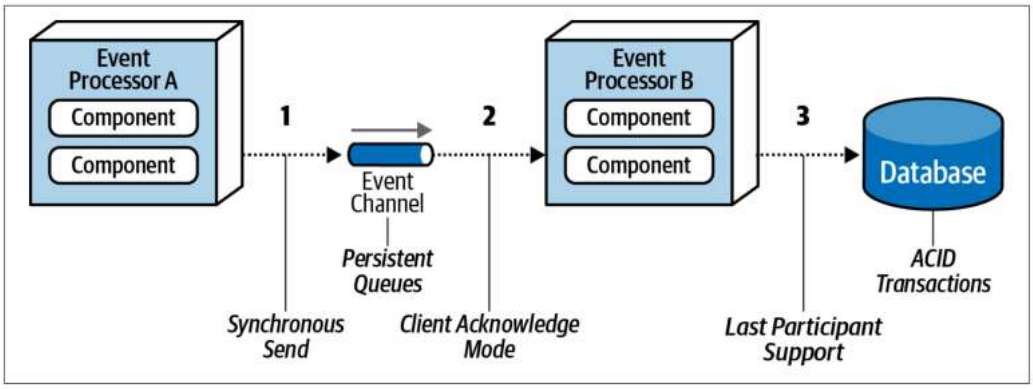


Figure 14-17. Preventing data loss within an event-driven architecture

- **Problema 1:**
  - Evitável com "*synchronous send*"
    - Mensagem é armazenada pelo broker não só em memória, mas em um datastore, que persiste caso o broker falhe.
    - Mensagem só é produzida quando broker garante que já foi persistida.
- **Problema 2:**
  - Evitável com *client ackowledge mode*
    - Mensagem é preservada na fila e intocável até que o processamento mande um sinal de sucesso.
- **Problema 3:**
  - Evitável com *last participant support (LPS)*
  - Uso de princípio ACID
  - Mensagem só é removida da fila quando há garantia da persistência no banco.

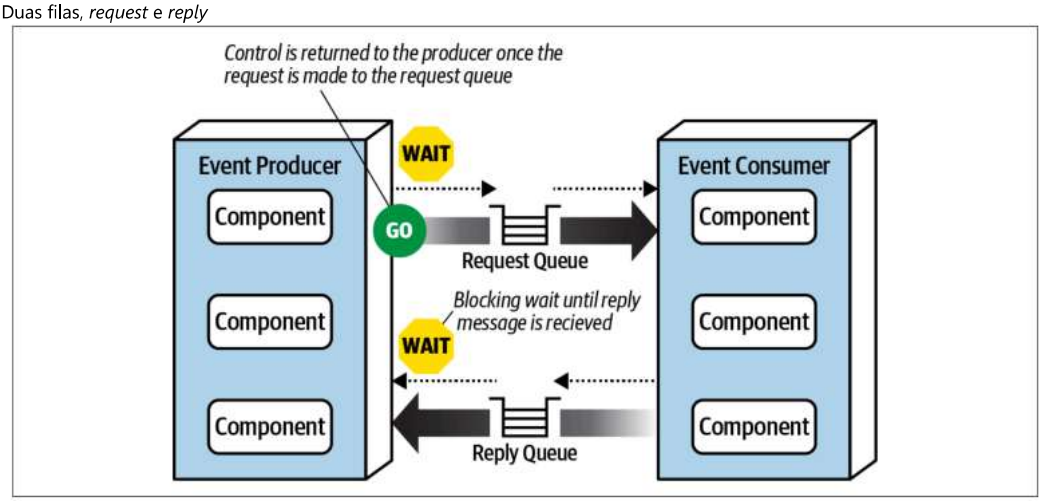
Capacidade de Broadcast

- Publicar eventos sem saber quem os receberá.
  - Pode ter vários subscribers da mesma mensagem.
- Altamente desacoplado.
- Requisito geralmente necessário.

Request-Reply

- E se uma ação depender de outra informação para ser executada?
  - e.g. comprar um livro requer ter um orderId
- Comunicação síncrona necessária.

- Mensageria *request-reply*
  - Duas filas, *request* e *reply*



  - *Figure 14-19. Request-reply message processing*
  - Produtor manda a request messag e espera
    - Consumidor processa e requisição e devolve mensagem para a fila de *reply*.
  - Comum usar um Correlation ID (CID) para isso
    - Produtor do evento guarda um CID de evento e espera até que receba de volta esse mesmo ID do consumidor.
  - Também pode ser usado uma fila de reply temporária:
    - Fila criada para uma requisição específica e deletada em seguida após receber a resposta.

Escolher entre Request-Based e Event-Based

- Request-based melhor para fluxo bem definido e que requer controle de fluxo.
- Event-based melhor para mais responsividade e escala.

Table 14-3. Trade-offs of the event-driven model

Advantages over request-based	Trade-offs
Better response to dynamic user content	Only supports eventual consistency
Better scalability and elasticity	Less control over processing flow
Better agility and change management	Less certainty over outcome of event flow
Better adaptability and extensibility	Difficult to test and debug
Better responsiveness and performance	
Better real-time decision making	
Better reaction to situational awareness	

Arquitetura Event-Driven Híbridas

- Arquitetura de eventos pode estar inserida em outra arquitetura para ter as vantagens da arquitetura de eventos.
  - E.g. microsserviços.

Características

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1 to many
Deployability	★★★
Elasticity	★★★
Evolutionary	★★★★★
Fault tolerance	★★★★★
Modularity	★★★★
Overall cost	★★★
Performance	★★★★★
Reliability	★★★
Scalability	★★★★★
Simplicity	★
Testability	★★

- *Figure 14-22. Event-driven architecture characteristics ratings*
- Altamente paralelizável e escalável.
- Não é simples e é difícil de testar.
  - Request-based é mais fácil de testar que event-based.
  - Árvore de possibilidades pode ser muito complexa.
- Fácil de estender.

15 - Arquitetura Baseada em Espaço

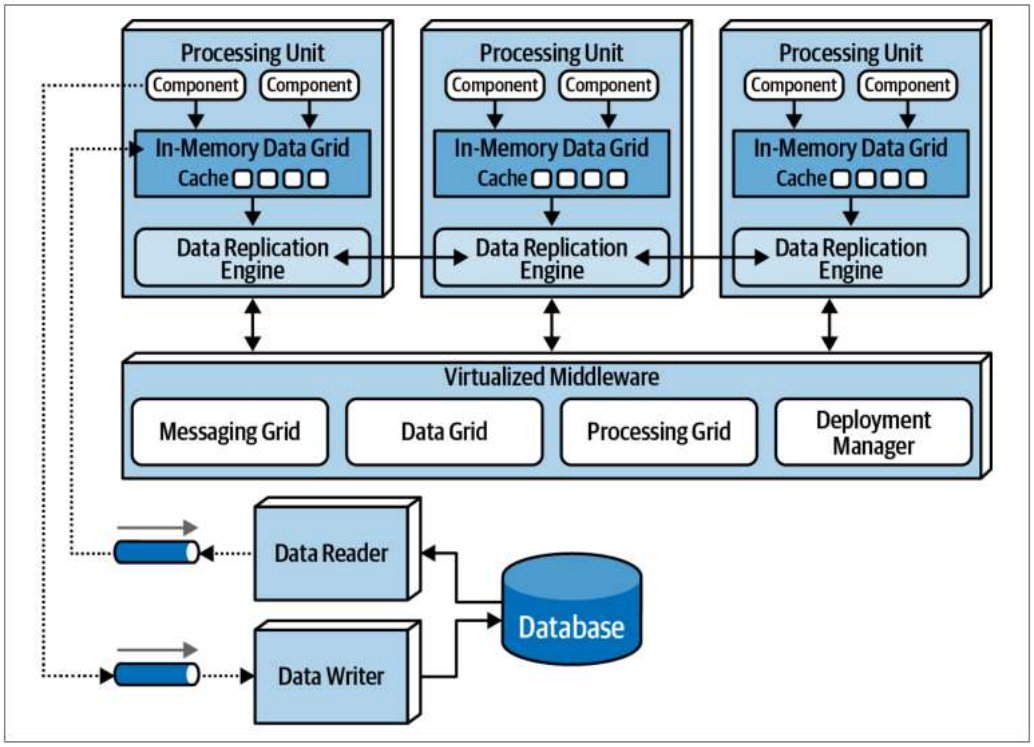
2025-12-22 08:03 am

Escalar uma aplicação web é desafiador:

- Webserver é mais fácil de escalar, que é mais fácil que a aplicação, que é mais fácil que escalar database.
- Acaba ficando escalado desigualmente.
- Baseado em espaço tenta consertar esse problema.
- Bom para aplicações de alto volume e de volume difícil de prever.

Topologia

- Não há gargalo de database central
- Dados são preservados em memória nas unidades de processamento.



- *Figure 15-2. Space-based architecture basic topology*

Unidade de Processamento

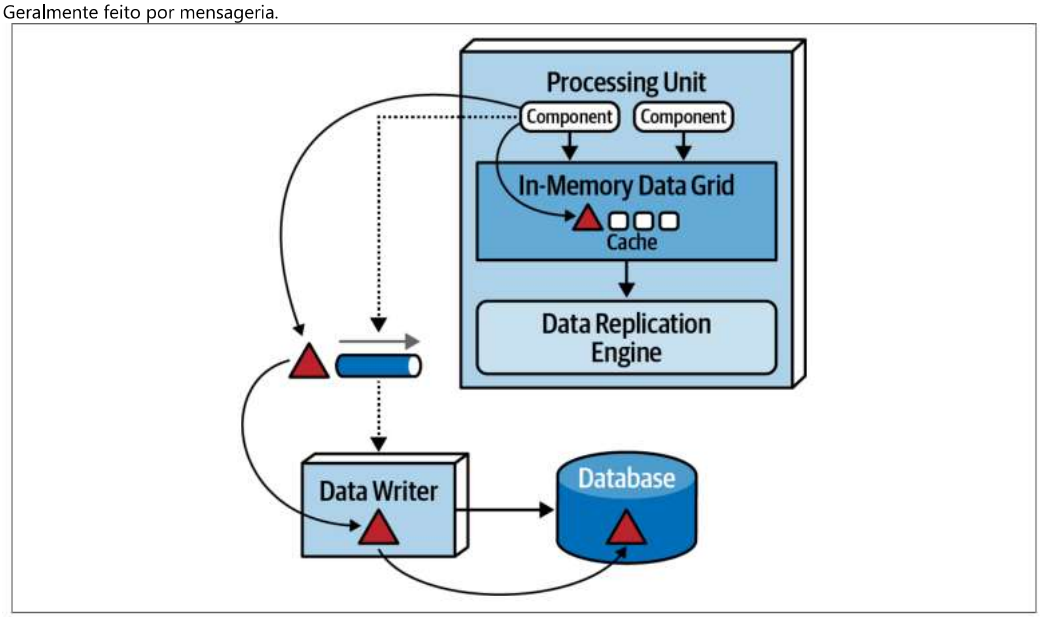
- Contém lógica da aplicação, inteira ou parcial.
  - Pode conter serviços em si.
- Também engine contém engine de dados-em-memória e de replicação.
  - Hazelcast, Apache Ignite ou Oracle Coherence.

Middleware Virtualizado

- Lida com a infraestrutura.
- Contém os seguintes componentes:
  - **Messaging Grid:**
    - Recebe a requisição para dentro do middleware virtualizado.
    - Determina quais unidades de processamento estão disponíveis para execução.
    - Geralmente feito com um webserver com load-balancing (e.g. Nginx)
  - **Data Grid:**
    - Coordena os data grids das unidades de processamento
    - Cada data grid em cada unidade de processamento tem que ter os mesmos dados, datagrid garante isso.
  - **Processing Grid:**
    - Componente opcional
    - Principalmente quando há várias unidades de processamento necessárias para uma única requisição.
    - Orquestra essa interação entre as unidades de processamento.
  - **Deployment Manager:**
    - Gerencia dinamicamente ligamento e desligamento de unidades de processamento a depender da carga.

Data Pumps

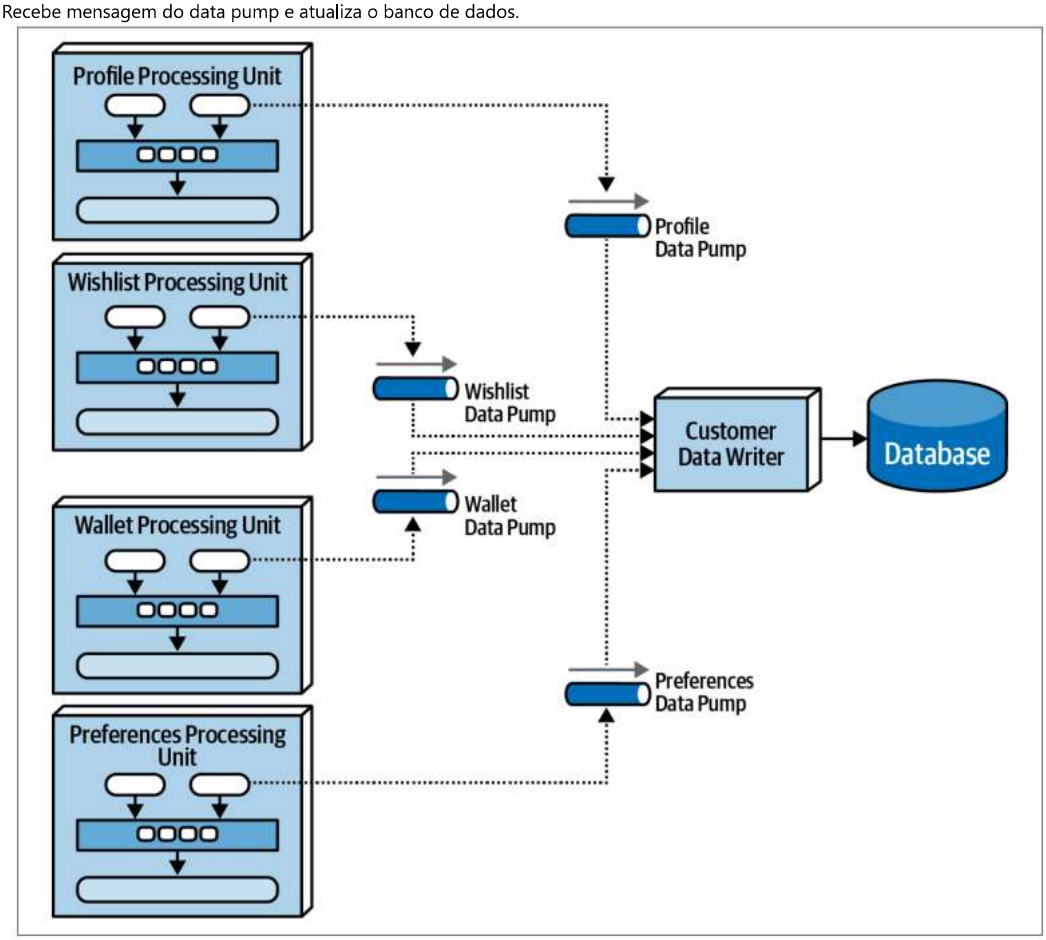
- Forma de mandar dados para outro processador.
- Unidades de processamento não leem ou escrevem diretamente a nenhum database.
- Unidade de processamento manda alterações que executou para serem sincronizadas pelas outras unidades.
    - Database será atualizado em algum momento.
- Geralmente feito por mensageria.



- *Figure 15-7. Data pump used to send data to a database*
- Costuma haver um data pump por domínio ou subdomínio.

Data Writers

- Recebe mensagem do data pump e atualiza o banco de dados.



- *Figure 15-8. Domain-based data writer*
- Podem ter granularidade variável
    - Vários pumps para um único writer, por exemplo.
    - Ou então um writer por data pump, também é possível.

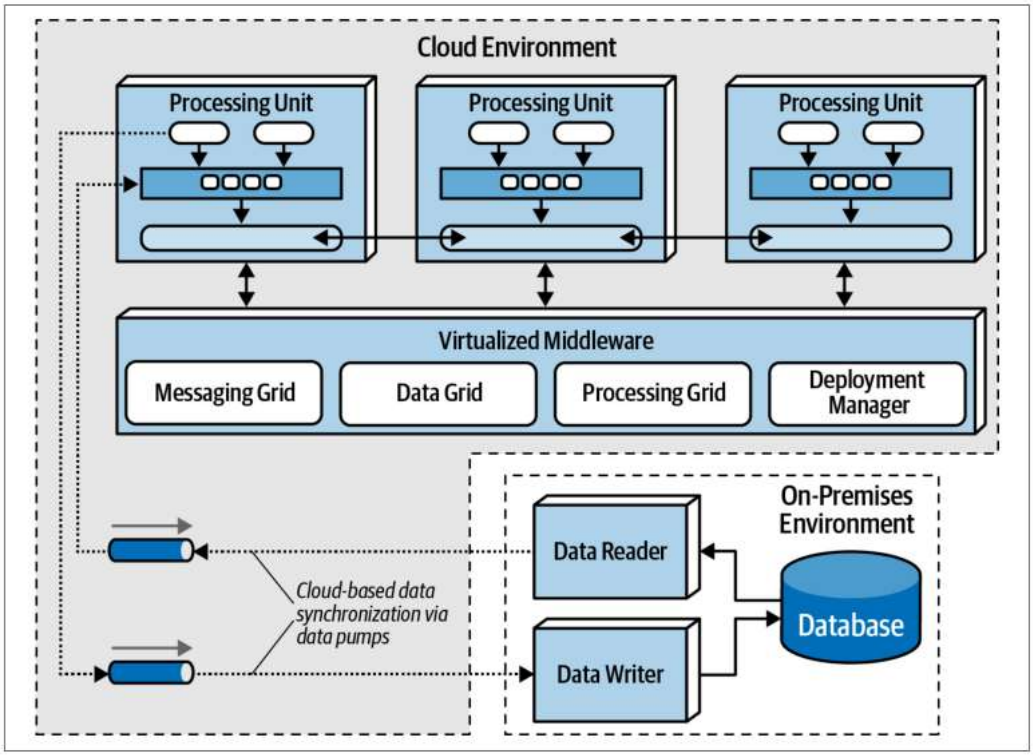
Data Readers

- Lê do database e manda pras unidades de processamento via data pump reversa.
- Só são invocados se:
    1. Todas as unidades de processamento com um cache estão derrubadas
    2. Um redeploy das unidade de processamento.
    3. Leitura de dados arquivados que não estão no cache
- Popula o cache das unidades processamento até não precisar mais
- Podem existir por domínio, assim como os writers.
- Junto com os data writers, formam a *camada de abstração de dados*
    - Unidades de processamento não acessam database diretamente, apensa essa camada.
    - Essa camada facilita lidar com mudanças no database.

Colisões de Dados

- Cache sofre atualização de um dado pela própria unidade de processamento e por replicação de outra unidade ao mesmo tempo devido a latência na replicação.
    - Replicação vinda externamente sobreescreveria o update local.
- Gera inconsistências.
- Taxa de colisão (`Colisões/unidade de tempo`).
    - $$CollisionRate = N * \frac{UR^2}{S} * RL$$
    - `N=número de serviços, UR=taxa de updates/ms, S=tamanho do cache, RL=Latência de Replicação`
    - Importante verificar a taxa para diferente momentos de carga sobre o sistema

Implementações Cloud Vs On-Premise

- Permite mesclar cloud e on-premise de um jeito que as outras arquiteturas não permitem.
    - E.g. Manter database on-prem e o resto na cloud.
    - 

    - *Figure 15-11. Hybrid cloud-based and on-prem topology*

Cache Distribuído Vs Replicado

- Geralmente requer que os cache entre as unidades sejam replicados (i.e. todas as unidades têm todas as informações em um namespace)
    - Mas cache distribuído também é possível (i.e. cada cache tem um pedaço das informações em um namespace)
- Replicação é rápido e tolerante a falhas.
- Mas se o volume do cache for muito grande (> 100MB) isso começa a se tornar um problema.
    - É o caso de usar caching distribuído.
- Caching distribuído requer um servidor externo que tem um cache centralizado.
    - As unidades de processamento então não têm um cache interno mas acessam este servidor central.

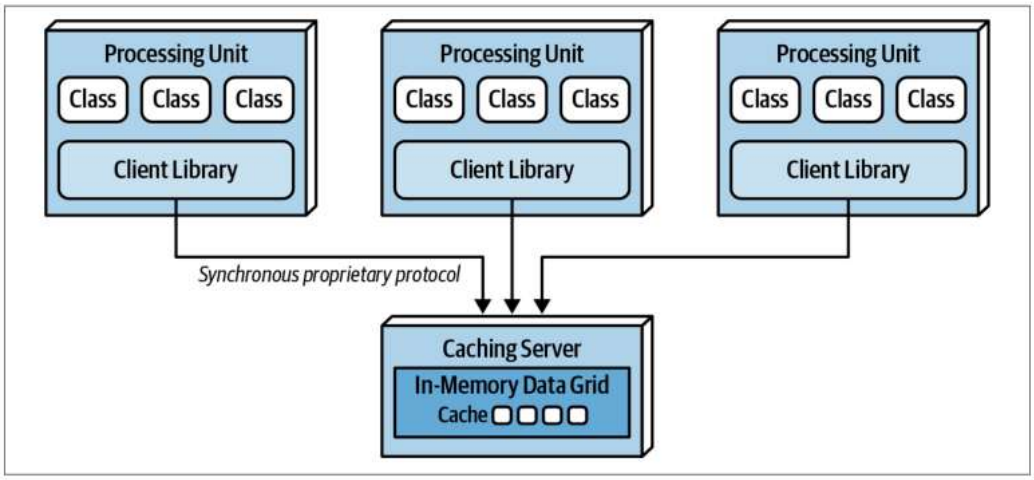


Figure 15-13. Distributed caching between processing units

- Menor desempenho que cache replicado simples, há mais latência nessa comunicação.
- Adiciona um ponto de falha central também.
  - Pode ser mitigado replicando o servidor central, mas daí surgem problemas de consistência entre as réplicas.
- Cache distribuído oferece melhor consistência de dados sobre cache replicado.
  - Dados ficam centralizados.
- Trade-off *consistência* VS *desempenho/tolerância a falhas*

Table 15-1. Distributed versus replicated caching

Decision criteria	Replicated cache	Distributed cache
Optimization	Performance	Consistency
Cache size	Small (<100 MB)	Large (>500 MB)
Type of data	Relatively static	Highly dynamic
Update frequency	Relatively low	High update rate
Fault tolerance	High	Low

- Pode ser bom usar os dois tipos de caching em pedaços diferentes da aplicação.
  - e.g. *inventário* requer alta consistência, *perfil do usuário* maximiza desempenho

Considerações de Near-Cache

- Near-cache é um caching híbrido.
  - In-memory com caching híbrido.
    - Caching distribuído --> *full backing cache*
    - Data Grid em memória --> *front cache*
- Front cache contém um subconjunto do backing caching
  - Vai renovando o conteúdo do front cache usando uma *eviction policy*
    - Por frequência, importância, recência, dos dados ou aleatório.

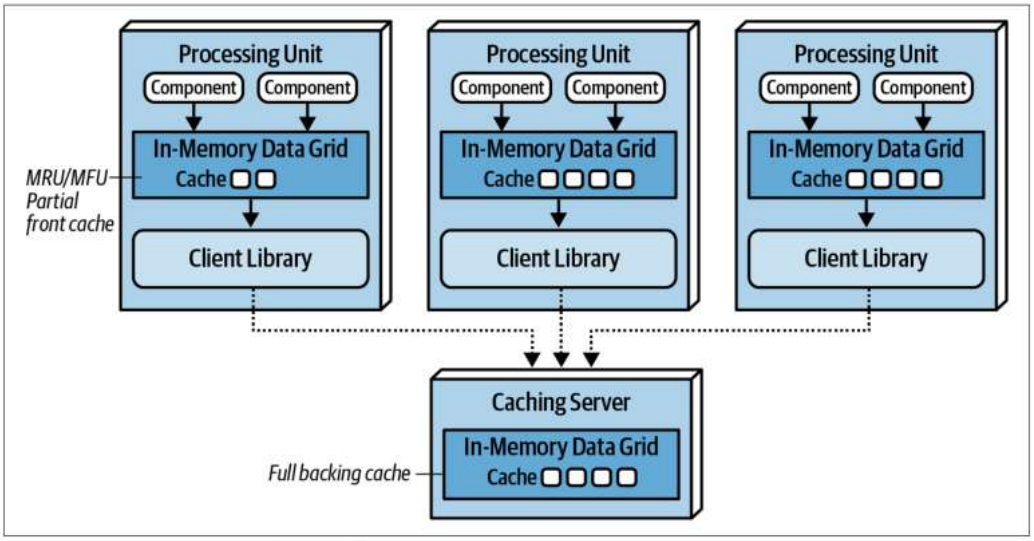


Figure 15-14. Near-cache topology

- Front caches não são sincronizados entre si.
  - Isso cria inconsistências de desesmpenho e responsividade entre as unidades de processamento.

- Por isso, Near-Cache, é **NÃO RECOMENDADO** para Arquitetura Baseada em Espaço.

Exemplos de Implementação

- Adequado para aplicações com alta variação de carga, com no mínimo 10.000 usuários simultâneos.
- **Sistema de Ingresso para Shows**
  - Volume de carga é baixo até ser altíssimo em instantes. Alta elasticidade.
  - Disponibilidade dos assentos deve ser altamente consistente e atualizar rapidamente.
    - Updates síncronos a um database seria impossível
  - Deployment manager lida com elasticidade da carga
    - De preferência antes das vendas começarem de fato.
- **Sistema de Leilão Online**
  - Também requer alto desempenho e elasticidade.

Características

Architecture characteristic	Star rating
Partitioning type	Domain and technical
Number of quanta	1 to many
Deployability	★★★
Elasticity	★★★★★
Evolutionary	★★★
Fault tolerance	★★★
Modularity	★★★
Overall cost	★★
Performance	★★★★★
Reliability	★★★★
Scalability	★★★★★
Simplicity	★
Testability	★

- *Figure 15-15. Space-based architecture characteristics ratings*
- Maximiza elasticidade e desempenho.
- Mas complexo e difícil de testar.
  - Difícil garantir que não há perda de dados caso algum elemento falhe.
  - Difícil testar volume de carga de uma situação extrema real.
- Tanto particionada tecnicamente quanto por domínio.
  - Unidade de processamento pode ser separada por domínio
  - Mas responsabilidades também tem camadas (como a abstração de dados) estritamente técnicas.
- Permite vários quanta.
  - Database não é um gargalo de características.

16 - Arquitetura Orchestration-Driven Service-Oriented

2025-12-22 09:43 am

- Arquitetura irrelevante
    - Embora faça sentido no papel.

História e Filosofia

- Restrições de Hardware direcionaram a arquitetura distribuída nos anos 90 para grandes empresas.
- Sistemas operacionais eram caros, não havia opção open-source confiável.
- Partição técnica ao extremo, tentativa de reusar tudo tanto quanto possível.

Topologia

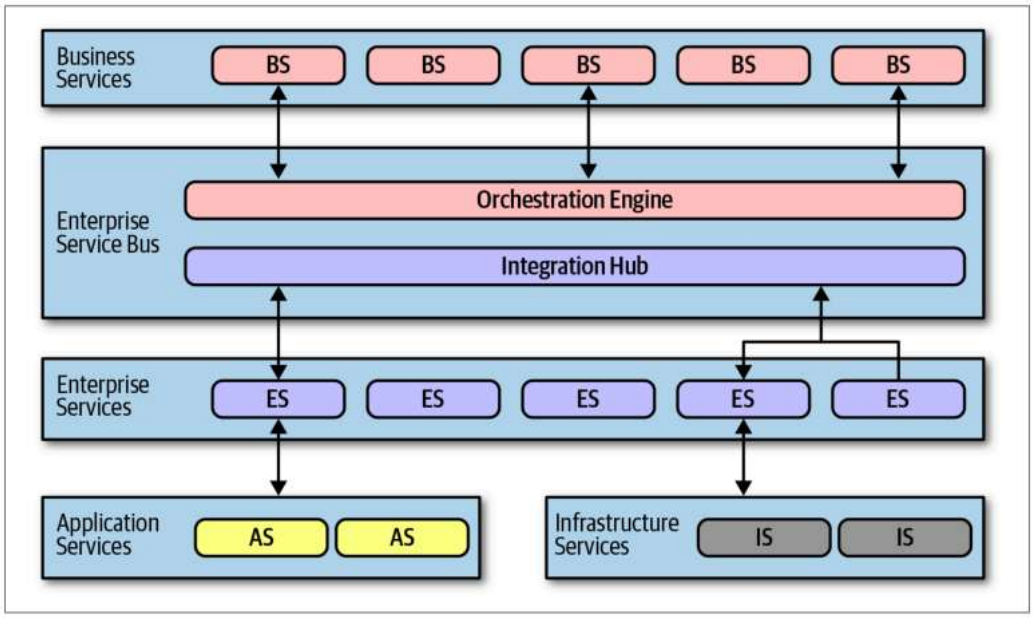


Figure 16-1. Topology of orchestration-driven service-oriented architecture

- Taxonomia de serviços, cada camada com uma responsabilidade.

Taxonomia

- Ideia é reusar código a nível de empresa.

Serviços de Negócio

- Ponto de entrada.
- Sem código, só definição de regras, input, output e schemas.
- Define uma operação (e.g. `ExecuteTrade`).

Serviços da Empresa

- Implementações bem específicas de ações (e.g. `CreateCustomer`)
- Blocos fundamentais dos serviços, acessível a toda a empresa.
- Mas a realidade é mais dinâmica do que essa ideia exige.

Serviços de Aplicação

- Serviços implementados uma vez só para serem reusados.
    - E.g. gerar geolocalização.

Serviços de Infraestrutura

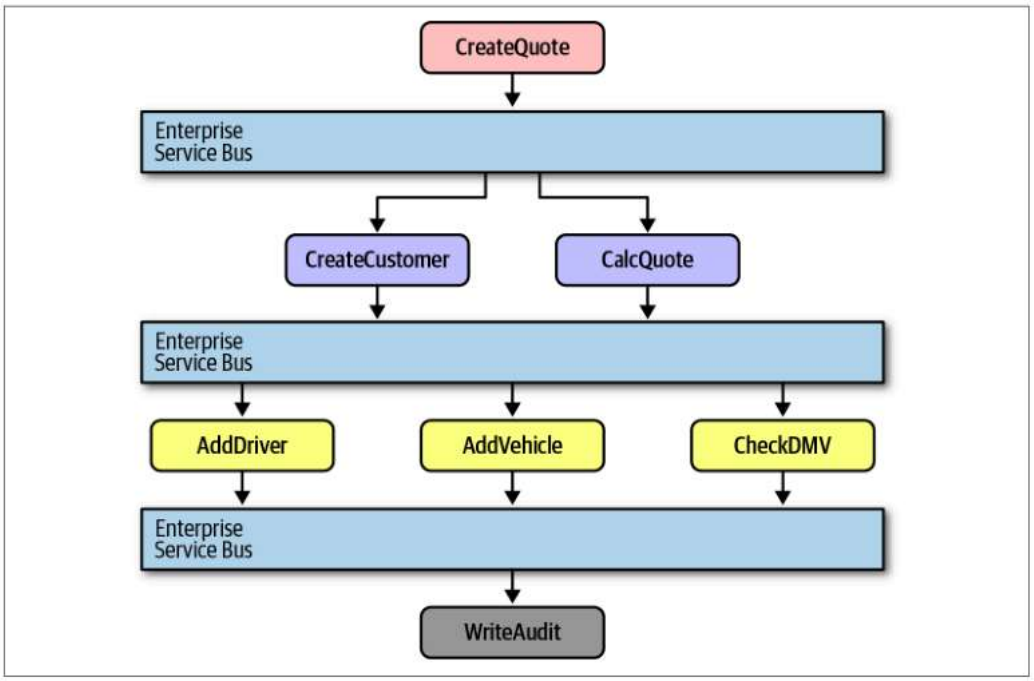
- Serviços de operação
    - Logging, monitoramento, autenticação, etc.

Engine de Orquestração

- Coordena todos os elementos.
- Associado a um banco de dados relacional.
- Define as relações entre Serviços de Negócio e de Empresa, como é a transação, etc.
- Resultado desastroso.
    - Integração impossível, complexa, burocrática.

Fluxo de Mensagens

- Todas as requisições passam pelo engine de orquestração.



- *Figure 16-2. Message flow with service-oriented architecture*

Reuso... e Acoplamento

- O grande objetivo desta arquitetura é o reuso a nível de serviço.
    - .e.g. Todos os setores têm a definição de Cliente
        - Por que não usar sempre o mesmo, uma vez só?
    - Mas se precisar mudar a definição de Cliente para um serviço, isso pode se propagar para todos os outros, mesmo não-relacionados.
- Comportamento fica num lugar só.
- Na prática, desastroso.

Características

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★
Elasticity	★★★
Evolutionary	★
Fault tolerance	★★★
Modularity	★★★
Overall cost	★
Performance	★★
Reliability	★★
Scalability	★★★★
Simplicity	★
Testability	★

-
- Difícil de testar, deployar.
- Complexo.
- Foi um marco na arquitetura de software para ensinar coisas a serem evitadas.

17 - Arquitetura de Microsserviços

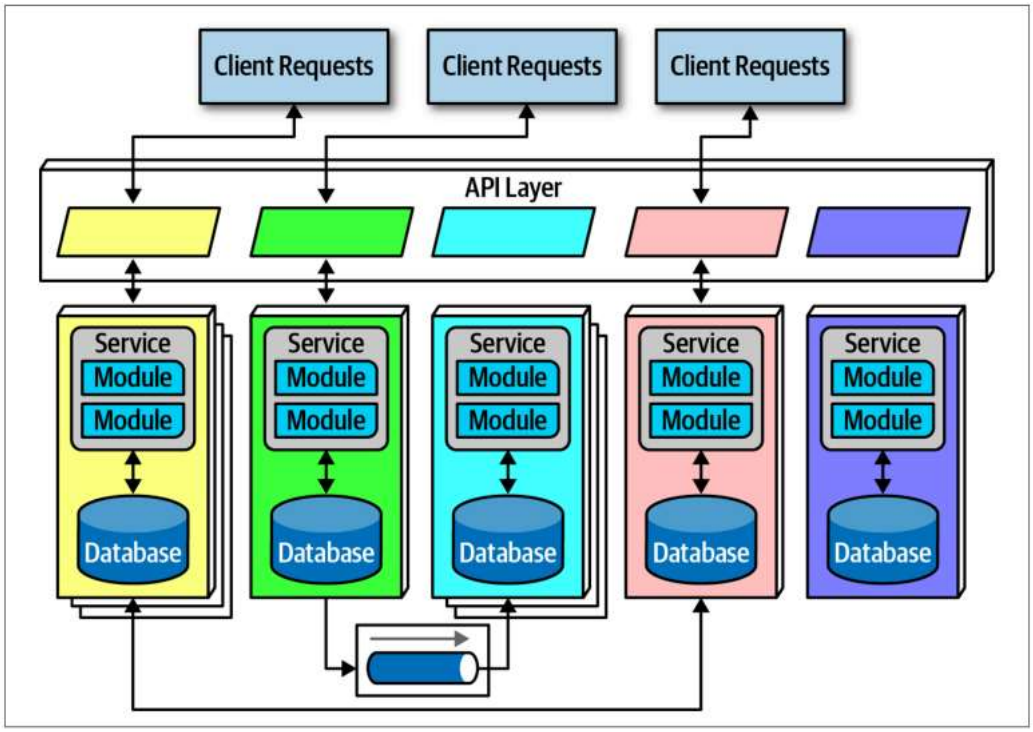
2025-12-22 14:20 pm

- Bastante popular ultimamente

História

- Geralmente uma nova arquitetura é uma soma descentralizada de muitos arquitetos diferentes tomando decisões similares por acaso que acaba se consolidando numa coisa só aos poucos organicamente.
- Mas microsserviços foi definido em Março de 2014 por Martin Fowler e James Lewis num blog.
- Inspirado por *Domain-Driven design*
  - *bounded context*
    - Forma de alto **desacoplamento**.
  - O contexto é acoplado dentro de si mas nunca fora dele.
    - Comportamento autocontido e independente.
- Trade-off entre *Reuso VS Acoplamento*
  - Para maximizar desacoplamento, é comum haver mais duplicação

Topologia



- *Figure 17-1. The topology of the microservices architecture style*
- Cada serviço tem tudo o que precisa para operar independentemente.

Distribuído

- Microsserviços são uma arquitetura distribuída
  - Pode haver vários numa mesma máquina.
    - Problemas de isolamento.
  - Hoje é fácil provisionar mais máquinas com mais sistema operacionais.
    - Antes era necessário compartilhar mais os recursos.
- Ter latência de rede e verificação de segurança em cada endpoint pode prejudicar o desempenho.
- Importante definir bem as fronteiras do sistema e evitar interações não apropriadas.

Bounded Context

- Cada serviço modela um domínio ou um workflow.
  - Contém dentro de si schema, classes, etc.
- Melhor duplicar código do que compartilhar para evitar acoplamento.
- Domain partitioned.

Granularidade

- Não é fácil achar a melhor granularidade para os serviços em "microsserviços".

    - É um erro fazê-los pequenos demais.
- Alguns critérios para definir as fronteiras:
    - **Propósito:**
        - Fronteira mais óbvia. Coesão funcional, contém um comportamento.
    - **Transações:**
        - Quais entidades que precisam cooperar revelam quais funções devem estar juntas. Tudo o mais constante, quanto menos transações, melhor.
    - **Coreografia:**
        - Mesmo vários serviços bem isolados poderiam ser um só serviço se houver muita comunicação entre eles.
- Granularidade ideal vem após refinar várias iterações.

Isolamento de Dados

- Requisito da arquitetura.
- Para evitar acoplamento, não pode haver um único database.
- *Entity Trap:*
    - Erro de modelar cada serviço para cada entidade num database.
- Não há como ter uma única fonte de verdade.
    - Duas alternativas:
        1. Definir um dado domínio para um dado como fonte de verdade
        2. Usar replicação de dados ou caching para distribuir informação
- Cada serviço com cada time pode tomar as melhores decisões para si sem afetar ou depender de mais ninguém.

Camada de API

- Pode ou não existir, entre os consumidores do sistema.
- Não pode ser um mediador ou orquestrador.
    - Contra filosofia de desacoplamento.

Reuso Operacional

- Como lidar com aspectos gerais do sistema, como logging, monitoramento, etc?
    - Cada microsserviço faz do seu modo, duplicando tudo?
- Para resolver isso, pode haver um sidecar em cada serviço com esses componentes, comum a todos os serviços.

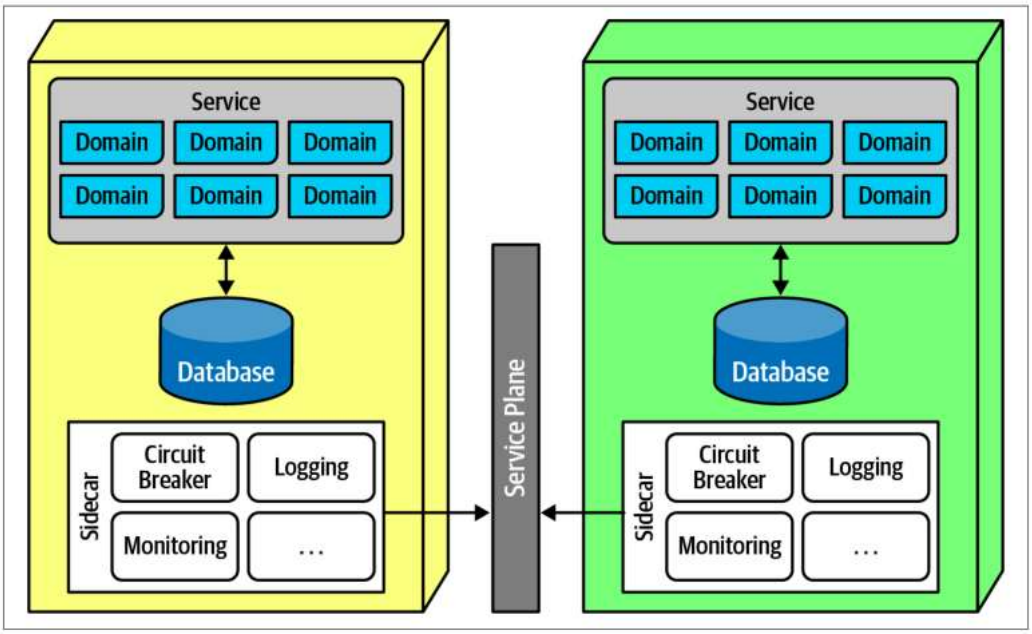


Figure 17-3. The service plane connects the sidecars in a service mesh

- *Service Discovery* tool faz parte de cada microsserviço para saber quais usar dinamicamente.
    - Pode estar na camade de API.

Frontends

- Difiícil ter o desacoplamento completo na interface do usuário.
- Duas alternativas comuns:
    - **Frontend Monolítico**
        - Se conecta à camada de API.
    - **Microfrontends**
        - Dividido em vários componentes.
        - e.g. **React**

Comunicação

- Pode ser **síncrona** ou **assíncrona**.
  - Espera ou não pela resposta.
- Para comunicação **síncrona**, costuma usar **protocol-aware[1] heterogeneous[2] interoperability[3]**:
  1. Microsserviços devem saber/descobrir como chamar uns aos outros
  2. Cada serviço pode ter uma stack de tecnologias diferente.
  3. Operam entre si pela rede.
- Usar stacks diferentes é uma forma não-convencional de garantir desacoplamento.
  - Oposto de padronização de tecnologias.
  - Garante que cada problema tenha a solução justa.
- Para comunicação **assíncrona:**
  - Comum usar eventos e mensagens.
  - Arquitetura orientada a eventos.

Coreografia e Orquestração

- *Coreografia* utiliza comunicação análoga a eventos com broker - simbiótico um com outro.
  - Sem coordenação central.
- *Isomorfismo de domínio e arquitetura* é quanto que uma dada arquitetura se encaixa com um certo estilo de arquitetura.
- Pode ser intuitivo usar um orquestrador em uma arquitetura de microsserviço para operações mais complexas.
  - Mas é contra a filosofia de desacoplamento.
  - É uma escolha do arquiteto fazer isso ou não. **Trade-off**

Transações e Sagas

- Desacoplamento extremo gera desafio de coordenar transações.
- Talvez surja vontade de fazer transações que violem as fronteiras de domínio
  - NÃO DEVE SER FEITO. (mas há exceções)
  - Geralmente se resolve aumentando a granularidade dos serviços.
- Padrão Saga de microsserviços:
  - Bem popular

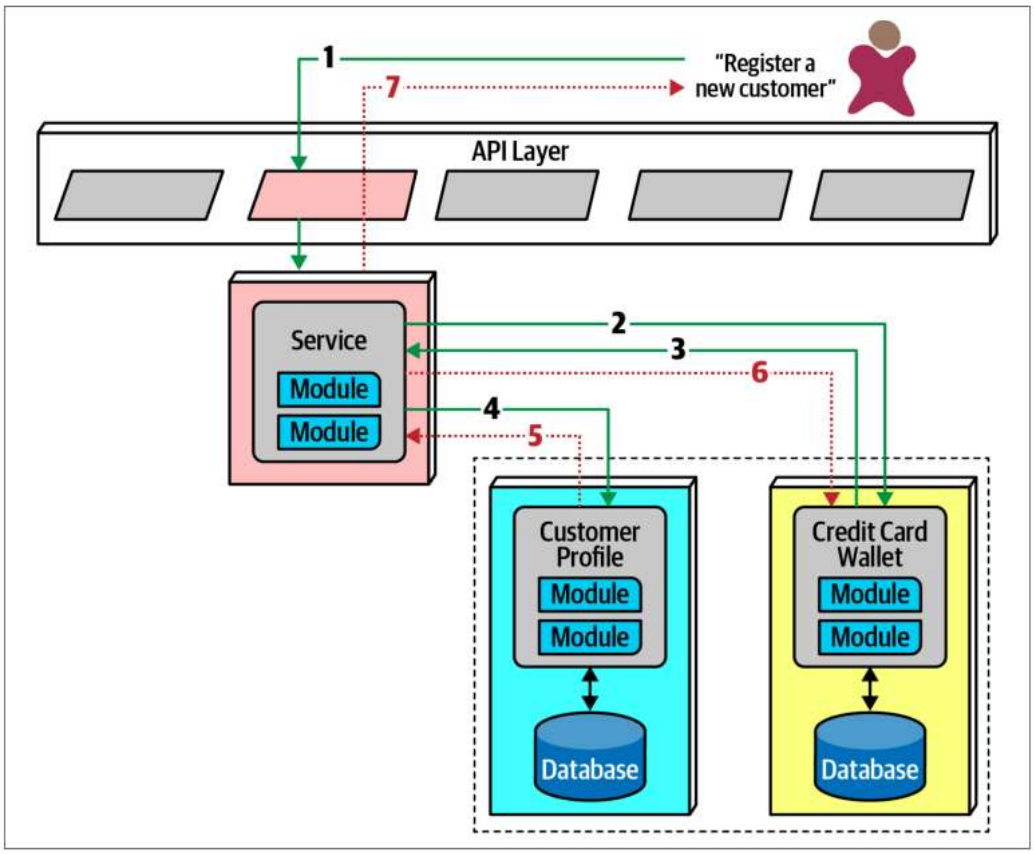


Figure 17-12. Saga pattern compensating transactions for error conditions

  - Serviço que media interação entre microsserviços sincronamente.
  - Consegue lidar com erros, pede para desfazer requisições se for preciso.
    - Mas desfazer costuma ser operação complexa.
  - Pode se tornar complexo se houver necessidade de coordenação e esperas de respostas.

Características

Architecture characteristic	Star rating
Partitioning type	Domain
Number of quanta	1 to many
Deployability	★★★★
Elasticity	★★★★★
Evolutionary	★★★★★
Fault tolerance	★★★★
Modularity	★★★★★
Overall cost	★
Performance	★★
Reliability	★★★★
Scalability	★★★★★
Simplicity	★
Testability	★★★★

-
- Moderno
- Tolerância a falhas e confiabilidade são desafios, contornáveis.
- Muito escalável e evolucionário.
- Latência de rede pode ser um gargalo de desempenho.
- Particionado por domínio.

18 - Escolhendo o Estilo Adequado de Arquitetura

2025-12-23 06:27 am

Sempre depende da situação.

A "Moda" que Muda na Arquitetura

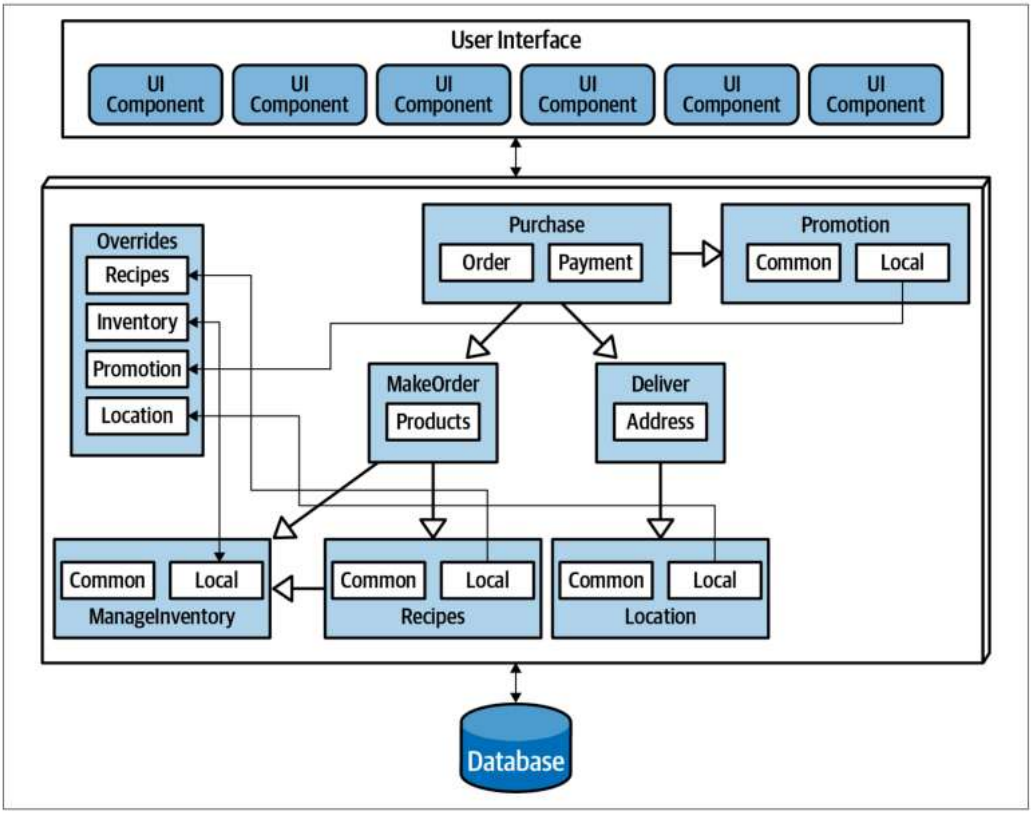
- Preferências mudam com o tempo.
- Experiências com o passado e mudanças no ecossistema influenciam.
- Novas capacidades (e.g. Docker) também alteram.
- Fatores externos como mudança no preço de licença também podem influenciar.

Critério de Decisão

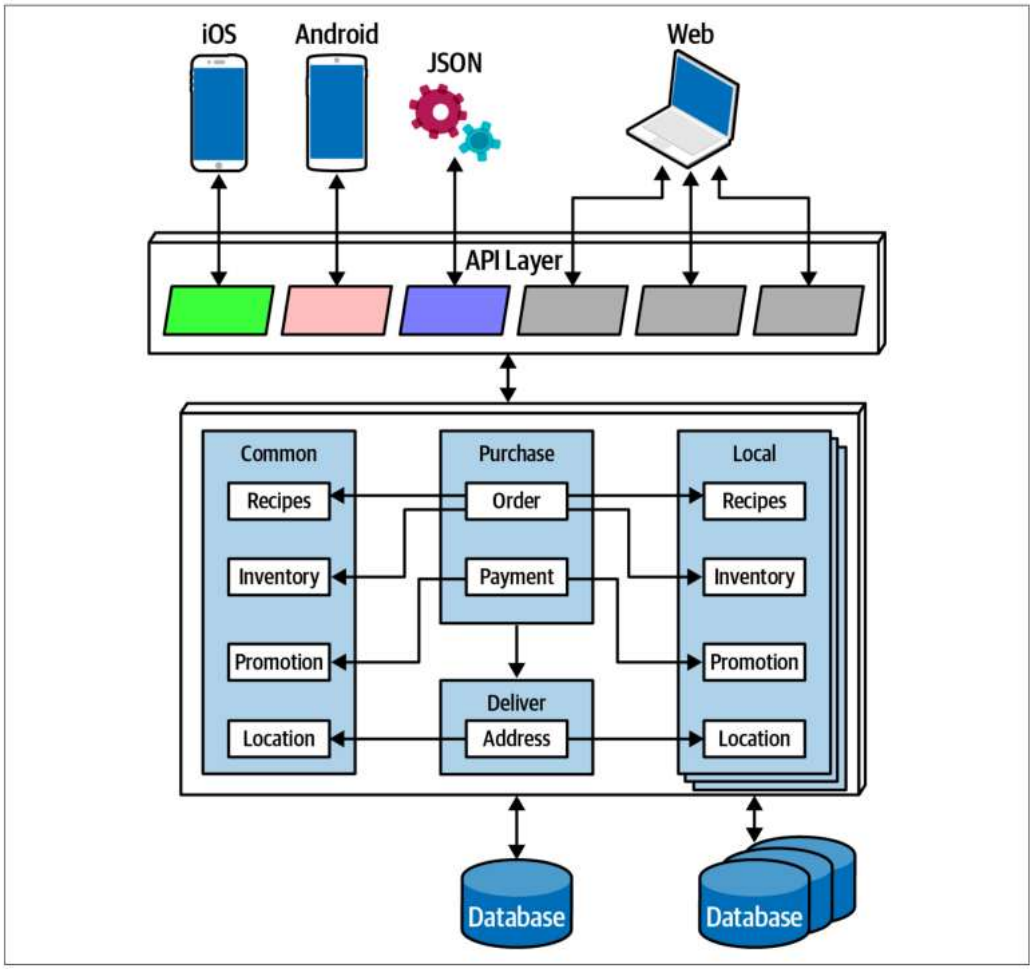
- Arquiteto deve considerar:
  - Domínio
  - Fatores organizacionais
  - Fatores organizacionais.
  - Isomorfismo entre domínio e arquitetura
  - Distrbuído vs Monolitoco?
  - Governança de Dados.
  - Síncrono ou Assíncrono?
    - Geralmente deve-se usar síncrono por default, assíncrono quando necessário.

Estudo de Caso Monolítico: Silicon Sandwiches

- Exemplo simples com um quanta só, monolítico atende.
- Monolito Modular

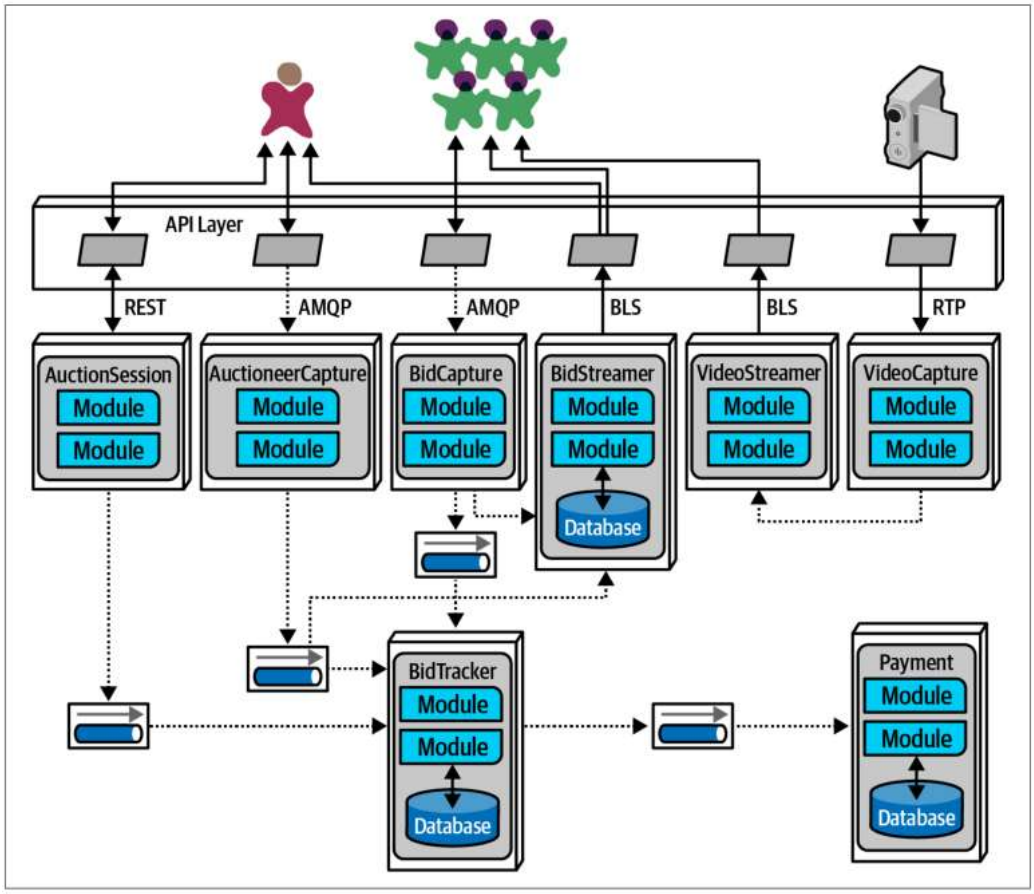


  - *Figure 18-1. A modular monolith implementation of Silicon Sandwiches*
  - Simples e barato. Database único, interface única.
  - Não muito flexível.
- Microkernel:



- *Figure 18-2. A microkernel implementation of Silicon Sandwiches*
  - Plug-ins não precisam interagir, podem ficar desacoplados.
  - Comunicação pode ser sempre síncrona
  - Sem grande requisito de desempenho.

- **Estudo de Caso Distribuído: Going, Going, Gone**
  - Leilão
  - Diferença de requisitos dentro da arquitetura.
  - Mais demanda de elasticidade e desempenho.
  - Microsserviço ou event-driven são bons candidatos.
    - Microsserviço permite mais segregação de características por componente



- *Figure 18-3. A microservices implementation of Going, Going, Gone*
- Cada componente é um serviço diferente.
- Algumas comunicações são síncronas, outras assíncronas, especialmente nas partes que requerem mais volume com resposta rápida.

19 - Decisões de Arquitetura

2025-12-23 06:49 am

Principal expectativa do arquiteto é tomar decisões de arquitetura

- Pode ou não envolver decisões de tecnologia.

Anti-padrões de Decisões de Arquitetura

Três principais antipadrões:

1. *Covering your assets*
    - Evita decisões por medo de errar
    - Uma forma de minimizar é esperar até o último momento responsável para decidir, esperar o máximo de informações possível.
    - Importante colaborar com o time de desenvolvimento continuamente
        - Colaboram pra dizer o que é ou não possível ou conveniente.
2. *Groundhog Day*
    - Ninguém sabe por que uma decisão foi tomada, então ela é discutida repetidamente.
    - Decisões têm que ter justificação ténica e de negócio
        - Apenas técnica não basta
    - Custo, tempo até mercado, satisfação do usuário e posicionamento estratégico costumam ser boas justificativas.
3. *Email-Driven Architecture*
    - Ninguém lembra ou sabe que houve uma decisão que está perdida entre e-mails.
    - Não é bom incluir decisão em um e-mail, costuma ter a decisão mas sem a justificativa.
    - Melhor apenas linkar o documento contendo a decisão bem explicada.
    - Apenas incluir as pessoas relevantes na comunicação.

Arquiteturalmente Significante

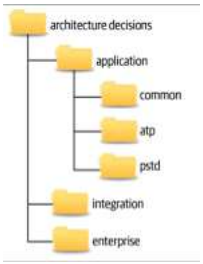
- Decisões de tecnologia podem ser decisões de arquitetura.
- Decisões relevantes são as que afetam:
    - *estrutura*
        - impacta padrões ou estilo de arquitetura (e.g. alteração do *bounded context*)
    - *características não-funcionais*
        - capacidades do sistema (e.g. desempenho)
    - *dependências*
        - Pontos de acoplamento entre componente/serviços
    - *interfaces*
        - Acesso e orquestração dos serviços (e.g. definição de contratos)
    - *técnicas de construção*
        - Alterações nas ferramentas e plataformas acessórias.

Registro de Decisões de Arquitetura

- *Architecture Decision Record* **(ADR)**
- Pequeno arquivo te texto de uma ou duas páginas especificando uma decisão.
- Cinco seções principais (pode ser diferente):

1. **Título**
    - Frase Descritiva
2. **Status**
    - Proposto, aceito ou substituído.
    - Quando é substituído, presume-se que havia sido aceito antes.
        - Refere-se qual ADR fez a substituição e também qual aquela ADR substituiu, se for o caso.
    - Pedir comentários também pode ser um Review
    - Pode ou não requerer aprovação externa, a depender do impacto e custo.
3. **Contexto**
    - Qual circunstância provocou a decisão?
    - Pode incluir alternativas possíveis.
4. **Decisão**
    - Decisão junto com a justificativa
5. **Consequências**
    - Qual o impacto esperado da decisão.
        - Bom e mau.
6. **Compliance**
    - Opcional
    - Como a decisão será medida e gerida.
        - Medição manual subjetiva ou com uma função de score.
7. **Anotações**
    - metadados como autor, datas, modificações, etc.
    - Útil mesmo dentro do Git.

Armazenando ADRs

- Cada decisão tem que ter o próprio arquivo separado.
- Uma alternativa é colocar no Github junto com o código.
    - Desvantagens, especialmente em organizações maiores:
        - Nem todo mundo da organização pode ter acesso
        - Não é bom quando existe um contexto fora da aplicação em si (e.g. integrações).
- Melhor ainda é colocar em uma wiki ou um file system compartilhado.
    - 

    - Dividido em:
        - Aplicação
        - Integração
        - Enterprise
            - Mudanças globais da organização

ADRs como Documentação

- Não há padrões amplamente definidos
- Funciona como documentação da arquitetura.

Usando ADRs para Padrões

- Define e justifica padrões utilizados.
- Permite refletir sobre padrões, se deve ou não ser utilizado.

Exemplo

ADR 76. Asynchronous Pub/Sub Messaging Between Bidding Services

STATUS
Accepted

CONTEXT
The Bid Capture Service, upon receiving a bid from an online bidder or from a live bidder via the auctioneer, must forward that bid onto the Bid Streamer Service and the Bidder Tracker Service. This could be done using asynchronous point-to-point (p2p) messaging, asynchronous publish-and-subscribe (pub/sub) messaging, or REST via the Online Auction API Layer.

DECISION
We will use asynchronous pub/sub messaging between the Bid Capture Service, Bid Streamer Service, and the Bidder Tracker Service.

The Bid Capture Service does not need any information back from the Bid Streamer Service or Bidder Tracker Service.

The Bid Streamer Service must receive bids in the exact order they were accepted by the Bid Capture Service. Using messaging and queues automatically guarantees the bid order for the stream.

Using async pub/sub messaging will increase the performance of the bidding process and allow for extensibility of bidding information.

CONSEQUENCES
We will require clustering and high availability of the message queues.

Internal bid events will be bypassing security checks done in the API layer.

UPDATE: Upon review at the April 14th, 2020 ARB meeting, the ARB decided that this was an acceptable trade-off and no additional security checks would be needed for bid events between these services.

COMPLIANCE
We will use periodic manual code and design reviews to ensure that asynchronous pub/sub messaging is being used between the Bid Capture Service, Bid Streamer Service, and the Bidder Tracker Service.

NOTES
Author: Subashini Nadella
Approved By: ARB Meeting Members, 14 APRIL 2020
Last Updated: 15 APRIL 2020 by Subashini Nadella

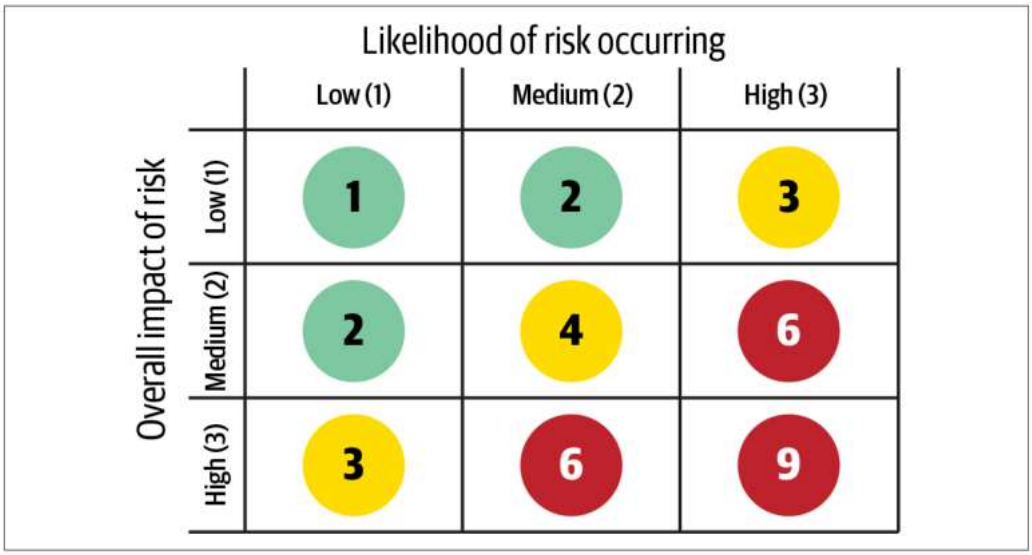
20 - Analisando Riscos de Arquitetura

2025-12-23 16:07 pm

Lidar com os riscos dos problemas de cada arquitetura.

Matriz de Risco

- Classificar o risco entre alto, médio e baixo.
- Impacto X Probabilidade.



- *Figure 20-1. Matrix for determining architecture risk*

Avaliação de Risco

- Risco por critério para cada domínio.

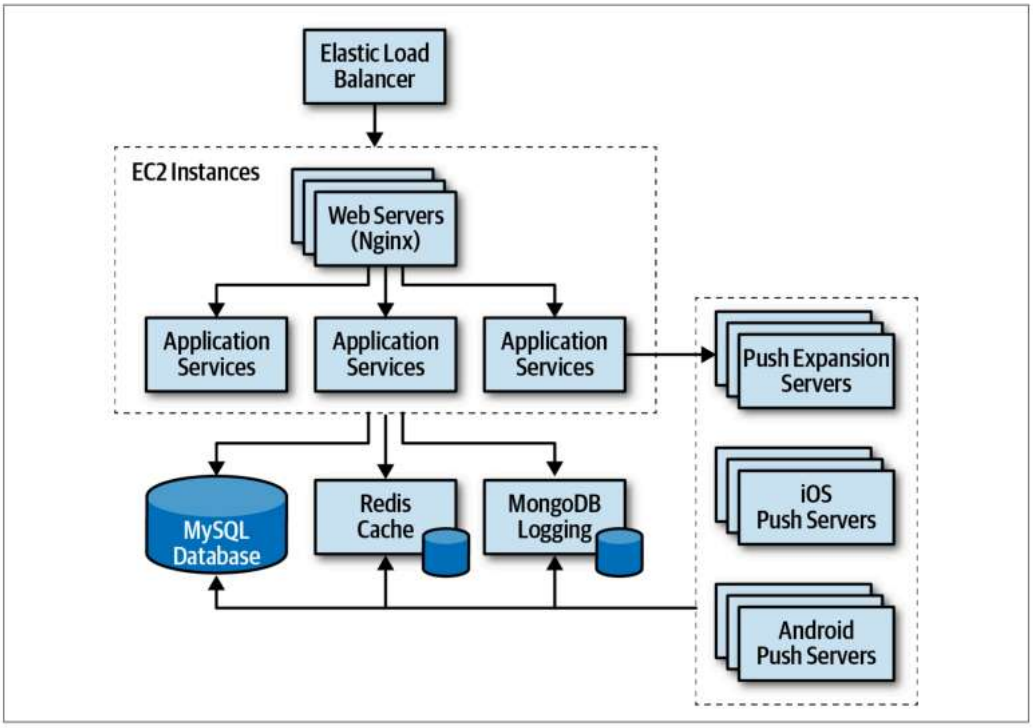
RISK CRITERIA	Customer registration	Catalog checkout	Order fulfillment	Order shipment	TOTAL RISK
Scalability	2	6	1	2	11
Availability	3	4	2	1	10
Performance	4	2	3	6	15
Security	6	3	1	1	11
Data integrity	9	6	1	1	17
TOTAL RISK	24	21	8	11	

- *Figure 20-2. Example of a standard risk assessment*
- Verifica-se o risco total de cada critério ou de cada domínio.
- Pode ser conveniente mostrar apenas os campos de alto risco numa apresentação.
  - Exclui todos os valores que não são 6 e 9 e a matriz é refeita, com o resto em branco.
- Não diz se está melhorando ou piorando com o tempo.
  - Podemos colocar ao lado de cada número um +/- ou um a seta indicando se subiu ou diminuiu do valor anterior.
- Funções de score podem definir o valor do risco objetivamente, se for possível.

Risk Storming

- Risco do sistema não pode ser determinado apenas pelo arquiteto sozinho.
  - Ignora ou desconhece alguns aspectos do sistema.
- Risk Storming é o exercício colaborativo de considerar riscos com outros arquitetos e desenvolvedores também.

    - Primeiro individualmente e depois em grupo.
- Três atividades:
    1. Identificação
    2. Consenso
    3. Mitigação
- Arquitetura exemplo:



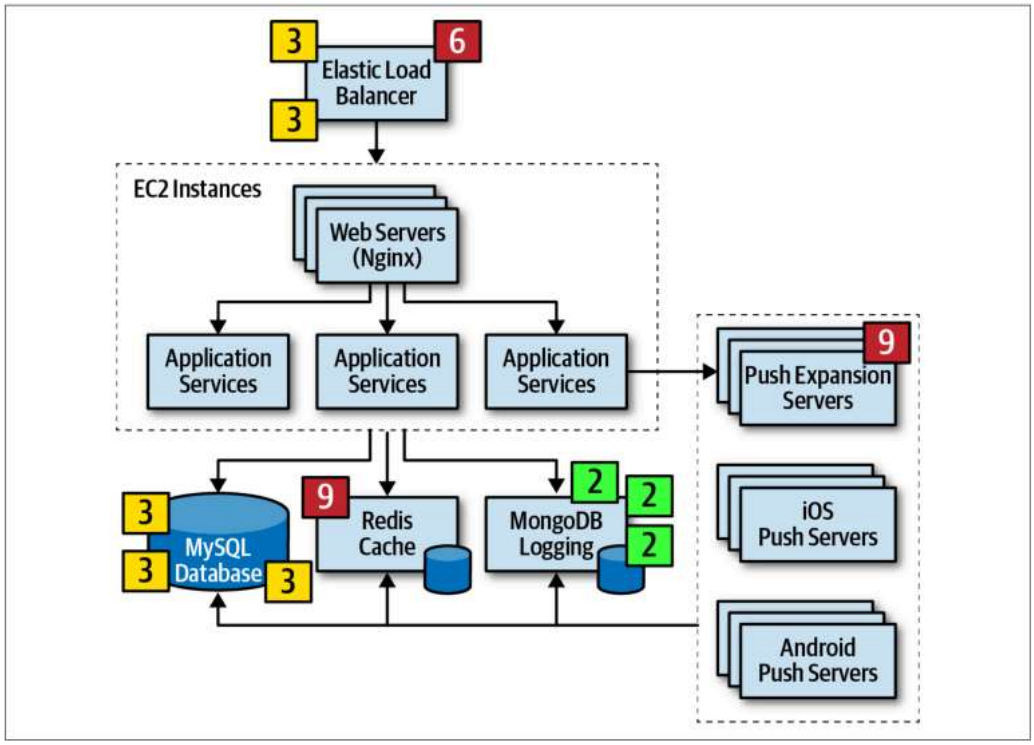
    - *Figure 20-6. Architecture diagram for risk storming example*

Identificação

- Cada participante descreve uma área de risco da arquitetura.
- Primeiro cada participante preenche a matriz de risco sozinho.
- Cada sessão de Risk Storming deve discutir um único critério (e.g. Desempenho), de preferência.

Consenso

- Etapa de discutir coletivamente
- É bom ter o diagrama da arquitetura desenhado bem grande em algum lugar.
- Participantes colocam post-its indicando nível de risco em cada lugar do diagrama.



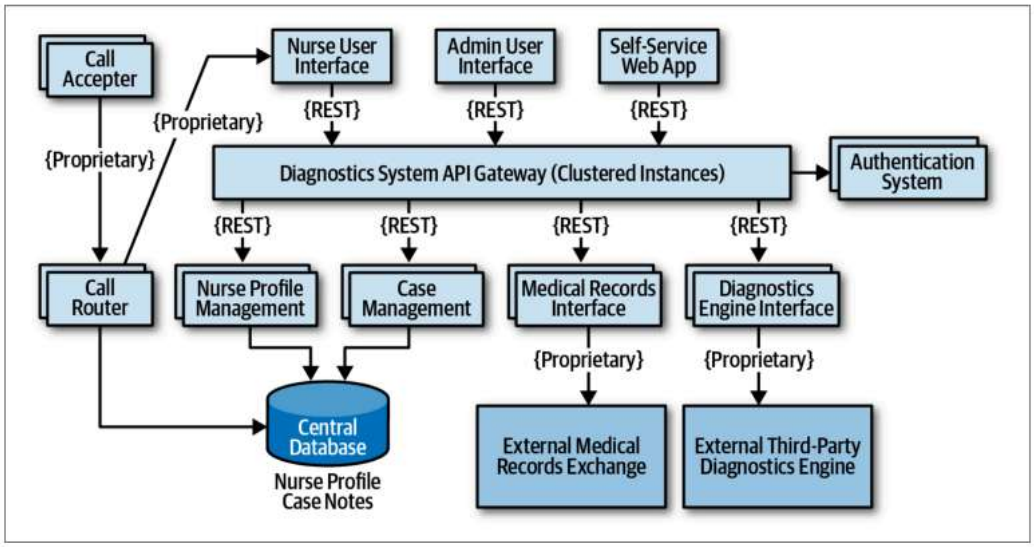
- *Figure 20-7. Initial identification of risk areas*
- Objetivo desta etapa é gerar um consenso.
- Onde já houve consenso inicial não há por que haver discussão mais neste momento.
- Cada participante argumenta em favor de sua avaliação.
- Para tecnologia desconhecidas ou não-testadas, sempre associar nota máxima (9) de risco.
- Discussão é feita até que reste só um post-it em cada área de risco que deva existir.

Mitigação

- Discutir mudanças e melhorias no sistema para minimizar riscos.
- Etapa requer um stakeholder para dizer qual custo faz sentido para mitigar qual risco.
    - Negociar estratégias de mitigação, mesmo as que sejam menos eficazes, até que o custo faça sentido.

Exemplos de Risk Storming

- Risk Storming não serve só pra arquitetura. Por exemplo jornada do usuário.
- Exemplo de arquitetura e estudo de caso:
    - Sistema de diagnósticos para enfermeiros.

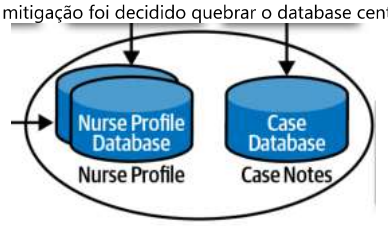


- *Figure 20-9. High-level architecture for nurse diagnostics system example*

Disponibilidade

- Database central foi identificado como risco nível 6
- E outros assessments também são tidos como consenso:

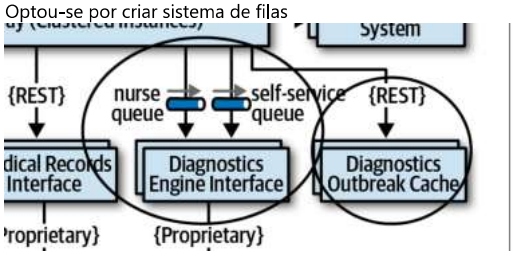
- Uma mitigação foi decidido quebrar o database central em dois databases diferentes.



- Mas mitigar disponibilidade de recursos externos é bem mais complicado
  - Após pesquisa concluiu-se que o risco é pequeno o suficiente para não ser mitigado.

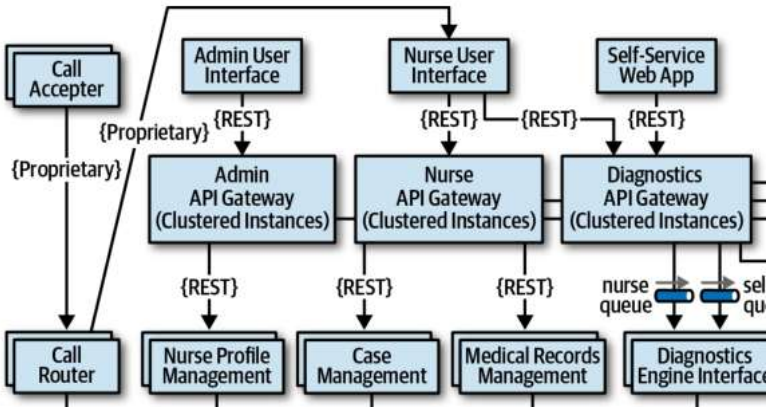
Elasticidade

- Problemas de surto de demanda (e.g. temporada de gripe)
- Identificou-se que interface não seria capaz de lidar com muitas requisições por segundo.
  - Optou-se por criar sistema de filas



Segurança

- Regulação rígida de dados de saúde.
- Identificou-se API de diagnósticos como risco em potencial, contra a expectativa do arquiteto que considerava o sistema seguro.
  - Todos usuários usam a mesma API, podem acessar dados controlados.
- Para mitigar, separar API gateways por tipo de usuário para granulirizar acesso e controle de segurança.



- Risk Storming é um processo contínuo ao longo do ciclo de vida de um sistema.

21 - Diagramando e Apresentando Arquiteturas

2025-12-23 20:42 pm

- Trabalho do arquiteto vai bem além do conhecimento técnico
- Comunicação é vital.
- Importante sempre apresentar as partes no contexto de todo o sistema.

Diagramas

- Compreensão do sistema como um todo.
- **Ferramentas:**
  - Importante não se apegar a algum diagrama só porque demorou pra fazer ou é bonito.
  - Existem várias ferramentas poderosas (e.g. *OmniGraffle*)
  - Deve ter os seguintes features:
    - **Camadas**
      - Poder mostrar ou esconder pedaços do sistema facilmente.
    - **Templates**
      - Ter biblioteca de componentes visuais comuns
    - **Magnets**
      - Ajudante pras linhas sempre ficarem retas, conectadas e bonitas.

Padrões de Diagramas

- Há vários:
  - **UML**
    - Unified Modeling Language
    - Bem pouco usado, mais usado para estrutura e workflow.
  - **C4**
    - Substituto moderno do UML
    - Quatro Cs:
      - **Contexto:**
        - Mostra todo o contexto do sistema, incluindo usuários e elementos externos
      - **Container:**
        - Limites físicos dos deployments
      - **Componente:**
        - Visão dos componentes
      - **Classe:**
        - Diagramas das Classes.
    - Ainda tem deficiências.
      - Mais adequado para arquiteturas monolíticas.
      - Nem tão adequado para distribuídos, como microsserviços.
  - **ArchiMate**
    - Suporta arquiteturas entre domínios de negócio.
    - Open source.
    - Popular. Tenta ser mais simples do que abrangente.

Diretrizes de Diagramas

- Cada pessoa tem um estilo, mas alguns elementos comuns a se pensar:
- **Título**
  - Ter tudo bem etiquetado
- **Linhas**
  - Não deve ser finas e devem ter direção quando convém comunicar isso.
  - Linhas sólidas são comunicação síncrona e pontilhadas assíncronas.
- **Formas**
  - Não há padrões de forma.
  - Geralmente elementos 3D são artefatos deployáveis, retângulos indicam pertencimento.
- **Labels**
  - Ter todos os itens descritos para tirar ambiguidade.
- **Cor**
  - Útil para separar, embora haja tradição de ser tudo monocromático.
- **Legenda**
  - Caso as formas sejam ambíguas.

Apresentação

- Importante dedicar esforço para estudar como fazer apresentações no powerpoint.

Manipulação do Tempo

- Um dos pontos mais importantes de apresentar. Apresentador controla fluxo de ideias, diferente da leitura em que é o leitor.
- Transições e animações podem ajudar nisso.
  - Podem indicar separação entre tópicos.

- Organizar o tempo por ideias, não por slides.
    - Algumas ideias demandam menos tempo e menos slides do que outras.

Construções Incrementais

- Evitar o slide ser o próprio texto lido pelo apresentador.
- Slides devem ter só as informação relevante da ideia, pouco a pouco, não ter uma parede de informação.
    - Revelar informação aos poucos, conforme desenvolve a ideia.

Infodeck Vs Apresentações

- Infodeck são slides que devem ser lidos e não apresentados.
- Infodecks precisam ser mais completos, não têm o complemento da fala, e não requerem animações e transições da mesma forma.

Slides são a Metade da História

- A apresentação em si deve ser relevante, e não supérflua, importante compor os dois meios de comunicação.

Invisibilidade

- Slide em branco para dar o foco no apresentador.
    - Enfatiza parte falada.

22 - Fazendo Times Efetivos

2025-12-24 08:00 am

- Arquiteto tem que guiar o time
- Tem que ser colaborativo.

Fronteiras de Time

- Arquiteto tem muita influência sobre o sucesso do desenvolvimento.
- Deve comunicar restrições e desafios com clareza.
    - Tem um ponto ótimo de restrição e direcionamento aos desenvolvedores.
    - Nem restrito demais nem liberdade demais.

Personalidades de Arquitetos

- Três tipos:
    - **Ultra-Controlador:**
        - Tenta controlar todo e cada detalhe do desenvolvimento.
        - Define todos os nomes, tecnologias, tudo.
        - Tenta substituir si próprio o trabalho dos desenvolvedores.
    - **Desconectado (Armchair):**
        - Não liga pros detalhes de implementação.
        - Desconectado do time de desenvolvimento, não atua como mentor.
        - Só desenha diagramas e fim.
        - Desenvolvedores acabam tomando o papel do arquiteto no desenvolvimento.
    - **Efetivo**:
        - Gera as restrições adequadas ao time.
        - Promove mentoria e colaboração.
        - Ajuda o time de desenvolvimento a superar os desafios.

Quanto Controle?

- *Liderança Elástica*
- Nível de controle deve depender das circunstâncias:
    - **Familiaridade do Time:**
        - Quanto time se conhece.
        - Quanto mais de conhecem, menos controle é necessário.
    - **Tamanho de Time:**
        - Quanto maior o time, mais controle é necessário.
    - **Experiência Geral:**
        - Nível de experiência é parecido dentro do time?
        - Quanto mais sênior, menos controle é necessário. Atuar mais como facilitador.
    - **Complexidade do Projeto:**
        - Quanto mais complexo, mais controle é necessário.
    - **Duração do Projeto:**
        - Quanto mais curto, menos controle é necessário.
        - Projeto curto não tem tempo para formalidades e testes extensivos, melhor deixar o time mais livre.
        - Projetos mais longos requerem ajuda do arquiteto para não deixar largar algum senso de urgência.
- Exemplos:

Table 22-1. Scenario 1 example for amount of control

Factor	Value	Rating	Personality
Team familiarity	New team members	+20	Control freak
Team size	Small (4 members)	-20	Armchair architect
Overall experience	All experienced	-20	Armchair architect
Project complexity	Relatively simple	-20	Armchair architect
Project duration	2 months	-20	Armchair architect
Accumulated score		-60	Armchair architect

    -

Table 22-2. Scenario 2 example for amount of control

Factor	Value	Rating	Personality
Team familiarity	Know each other well	-20	Armchair architect
Team size	Large (12 members)	+20	Control freak
Overall experience	Mostly junior	+20	Control freak
Project complexity	High complexity	+20	Control freak
Project duration	6 months	-20	Armchair architect
Accumulated score		-20	Control freak

Sinais de Atenção no Time

- Para definir tamanho ideal de time há três fatores:
  1. *Process Loss:*
     - Lei de Brook
     - Quanto mais pessoas, mais tempo o projeto toma.
     - *Process Loss* é a diferença entre a produtividade teórica máxima e a produtividade real do time.
     - Quantidade de *Merge Conflicts* são sinal de process loss.
     - Paralelizar tarefas é uma boa ideia para minimizar isso.
       - Se não há tarefas paralelas, talvez o time pudesse ser menor.
  2. *Pluralistic Ignorance:*
     - Quanto maior o time, menor a chance de alguém se objetar a alguma decisão.
       - Medo de não ter percebido algum detalhe que todo o resto percebeu (ou finge perceber)
     - Arquiteto deve ser capaz de identificar e evitar que as pessoas só aceitem uma decisão por inércia.
  3. *Diffusion of Responsibility:*
     - Quanto maior o time, pior a comunicação.
     - Incertezas sobre quem faz o quê e tarefas serem abandonadas é um bom sinal de time grande demais.

Aproveitando o Máximo das Checklists

- Checklists são ferramentas poderosas.
  - Usado por cirurgiões e pilotos de avião.
- Arquitetos devem saber quando usar ou não checklists.
- Checklist não deve conter passos dependentes um do outros.
  - (não concordo muito aqui, mas está escrito)
- Também não deve ter tarefas simples demais e conhecidas que já são executadas bem.
- Itens devem ser independentes.
- Não transformar tudo em checklist.
- Têm que ser o menor possível mas ainda significativa com todos os passos necessários.
  - Checklists grandes demais são ignoradas.
- Tudo que pode ser automatizado não deve estar na checklist.
- *Hawthorne Effect:*
  - Pessoas agem diferente quando sabem que podem estar sendo observadas (estejam de fato ou não)
  - Evita que desenvolvedores marquem checklists sem ter concluído de fato.
- Três checklists comuns bastante úteis:
  1. **Checklist de Conclusão de Código pelos Desenvolvedores:**
     - Ter a definição de "Concluído" clara, com uma checklist.
     - Padronização de código e formatação, exceções não-consideradas, padrões do projeto, etc.

Done	Task description
☐	Run code cleanup and code formatting
☐	Execute custom source validation tool
☐	Verify the audit log is written for all updates
☐	Make sure there are no absorbed exceptions
☐	Check for hardcoded values and convert to constants
☐	Verify that only public methods are calling setFailure()
☐	Include @ServiceEntrypoint on service API class

     - *Figure 22-12. Example of a developer code completion checklist*
     - Ferramentas de automação podem facilitar esse processo.
  2. **Checklist de Testes Unitários e Funcionais:**
     - Contém edge-cases.
       - e.g. *checar comportamento para campos faltantes*
     - Costuma ser grande.
  3. **Checklist de Software Release:**
     - Deploy pode conter erros.
     - Mudanças de configuração, adição de bibliotecas externas, updates de database, etc.
     - Cada falha pode virar um item na checklist para um deploy futuro.

Provendo Orientação

- Comunicar design e e guiar as questões principais de implementação.
- Tenta elucidar e fazer as decisões terem justificativa clara.
    - Traz a luz se as decisões e opiniões fazem sentido ou não, tanto técnico quanto de negócio.
- Decisões de usar Frameworks, Bibliotecas Gerais e Bibliotecas Específicas (e.g. leitor de PDF)
    - Quanto mais geral (e.g. framework), mais próximo do arquiteto.
    - Decisões menores não precisam do arquiteto ou desenhando ou aprovando.

23 - Habilidades de Liderança e Negociação

2025-12-24 10:10 am

Negociação e Facilitação

- Necessário entender o clima político da empresa.
- Toda decisão do arquiteto será questionada, tanto tecnicamente pelos desenvolvedores e outros arquitetos quanto stakeholders por restrições de negócio.
- Por exemplo, arquiteto tem que negociar transformar um database único em vários federados por domínio para diminuir riscos de disponibilidade.

Negociando com Stakeholders

```
- Cenário: Como convencer que uma nível de disponibilidade extremamente alto não é necessário, contrário à opinião do vice-presi
```

- Difícil navegação.
- "Quero downtime zero" não significa nada de fato mas significa que a disponibilidade é um critério crítico.
- Arquiteto deve esclarecer o significado das decisões técnicas com dados e informações.
    - Mencionar custo e tempo deve ser só o último passo da negociação, para não melar discussão já de início.
    - As outras justificativas devem vir primeiro.
- "Dividir para conquistar"
    - Qualifique as necessidades do projeto por pedaço do sistema, reduz o escopo da negociação.

Negociando com Outros Arquitetos

```
- Cenário: Dois arquitetos discordam se comunicação assíncrona (defendido pelo arquiteto-chefe) ou REST (defendido por outro arc
```

- Apelar para hierarquia é uma má opção, cria ambiente ruim.
- **Demonstração vale mais que discussão**
    - Ideal é rodar um experimento teste no ambiente de produção.
- Discussão acalorada e pessoal deve ser sempre evitada.

Negociação com Desenvolvedores

- Arquiteto não deve sair mandando a esmo nos desenvolvedores.
- Desenvolvedores não podem sentir que arquiteto está numa torre de marfim.
- Arquiteto deve sempre justificar e argumentar, não apenas ditar.
- Ideal é que por argumentação e discussão o próprio desenvolvedor chegue à conclusão do arquiteto.
    - Se houver discordância, desenvolvedor deve ter a chance de justificar e defender sua escolha, seja para descartá-la ou reforçá-la.

Arquiteto de Software Como Líder

- Metade do que é ser um bom arquiteto é saber lidar com pessoas e ter habilidades de liderança.

Os Quatro Cs da Arquitetura

- Há problemas que são complexos porque são complexos (*complexidade essencial*) e outros que são complexos porque foram feitos complexos desnecessariamente (*complexidade acidental*)
    - Desenvolvedores são atraídos pela complexidade como mariposas pela luz.
- Complexidade acidental deve ser evitada a todo custo.
- Evita-se isso com os 4 C's de um bom líder e bom time:
    - **Comunicação**
    - **Colaboração**
    - **Clareza**
    - **Concisão**

Seja Pragmático, Mas ainda Visionário

- Planejamento para o futuro mas sem largar a praticidade no presente.
- Pensamento estratégico.
- Orçamento, tempo, limitações técnicas e habilidades do time são restrições a serem consideradas.
- Não basta bolar um plano imenso complexo e mirabolante na teoria.
- Equilíbrio entre os dois não é fácil.

Liderando Times por Exemplo

- Liderança não vem do título.
- Todo problema é um problema que envolve pessoas.
- Conhecimento técnico é só uma parte.
- Tentar achar soluções colaborativamente.
- Linguagem importa, evitar imperativos.
- Pronunciar nomes das pessoas é importante.
    - Como e quando apertar as mãos também é um detalhe importante.

- Às vezes é bom transformar demandas em pedidos de favor.
- Arquiteto deve se tornar a pessoa de referência do time.
- Convite para almoços informais para discutir questões no trabalho pode ajudar no relacionamento.

Integrando com o Time de Desenvolvimento

- Arquiteto tem muitas reuniões.
    - Nem toda reunião é necessária
        - Se for só para manter arquiteto informado, anotações bastam.
        - Ou participar só das partes relevantes.
- Ajudar tech lead com responsabilidades e reuniões pode criar senso de time.
- Número de reuniões convocadas pelo arquiteto devem ser o mínimo possível.
    - O que vale mais, a reunião ou o trabalho que os desenvolvedores estariam fazendo?
    - Ser rígido ao cronograma e assunto da reunião.

_README

O'REILLY®

Fundamentos da Arquitetura de Software

Uma abordagem de engenharia

Mark Richards e Neal Ford

ALTA BOOKS
EDITORA

Table of Contents

24 - Desenvolvendo uma Trajetória de Carreira

2025-12-24 12:09 pm

- Aprendizado contínuo é necessário.
- Tudo muda muito rápido, joga-se fora muita coisa ultrapassada.

Regra dos 20 Minutos

- Abrangência é mais importante que profundidade.
- Dedicar ao menos 20 minutos do dia para aprender algo novo ou se aprofundar em algum assunto.
- Idealmente no comecinho do dia antes de tudo, não no almoço ou no fim do dia.
    - Mais difícil de surgir imprevistos.

Desenvolvendo um Radar Pessoal

- Nunca ignorar as tendências de tecnologia
- Também não se comprometer com alguma tecnologia demais, pode colapsar a qualquer momento.
- Importante ter um radar pessoal para averiguar estes movimentos.

ThoughtWorks Technology Radar

- Technology Radar, guia de tendências
- Technology Radar | Guide to technology landscape | Thoughtworks
    - Indica o que está crescendo e o que está diminuindo.
    - O que já está maduro e o que ainda é especulativo.
- Pode-se construir o próprio radar.

Usando Mídias Sociais

- Novos empregos costumam vir de pessoas conhecidas mas fora do círculo social mais próximo.
- Mídias sociais aumentam abrangência técnica, permite conhecer mais coisas acontecendo.

Últimas Palavras

- Prática é o que há de mais importante para melhorar.
- Realmente desenhar arquiteturas é indispensável.