Arquitetura de software: As partes difíceis

Análises modernas de trade-off para arquiteturas distribuídas

Neal Ford, Mark Richards,
Pramod Sadalage & Zhamak Dehghani

ALTA BOOKS
GRUPO EDITORIAL
Rio de Janeiro, 2024

Sumário

PARTE II: JUNTANDO AS COISAS

1.

O que Acontece quando Não Há "Melhores Práticas"?

Por que um tecnólogo como um arquiteto de software se apresenta em uma conferência ou escreve um livro? Porque ele descobriu o que é coloquialmente conhecido como uma "melhor prática", um termo tão usado que aqueles que o falam sofrem com uma reação negativa cada vez mais forte. Independentemente do termo, os tecnólogos escrevem livros quando descobrem uma nova solução para um problema geral e querem transmiti-la a um público mais amplo.

Mas o que acontece com esse vasto conjunto de problemas que não têm boas soluções? Existem classes inteiras de problemas na arquitetura de software que não têm boas soluções gerais, mas apresentam um conjunto confuso de trade-offs moldados para um conjunto (quase) igualmente confuso.

Quando você é um desenvolvedor de software, desenvolve habilidades excepcionais na pesquisa online de soluções para seu problema atual. Por exemplo, se você precisa descobrir como configurar uma ferramenta específica em seu ambiente, o uso especializado do Google encontra a resposta.

Mas isso não é verdade para os arquitetos.

Para os arquitetos, muitos problemas apresentam desafios únicos porque combinam o ambiente e as circunstâncias exatas de sua organização — quais são as chances de alguém ter encontrado exatamente esse cenário *e* o ter publicado em um blog ou no Stack Overflow?

Os arquitetos podem se perguntar por que existem tão poucos livros sobre arquitetura em comparação com tópicos técnicos como frameworks, APIs e assim por diante. Arquitetos raramente enfrentam problemas comuns, mas lutam constantemente com a tomada de decisões em novas situações. Para os arquitetos, todo problema é único como um floco de neve. Em muitos casos, o problema é novo não apenas dentro de uma organização em particular, mas em todo o mundo. Não existem livros ou sessões de conferência para esses problemas!

Os arquitetos não devem buscar constantemente soluções mágicas para seus problemas; elas são tão raras agora quanto eram em 1986, quando Fred Brooks cunhou a frase:

> Não há um único desenvolvimento, seja em tecnologia ou técnica de gestão, que por si só prometa uma melhoria de uma ordem de magnitude [dez vezes maior] em uma década em produtividade, confiabilidade e simplicidade.
>
> — Fred Brooks, no artigo "No Silver Bullet"

Como praticamente todos os problemas apresentam novos desafios, o verdadeiro trabalho de um arquiteto reside em sua capacidade de determinar e avaliar objetivamente o conjunto de trade-offs em ambos os lados de uma decisão consequencial para resolvê-lo da melhor maneira possível. Os autores não falam sobre "melhores soluções" (neste livro ou no mundo real) porque "melhor" implica que um arquiteto conseguiu maximizar todos os possíveis fatores concorrentes dentro do projeto. Em vez disso, nosso conselho um tanto irônico é o seguinte:

Não tente encontrar o melhor design na arquitetura de software; em vez disso, esforce-se pela combinação menos ruim de trade-offs.

Frequentemente, o melhor design que um arquiteto pode criar é o conjunto menos ruim de trade-offs — nenhuma característica de arquitetura única se destaca se estiver sozinha, mas o equilíbrio de todas as características de arquitetura concorrentes promove o sucesso do projeto.

O que levanta a questão: "Como um arquiteto pode *encontrar* a combinação menos ruim de trade-offs (e documentá-los de forma eficaz)?" Este livro trata principalmente da tomada de decisões, permitindo que os arquitetos tomem melhores decisões quando confrontados com novas situações.

Por que "As Partes Difíceis"?

Por que chamamos este livro de *Arquitetura de Software: As Partes Difíceis*? Na verdade, o "difícil" no título cumpre dupla função. Primeiro, *difícil* conota *dificuldade*, e arquitetos constantemente enfrentam problemas difíceis que literalmente (e figurativamente) ninguém enfrentou antes, envolvendo inúmeras decisões tecnológicas com implicações de longo prazo em camadas sobre o ambiente interpessoal e político onde a decisão deve ocorrer.

Segundo, *difícil* conota *dureza* — assim como na separação de *hardware* e *software*, o *duro* [hard] deve mudar muito menos porque fornece a base para *macio* [soft]. Da mesma forma, os arquitetos discutem a distinção entre *arquitetura* e *design*, em que a primeira é estrutural e o segundo é mais facilmente alterado. Assim, neste livro, falamos sobre as partes fundamentais da arquitetura.

A própria definição de arquitetura de software proporcionou muitas horas de conversa improdutiva entre seus praticantes. Uma definição favorita um tanto irônica é que "arquitetura de software é a *coisa* que é difícil de mudar mais tarde". É sobre essa *coisa* que trata nosso livro.

Dando Conselhos Atemporais Sobre Arquitetura de Software

O ecossistema de desenvolvimento de software muda e cresce de forma constante e caótica. Tópicos que estavam na moda há alguns anos foram incluídos no ecossistema e desapareceram ou foram substituídos por algo diferente/melhor. Por exemplo, há dez anos, o estilo de arquitetura predominante para grandes empresas era a arquitetura orientada a serviços e orientada por orquestração. Agora, praticamente ninguém mais constrói nesse estilo de arquitetura (por razões que descobriremos ao longo do caminho); o estilo preferido atualmente para muitos sistemas distribuídos são os microsserviços. Como e por que essa transição aconteceu?

Quando os arquitetos olham para um estilo em particular (especialmente um estilo histórico), eles devem considerar as restrições existentes que levam essa arquitetura a se tornar dominante. Na época, muitas empresas estavam se fundindo para se tornarem *empreendimentos*, com todos os problemas de integração decorrentes dessa transição. Além disso, o código aberto não era uma opção viável (geralmente por motivos políticos e não técnicos) para grandes empresas. Assim, os arquitetos enfatizaram recursos compartilhados e orquestração centralizada como solução.

No entanto, nos anos seguintes, o código aberto e o Linux tornaram-se alternativas viáveis, tornando os sistemas operacionais *comercialmente* livres. Mas o verdadeiro ponto de inflexão ocorreu quando o Linux se tornou *operacionalmente* livre com o advento de ferramentas como Puppet e Chef, que permitiram que as equipes de desenvolvimento girassem programaticamente seus ambientes como parte de uma compilação automatizada. Quando esse recurso chegou, ele promoveu uma revolução arquitetônica com microsserviços e a infraestrutura emergente de contêineres e ferramentas de orquestração como o Kubernetes.

O que isso ilustra é que o ecossistema de desenvolvimento de software se expande e evolui de maneiras completamente inesperadas. Um novo recurso leva a outro, que inesperadamente cria novos recursos. Ao longo do tempo, o ecossistema se substitui completamente, uma peça de cada vez.

Isso apresenta um problema antigo para autores de livros sobre tecnologia em geral e arquitetura de software especificamente — como podemos escrever algo que não seja ultrapassado imediatamente?

Não nos concentramos em tecnologia ou outros detalhes de implementação neste livro. Em vez disso, nos concentramos em *como* os arquitetos tomam decisões e como ponderar objetivamente os trade-offs quando apresentados a novas situações. Usamos cenários e exemplos contemporâneos para fornecer detalhes e contexto, mas os princípios subjacentes se concentram na análise de trade-offs e na tomada de decisões quando confrontados com novos problemas.

A Importância dos Dados na Arquitetura

> Os dados são uma coisa preciosa e durarão mais do que os próprios sistemas.
>
> Tim Berners-Lee

Para muitos na arquitetura, os dados são tudo. Toda empresa que constrói qualquer sistema deve lidar com dados, pois eles tendem a viver muito mais do que sistemas ou arquitetura, exigindo pensamento e design diligentes. No entanto, muitos dos instintos dos arquitetos de dados para construir sistemas fortemente acoplados criam conflitos nas arquiteturas distribuídas modernas. Por exemplo, arquitetos e DBAs devem garantir que os dados de negócios sobrevivam à quebra de sistemas monolíticos e que os negócios ainda possam obter valor de seus dados, independentemente das ondulações da arquitetura.

Já foi dito que os *dados são o ativo mais importante de uma empresa*. As empresas desejam extrair valor dos dados que possuem e estão encontrando novas maneiras de implantar dados na tomada de decisões. Todas as partes da empresa agora são orientadas por dados, desde atender clientes existentes até adquirir novos clientes, aumentar a retenção de clientes, melhorar produtos, prever vendas e outras tendências. Essa confiança nos dados significa que toda a arquitetura de software está a serviço dos dados, garantindo que os dados corretos estejam disponíveis e possam ser usados por todas as partes da empresa.

Os autores construíram muitos sistemas distribuídos algumas décadas atrás, quando estes se tornaram populares, mas a tomada de decisões em

microsserviços modernos parece mais difícil, e queríamos descobrir o porquê. Acabamos percebendo que, nos primórdios da arquitetura distribuída, ainda mantínhamos dados em um único banco de dados relacional. No entanto, em microsserviços e na aderência filosófica a um *contexto delimitado* do design orientado ao domínio, como forma de limitar o escopo do acoplamento de detalhes de implementação, os dados passaram a ser uma preocupação arquitetural, juntamente com a transacionalidade. Muitas das partes difíceis da arquitetura moderna derivam de tensões entre dados e preocupações de arquitetura, que desvendaremos na Parte I e na Parte II.

Uma distinção importante que abordamos em vários capítulos é a separação entre dados *operacionais* e *analíticos*.

Dados operacionais
: Dados usados para a operação do negócio, incluindo vendas, dados transacionais, estoque, e assim por diante. Esses dados são o que a empresa usa — se algo interromper esses dados, a organização não poderá funcionar por muito tempo. Esses tipos de dados são definidos como *Processamento de Transações Online* [em inglês, *Online Transactional Processing* — OLTP] e normalmente envolvem a inserção, atualização e exclusão de dados em um banco de dados.

Dados analíticos
: Dados usados por cientistas de dados e outros analistas de negócios para previsões, tendências e outras inteligências de negócios. Esses dados geralmente não são transacionais e não são relacionais — podem estar em um banco de dados de grafos ou instantâneos em um formato diferente de seu formato transacional original. Esses dados não são críticos para a operação do dia a dia, mas sim para a direção e decisões estratégicas de longo prazo.

Cobrimos o impacto dos dados operacionais e analíticos ao longo do livro.

Registros de Decisão de Arquitetura

Uma das formas mais eficazes de documentar as decisões de arquitetura é por meio de *Registros de Decisão de Arquitetura* [em inglês, *Architectural Decision Records* — ADRs]. Os ADRs foram primeiramente evangelizados por Michael Nygard em uma postagem de blog e posteriormente marcados como "adotar" no ThoughtWorks Technology Radar. Um ADR consiste em um arquivo de texto curto (geralmente de uma a duas páginas) descrevendo uma decisão de arquitetura específica. Embora os ADRs possam ser escritos usando-se texto simples, eles geralmente são escritos em algum tipo de formato de documento de texto como AsciiDoc ou Markdown. Alternativamente, um ADR também pode ser

escrito usando-se um modelo de página wiki. Dedicamos um capítulo inteiro a ADRs em nosso livro anterior, *Fundamentos de Arquitetura de Software*.

Aproveitaremos os ADRs como forma de documentar várias decisões de arquitetura feitas ao longo do livro. Para cada decisão de arquitetura, usaremos o seguinte formato de ADR, com a suposição de que cada ADR seja aprovado:

> *ADR: Uma frase substantiva curta contendo a decisão de arquitetura*
>
> *Contexto*
> Nesta seção, adicionaremos uma breve descrição de uma ou duas frases do problema e listaremos as soluções alternativas.
>
> *Decisão*
> Nesta seção, declararemos a decisão de arquitetura e forneceremos uma justificativa detalhada da decisão.
>
> *Consequências*
> Nesta seção do ADR, descreveremos quaisquer consequências após a aplicação da decisão e também discutiremos os trade-offs que foram considerados.

Uma lista de todos os Registros de Decisão de Arquitetura criados neste livro pode ser encontrada no Apêndice B.

Documentar uma decisão é importante para um arquiteto, mas *governar* o uso adequado da decisão é um tópico separado. Felizmente, as práticas modernas de engenharia permitem automatizar muitas preocupações comuns de governança usando *fitness functions de arquitetura*.

Fitness Functions de Arquitetura

Uma vez que um arquiteto tenha identificado o relacionamento entre os componentes e codificado isso em um projeto, como ele pode garantir que os implementadores aderirão a esse projeto? Mais amplamente, como os arquitetos podem garantir que os princípios de design que eles definem se tornem realidade se não forem eles a implementá-los?

Essas questões se enquadram no título de *governança de arquitetura*, que se aplica a qualquer supervisão organizada de um ou mais aspectos do desenvolvimento de software. Como este livro aborda principalmente a estrutura da arquitetura, abordamos como automatizar os princípios de design e qualidade por meio de fitness functions em muitos lugares.

O desenvolvimento de software evoluiu lentamente ao longo do tempo para adaptar práticas exclusivas de engenharia. Nos primórdios do desenvolvimento de software, uma metáfora de fabricação era comumente aplicada às práticas de software, tanto em casos grandes (como o processo de desenvolvimento em Cascata) quanto nos pequenos (práticas de integração em projetos). No início da década de 1990, um repensar das práticas de engenharia de desenvolvimento de software, liderada por Kent Beck e os outros engenheiros do projeto C3, chamado eXtreme Programming (XP), ilustrou a importância do feedback incremental e da automação como facilitadores-chave da produtividade do desenvolvimento de software. No início dos anos 2000, as mesmas lições foram aplicadas à interseção de desenvolvimento de software e operações, gerando o novo papel do DevOps e automatizando muitas tarefas operacionais anteriormente manuais. Assim como antes, a automação permite que as equipes sejam mais rápidas porque não precisam se preocupar com as coisas dando errado sem um bom feedback. Assim, *automação* e *feedback* tornaram-se princípios centrais para um desenvolvimento de software eficaz.

Considere os ambientes e as situações que levam a avanços na automação. Na era anterior à integração contínua, a maioria dos projetos de software incluía uma longa fase de integração. Esperava-se que cada desenvolvedor trabalhasse em algum nível de isolamento dos outros e, ao final, integrasse todo o código em uma fase de integração. Vestígios dessa prática ainda permanecem nas ferramentas de controle de versão que forçam a ramificação e impedem a integração contínua. Não surpreendentemente, existia uma forte correlação entre o tamanho do projeto e a dor da fase de integração. Ao ser pioneira na integração contínua, a equipe XP demonstrou o valor do feedback rápido e contínuo.

A revolução do DevOps seguiu um curso semelhante. À medida que o Linux e outros softwares de código aberto se tornaram "bons o suficiente" para as empresas, combinados com o advento de ferramentas que permitiam a definição programática de (eventualmente) máquinas virtuais, o pessoal de operações percebeu que poderia automatizar as definições de máquinas e muitas outras tarefas repetitivas.

Em ambos os casos, os avanços em tecnologia e insights levaram à automatização de um trabalho recorrente que era tratado por uma função cara — que descreve o estado atual da governança de arquitetura na maioria das organizações. Por exemplo, se um arquiteto escolhe um determinado estilo de arquitetura ou meio de comunicação, como ele pode garantir que um desenvolvedor o implemente corretamente? Quando feito manualmente, os arquitetos realizam revisões de código ou talvez mantenham comitês de revisão de arquitetura para avaliar o estado da governança. No entanto, assim como na configuração manual de computadores em operação, detalhes importantes podem facilmente cair em revisões superficiais.

Usando Fitness Functions

No livro de 2017 *Building Evolutionary Architectures* [*Construindo Arquiteturas Evolucionárias*, em tradução livre], os autores (Neal Ford, Rebecca Parsons e Patrick Kua) definiram o conceito de uma *fitness function de arquitetura*: qualquer mecanismo que realize uma avaliação objetiva da integridade de alguma característica da arquitetura ou combinação de características da arquitetura.

Qualquer mecanismo
: Os arquitetos podem usar uma ampla variedade de ferramentas para implementar fitness functions; mostraremos vários exemplos ao longo do livro. Por exemplo, existem bibliotecas de teste dedicadas a testar a estrutura da arquitetura, os arquitetos podem usar monitores para testar as características da arquitetura operacional, como desempenho ou escalabilidade, e frameworks de engenharia de caos testam a confiabilidade e resiliência — todos exemplos de fitness functions.

Avaliação de integridade objetiva
: Um facilitador-chave para a governança automatizada está nas definições objetivas das características da arquitetura. Por exemplo, um arquiteto não pode especificar que deseja um site de "alto desempenho"; ele deve fornecer um valor de objeto que possa ser medido por um teste, monitorador ou outra fitness function.

  Os arquitetos devem estar atentos às *características da arquitetura composta* — aquelas que não são objetivamente mensuráveis, mas, na verdade, são compostas de outras coisas mensuráveis. Por exemplo, "agilidade" não é mensurável, mas se um arquiteto começar a separar o termo amplo de *agilidade*, o objetivo é que as equipes possam responder com rapidez e confiança às mudanças, seja no ecossistema ou no domínio. Assim, um arquiteto pode encontrar características mensuráveis que contribuam para a agilidade: implantabilidade, testabilidade, tempo de ciclo, e assim por diante. Muitas vezes, a falta de capacidade de medir uma característica de arquitetura indica uma definição muito vaga. Se os arquitetos se esforçam para obter propriedades mensuráveis, isso permite que eles automatizem a aplicação da fitness function.

Alguma característica de arquitetura ou combinação de características de arquitetura
: Esta característica descreve os dois escopos para fitness functions.

Atômica
: Essas fitness functions lidam com uma única característica de arquitetura isoladamente. Por exemplo, uma fitness function que verifica os ciclos de componentes em uma base de código tem escopo atômico.

Holística

> O contrário são as fitness functions *holísticas*, que validam uma combinação de características de arquitetura. Um aspecto complicado das características da arquitetura é a sinergia que elas às vezes exibem com outras características da arquitetura. Por exemplo, se um arquiteto deseja melhorar a segurança, existe uma boa chance de que isso afete o desempenho. Da mesma forma, escalabilidade e elasticidade às vezes estão em desacordo — o suporte a muitos usuários simultâneos pode dificultar o manuseio de surtos repentinos. As fitness functions holísticas exercem uma combinação de características de arquitetura interligadas para garantir que o efeito combinado não afete negativamente a arquitetura.

Um arquiteto implementa fitness functions para construir proteções em torno de mudanças inesperadas nas características da arquitetura. No mundo do desenvolvimento de software Ágil, os desenvolvedores implementam testes de unidade, funcionais e de aceitação do usuário para validar diferentes dimensões do design de *domínio*. No entanto, até agora, não existia nenhum mecanismo semelhante para validar a parte das *características da arquitetura* do projeto. Na verdade, a separação entre fitness functions e testes de unidade fornece uma boa diretriz de escopo para arquitetos. As fitness functions validam as características da arquitetura, não os critérios de domínio; testes de unidade são o oposto. Assim, um arquiteto pode decidir se uma fitness function ou um teste de unidade é necessário fazendo a pergunta: "É necessário algum conhecimento de domínio para executar este teste?" Se a resposta for "sim", então um teste de aceitação de unidade/função/usuário é apropriado; se "não", então é necessária uma fitness function.

Por exemplo, quando os arquitetos falam sobre *elasticidade*, trata-se da capacidade do aplicativo de resistir a um surto repentino de usuários. Observe que o arquiteto não precisa saber detalhes sobre o domínio — pode ser um site de comércio eletrônico, um jogo online ou qualquer outra coisa. Assim, a *elasticidade* é uma preocupação de arquitetura e está dentro do escopo de uma fitness function. Se, por outro lado, o arquiteto quiser validar as partes corretas de um endereço de correspondência, isso é coberto por um teste tradicional. É claro que essa separação não é puramente binária — algumas funções de aptidão vão tocar no domínio, e vice-versa, mas os diferentes objetivos fornecem uma boa maneira de separá-los mentalmente.

Aqui estão alguns exemplos para tornar o conceito menos abstrato.

Um objetivo comum do arquiteto é manter uma boa integridade estrutural interna na base de código. No entanto, forças malévolas trabalham contra as boas intenções do arquiteto em muitas plataformas. Por exemplo, ao codificar em qualquer ambiente de desenvolvimento Java ou .NET popular, assim que um

desenvolvedor faz referência a uma classe ainda não importada, o IDE apresenta uma caixa de diálogo perguntando ao desenvolvedor se ele deseja importar automaticamente a referência. Isso ocorre com tanta frequência que a maioria dos programadores desenvolve o hábito de afastar a caixa de diálogo de importação automática como uma ação reflexa. No entanto, importar classes ou componentes arbitrariamente entre si significa um desastre para a modularidade. Por exemplo, a Figura 1-1 ilustra um antipadrão particularmente prejudicial que os arquitetos desejam evitar.

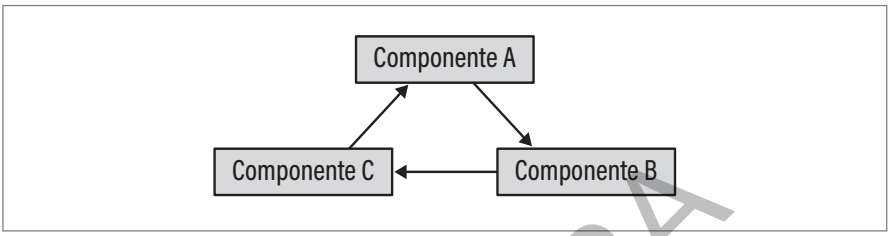

Figura 1-1. Dependências cíclicas entre componentes.

Na Figura 1-1, cada componente faz referência a algo nos outros. Ter uma rede de componentes como essa prejudica a modularidade porque um desenvolvedor não pode reutilizar um único componente sem também trazer os outros. E, claro, se os outros componentes forem acoplados a outros componentes, a arquitetura tende cada vez mais para o antipadrão da Grande Bola de Lama. Como os arquitetos podem controlar esse comportamento sem estar constantemente preocupados com os desenvolvedores sempre prontos a atirar? As revisões de código ajudam, mas acontecem muito tarde no ciclo de desenvolvimento para serem eficazes. Se um arquiteto permitir que uma equipe de desenvolvimento importe desenfreadamente toda a base de código por uma semana até a revisão do código, já terá ocorrido um dano sério na base de código.

A solução para este problema é escrever uma fitness function para evitar ciclos de componentes, como mostrado no Exemplo 1-1.

Exemplo 1-1. Fitness function para detectar ciclos de componentes

```
public class CycleTest {
   private JDepend jdepend;

   @BeforeEach
   void init() {
       jdepend = new JDepend();
         jdepend.addDirectory("/path/to/project/persistence/classes");
         jdepend.addDirectory("/path/to/project/web/classes");
         jdepend.addDirectory("/path/to/project/thirdpartyjars");
   }
```