O'REILLY®

Microsserviços prontos para a produção

CONSTRUINDO SISTEMAS PADRONIZADOS EM UMA ORGANIZAÇÃO DE ENGENHARIA DE SOFTWARE

novatec

Susan J. Fowler

Microsserviços prontos para a produção

CONSTRUINDO SISTEMAS PADRONIZADOS EM UMA ORGANIZAÇÃO DE ENGENHARIA DE SOFTWARE

Susan J. Fowler

Novatec

Tradução em português autorizada da edição em inglês da obra Production-Ready Microservices, ISBN 9781491965979 © 2017 Susan J. Fowler. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

Editor: Rubens Prates
Tradução: Claudio Adas
Revisão gramatical: Marta Almeida de Sá
Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-747-3

Histórico de edições impressas:
Novembro/2018 Primeira reimpressão
Setembro/2017 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Email: novatec@novatec.com.br
Site: www.novatec.com.br
Twitter: twitter.com/novateceditora
Facebook: facebook.com/novatec
LinkedIn: linkedin.com/in/novatec

Sumário

Prefácio

Este livro nasceu a partir de uma iniciativa de disponibilidade de produção que comecei a implantar vários meses depois de entrar na Uber Technologies como engenheira de confiabilidade (SRE). A gigantesca e monolítica API do Uber estava lentamente sendo dividida em microsserviços, e quando entrei na empresa havia mais de mil microsserviços que surgiram a partir da divisão da API e foram executados junto a ela. Cada um desses microsserviços foi projetado, construído e mantido por uma equipe de desenvolvimento própria, e mais de 85% desses serviços tinham pouco ou nenhum envolvimento do SRE[1] e nenhum acesso aos recursos do SRE.

Contratar SREs e montar equipes de SRE é uma tarefa absurdamente difícil, pois SREs são provavelmente o tipo de engenheiro mais difícil de encontrar: a engenharia de confiabilidade é um campo relativamente novo, e SREs precisam ser especialistas (ao menos, em certo nível) em engenharia de software, engenharia de sistemas e arquitetura de sistemas distribuídos. Não era possível prover rapidamente todas as equipes com sua própria equipe de SRE integrada, e foi assim que minha equipe (a Equipe de Consultoria de SRE) surgiu. Nossa diretiva era simples: encontrar um modo de implantar altos padrões em 85% dos microsserviços que não tinham qualquer envolvimento com SRE.

Nossa missão era simples, e a diretiva era vaga o suficiente para permitir que eu e minha equipe tivéssemos uma liberdade considerável para definir um conjunto de padrões que seria seguido por todos os microsserviços do Uber. Definir altos padrões que pudessem ser aplicados a todos os microsserviços sendo executados dentro dessa grande organização de engenharia de software não foi fácil, portanto, com a ajuda de meu incrível colega Rick Boone (cujos altos padrões para os microsserviços suportados por ele inspiraram este livro), criei uma checklist detalhada dos padrões que eu achava que todo serviço do Uber deveria atender antes que ele fosse autorizado a receber tráfego de produção.

Isso exigia identificar um conjunto de princípios gerais e abrangentes que

cobrisse todos os requisitos específicos, e assim nós definimos oito princípios: todo microsserviço do Uber deveria ser *estável*, *confiável*, *escalável*, *tolerante a falhas*, *de alto desempenho*, *monitorado*, *documentado* e *preparado para qualquer catástrofe*. Sob cada um desses princípios havia critérios separados que definiam o que significava um serviço ser estável, confiável, escalável, tolerante a falhas, de alto desempenho, monitorado, documentado e preparado para qualquer catástrofe. O importante é que exigimos que cada princípio fosse quantificável e que cada critério nos fornecesse resultados mensuráveis que aumentassem dramaticamente a disponibilidade de nossos microsserviços. Um serviço que atendesse a esses critérios e se adequasse a esses requisitos seria considerado *pronto para produção*.

O próximo passo era implantar esses padrões nas equipes de maneira eficaz e eficiente. Criei um processo cuidadoso no qual as equipes de SRE se encontravam com serviços críticos ao negócio (serviços cuja interrupção derrubaria a aplicação), revisavam a arquitetura com outras equipes, auditavam seus serviços (verificações simples que respondessem com um "sim" ou "não" conforme o serviço atendesse a cada um dos requisitos de disponibilidade de produção), criavam roteiros detalhados (guias passo a passo que descreviam como tornar o serviço em questão pronto para produção) e atribuíam notas de disponibilidade de produção para cada serviço.

Revisar a arquitetura era a parte mais importante do processo: minha equipe reuniria cada desenvolvedor que estivesse trabalhando em um serviço em uma sala de conferências e pediria que eles descrevessem com diagramas a arquitetura de seu serviço em trinta minutos ou menos. Isso permitia que minha equipe e a equipe anfitriã identificassem de forma rápida e fácil o ponto e o motivo da falha do serviço: quando um microsserviço era diagramado em toda a sua glória (pontos finais, fluxos de solicitação, dependências e tudo mais), cada ponto de falha se destacava como um polegar inchado.

Cada revisão de arquitetura produzia bastante material. Depois de cada revisão, nós analisávamos a checklist para ver se o serviço atendia aos requisitos de disponibilidade de produção, e então compartilhávamos esta auditoria com os gerentes e desenvolvedores da equipe. Uma pontuação foi acrescentada às auditorias quando percebi que a ideia *pronto ou não*

para produção simplesmente não era granular o suficiente para ser útil quando fôssemos avaliar a disponibilidade de produção dos serviços, portanto cada requisito recebia certo número de pontos e era dada uma nota geral ao serviço.

Essas auditorias resultaram em roteiros que continham uma lista de requisitos de disponibilidade de produção que não eram cumpridos pelo serviço, junto aos links para informações sobre recentes interrupções causadas pelo não cumprimento daquele requisito, descrições do trabalho que precisava ser feito para cumprir o requisito, a um link para uma tarefa aberta e ao nome do(s) desenvolvedor(es) designado(s) para a tarefa relevante.

Depois de fazer minha própria verificação de disponibilidade de produção desse processo (também conhecida como “disponibilidade de produção de um processo como serviço” de Susan Fowler), eu sabia que o próximo passo deveria ser a automação de todo o processo a ser executado em todos os microsserviços do Uber o tempo todo. No momento em que escrevo este livro, todo este sistema de disponibilidade de produção está sendo automatizado por uma incrível equipe de SRE do Uber comandada pela destemida Roxana del Toro.

Cada requisito de disponibilidade de produção dentro dos padrões de disponibilidade de produção e os detalhes de sua implementação são o resultado de inúmeras horas de trabalho cuidadoso e criterioso conduzido por mim e meus colegas na Uber SRE. Ao criar a lista de requisitos e tentar implementá-los em todos os microsserviços do Uber, tomamos inúmeras notas, discutimos demoradamente uns com os outros e pesquisamos tudo que havia na atual literatura de microsserviços (que é muito esparsa e quase inexistente). Eu me reuni com uma grande variedade de equipes de desenvolvedores de microsserviços, tanto no Uber quanto em outras empresas, tentando determinar como microsserviços podem ser padronizados e se existe um conjunto universal de princípios de padronização que poderia ser aplicado a qualquer microsserviço em todas as empresas e que produzisse resultados mensuráveis e capazes de impactar os negócios. A partir dessas notas, discussões, reuniões e pesquisas, surgiram as bases deste livro.

Só depois que comecei a compartilhar meu trabalho com engenheiros de confiabilidade e engenheiros de software de outras empresas da área da

grande São Francisco, conhecida como Bay Area, é que percebi como isso era novidade não apenas no mundo de SRE, mas na indústria de tecnologia como um todo. Quando engenheiros começaram a me solicitar todo tipo de informação e orientação possível sobre como padronizar seus microsserviços e torná-los prontos para produção, comecei a escrever.

No momento em que escrevo este livro, há pouca literatura sobre padronização de microsserviços e poucos guias sobre como manter e construir ecossistemas de microsserviços. Além disso, não há livros que respondam à dúvida que muitos engenheiros têm depois de dividir suas aplicações monolíticas em microsserviços: o que fazer em seguida? A ambiciosa meta deste livro é preencher esta lacuna e responder exatamente a esta questão. Resumindo, este é o livro que eu gostaria de ter tido em mãos quando comecei a padronizar os microsserviços do Uber.

Para quem este livro é escrito

Este livro é escrito principalmente para engenheiros de software e engenheiros de confiabilidade que dividiram aplicações monolíticas e agora se perguntam “o que fazer em seguida?” ou estão construindo microsserviços a partir do zero e querem projetar microsserviços estáveis, confiáveis, escaláveis, tolerantes a falhas e de alto desempenho desde o início.

Entretanto a relevância dos princípios contidos neste livro não se limita ao público principal. Muitos dos princípios, desde um bom monitoramento até um bem-sucedido dimensionamento da aplicação, podem ser usados para melhorar serviços e aplicações de qualquer tamanho e arquitetura em qualquer organização de software. Engenheiros, gerentes de engenharia, gerentes de produto e executivos de alto escalão poderão achar este livro útil por várias razões, incluindo definir padrões para suas aplicações, entender as mudanças na estrutura organizacional resultantes de decisões de arquitetura ou determinar e implantar a visão operacional e de arquitetura de suas organizações de engenharia de software.

Eu suponho que o leitor esteja familiarizado com os conceitos básicos de microsserviços, com a arquitetura de microsserviços e com os fundamentos de modernos sistemas distribuídos – leitores que compreendem bem esses conceitos obterão o máximo deste livro. Para

leitores não familiarizados com esses tópicos eu dediquei o primeiro capitulo a uma breve visão geral da arquitetura de microsserviços, do ecossistema de microsserviços, dos desafios organizacionais que acompanham os microsserviços e dos aspectos essenciais e práticos de dividir uma aplicação monolítica em microsserviços.

O que este livro não é

Este livro não é um guia passo a passo: ele não é um tutorial explícito sobre como fazer cada uma das atividades tratadas em seus capítulos. Escrever tal tutorial exigiria muitos volumes: cada seção de cada um dos capítulos deste livro poderia ser expandida em seu próprio livro.

Como resultado, este é um livro altamente abstrato, escrito para ser genérico o suficiente para que as lições aprendidas aqui possam ser aplicadas a praticamente qualquer microsserviço em quase qualquer organização, porém é específico e granular o suficiente para que possa ser incorporado a uma organização de engenharia de software e oferecer uma orientação real e tangível sobre como melhorar e padronizar microsserviços. Uma vez que o ecossistema de microsserviços difere de empresa para empresa, não há vantagem em adotar uma abordagem passo a passo ou educacional. Em vez disso, decidi introduzir conceitos, explicar sua importância para construir microsserviços prontos para produção, oferecer exemplos de cada conceito e compartilhar estratégias para sua implementação.

E, o que é mais importante, este livro não é uma descrição enciclopédica de todas as formas possíveis de construir e executar microsserviços e ecossistemas de microsserviços. Sou a primeira a admitir que há muitas maneiras válidas de construir e executar microsserviços e ecossistemas de microsserviços. (Por exemplo, existem muitas maneiras diferentes de testar novas versões, além da abordagem staging/pré-release/produção, em que um pequeno grupo de usuários recebe a nova versão, introduzida e defendida no Capítulo 3, *Estabilidade e confiabilidade*). Porém algumas maneiras são melhores que outras, e tentei o máximo possível apresentar apenas as melhores maneiras de construir e executar microsserviços prontos para produção e aplicar cada princípio de disponibilidade de produção a organizações de engenharia de software.

Além disso, a tecnologia se move e muda de forma incrivelmente rápida. Sempre que possível eu tentei evitar que o leitor ficasse limitado a uma tecnologia ou a um conjunto de tecnologias desenvolvidas para sua implementação. Por exemplo, em vez de defender que todo microsserviço use Kafka para gravação de logs, eu apresento os aspectos importantes da gravação de logs pronta para produção e deixo que o leitor escolha a tecnologia específica e a implementação real.

Finalmente, este livro não é uma descrição da organização de engenharia de software do Uber. Os princípios, padrões, exemplos e as estratégias não são específicos do Uber nem inspirados exclusivamente pelo Uber: eles foram desenvolvidos e inspirados por microsserviços de muitas empresas de tecnologia e podem ser aplicados a qualquer ecossistema de microsserviços. Ele não é um relato descritivo ou histórico, mas um guia prescritivo para construir microsserviços prontos para produção.

Como usar este livro

Há várias maneiras de usar este livro.

A primeira abordagem é a que exige menos envolvimento: ler apenas os capítulos de seu interesse e folhear (ou pular) o restante. Esta abordagem tem muitas vantagens: você será apresentado a novos conceitos, terá insights sobre conceitos com os quais você pode estar familiarizado e descobrirá novas maneiras de pensar sobre aspectos de engenharia de software e arquitetura de microsserviços que podem ser úteis em sua vida e em seu trabalho cotidiano.

Outra abordagem exige um pouco mais de envolvimento. Você percorrerá as páginas do livro e lerá com atenção as seções que são relevantes para suas necessidades, aplicando alguns dos princípios e padrões aos seus microsserviços. Por exemplo, se seus microsserviços estiverem precisando de um melhor monitoramento, você poderá folhear a maior parte do livro, lendo apenas o Capítulo 6, *Monitoramento*, com atenção e então usar o material contido neste capítulo para melhorar os processos de monitoramento, alerta e resposta a interrupções de seus serviços.

A última abordagem é (provavelmente) a mais recompensadora e aquela que deve ser adotada se o seu objetivo for padronizar completamente o microsserviço sob sua responsabilidade ou todos os microsserviços de sua

empresa, tornando-os realmente prontos para produção. Se o seu objetivo for tornar seus microsserviços estáveis, confiáveis, escaláveis, tolerantes a falhas, de alto desempenho, adequadamente monitorados, bem documentados e preparados para qualquer catástrofe, recomenda-se usar esta abordagem. Para isso, cada capítulo deve ser lido com atenção, cada padrão deve ser compreendido e cada requisito, ajustado e aplicado de acordo com as necessidades de seus microsserviços.

Ao final de cada capítulo sobre padronização (Capítulos 3 a 7), você encontrará uma seção chamada “Avalie seu microsserviço”, que contém uma pequena lista de perguntas a serem feitas sobre o seu microsserviço. As perguntas estão organizadas por tópico de modo que você (o leitor) possa rapidamente selecionar as que forem relevantes aos seus objetivos, respondê-las para o seu microsserviço e então determinar quais medidas tomar para tornar seu microsserviço pronto para produção. Ao final do livro, você encontrará dois apêndices (Apêndice A – *Checklist de disponibilidade de produção* – e Apêndice B – *Avalie seu microsserviço*) que o ajudarão a acompanhar os padrões de disponibilidade de produção e as perguntas da seção “Avalie seu microsserviço”, que estão espalhadas pelo livro.

Como este livro é estruturado

Como o título sugere, o Capítulo 1, *Microsserviços*, é uma introdução a microsserviços. Ele aborda os aspectos básicos da arquitetura de microsserviços, alguns detalhes de como dividir uma aplicação monolítica em microsserviços, introduz as quatro camadas de um ecossistema de microsserviços e conclui com uma seção dedicada a esclarecer alguns dos desafios organizacionais e os compromissos que resultam da adoção da arquitetura de microsserviços.

No Capítulo 2, *Disponibilidade de produção*, são apresentados os desafios da padronização de microsserviços e os oito padrões de disponibilidade de produção, todos motivados pela disponibilidade do microsserviço.

O Capítulo 3, *Estabilidade e confiabilidade*, trata dos princípios da construção de microsserviços estáveis e confiáveis. Também são cobertos tópicos como ciclo de desenvolvimento, deployment pipeline, como lidar com dependências, roteamento e descoberta (discovery), e a depreciação e

desativação estáveis e confiáveis de microsserviços.

O Capítulo 4, *Escalabilidade e desempenho*, se concentra nos requisitos para construir microsserviços escaláveis e de bom desempenho, incluindo conhecer o grau de crescimento dos microsserviços, o uso eficiente de recursos, ter consciência dos recursos, capacidade de planejamento, dimensionamento de dependência, gerenciamento de tráfego, tratamento e processamento de tarefas e armazenamento escalável de dados.

O Capítulo 5, *Tolerância a falhas e prontidão para catástrofes*, aborda os princípios para criar microsserviços que sejam tolerantes a falhas e que estejam preparados para qualquer catástrofe, incluindo catástrofes comuns e cenários de falhas, estratégias para detecção e correção de falhas, as peculiaridades do teste de resiliência e maneiras de lidar com incidentes e interrupções.

O Capítulo 6, *Monitoramento*, trata dos detalhes essenciais do monitoramento de microsserviços e de como evitar as complexidades do monitoramento de microsserviços por meio de padronização. Também são tratados neste capítulo tópicos como logging, criação de dashboards de controle úteis e tratamento adequado de alertas.

Por fim, porém não menos importante, temos o Capítulo 7, *Documentação e compreensão*, que trata da documentação adequada dos microsserviços e das maneiras de aumentar a compreensão da arquitetura e da operação por parte das equipes de desenvolvimento em toda a organização, e também contém estratégias práticas para implementar padrões de disponibilidade de produção em uma organização de engenharia de software.

Há dois apêndices no final deste livro. O Apêndice A, *Checklist de disponibilidade de produção*, é a checklist descrita no final do Capítulo 7, *Documentação e compreensão*, e é um resumo conciso de todos os padrões de disponibilidade de produção que estão espalhados pelo livro, junto a seus requisitos correspondentes. O Apêndice B, *Avalie seu microsserviço*, é um conjunto de todas as perguntas “Avalie seu microsserviço” encontradas nas seções correspondentes no final dos capítulos 3 a 7.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico

Indica novos termos, URLs, endereços de e-mail, nomes de arquivo e extensões de arquivo.

Largura constante

Usada para listagens de programas, assim como dentro de parágrafos para se referir a elementos de programa, como nomes de variáveis ou funções, databases, tipos de dados, variáveis de ambiente, declarações e palavras-chave.

Largura constante em negrito

Exibe comandos ou outros textos que devem ser digitados literalmente pelo usuário.

Largura constante em itálico

Exibe texto que deve ser substituído por valores fornecidos pelo usuário ou valores determinados pelo contexto.


Este elemento indica uma dica ou sugestão.


Este elemento indica um aviso ou uma advertência.

Como entrar em contato conosco

Envie seus comentários e suas dúvidas sobre este livro à editora pelo email: *novatec@novatec.com.br*.

Temos uma página web para este livro, na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

• Página da edição traduzida para o português

http://www.novatec.com.br/livros/microsservicos-para-producao

• Página da edição original em inglês

http://bit.ly/prod-ready_microservices

Para obter mais informações sobre os livros da Novatec, acesse nosso site em: *http://www.novatec.com.br*.

Agradecimentos

Este livro é dedicado à minha melhor metade, Chad Rigetti, que deixou um pouco de lado a construção de computadores quânticos para ouvir todas as minhas reclamações sobre microsserviços e que me encorajou alegremente durante todo o processo. Eu não poderia ter escrito este livro sem todo o seu amor e apoio incondicional.

Ele também é dedicado às minhas irmãs Martha e Sara, cuja determinação, resiliência, coragem e alegria me inspiraram em todos os momentos e aspectos da minha vida, e também a Shalon Van Tine, que tem sido minha amiga mais próxima e apoiadora mais entusiasmada há muitos anos.

Sou muito grata a todos aqueles que fizeram comentários sobre as versões iniciais deste livro, aos meus colegas do Uber e aos engenheiros que trabalharam corajosamente para implementar os princípios e estratégias contidos neste livro em suas próprias organizações de engenharia de software. Sou especialmente grata a Roxana del Toro, Patrick Schork, Rick Boone, Tyler Dixon, Jonah Horowitz, Ryan Rix, Katherine Hennes, Ingrid Avendano, Sean Hart, Shella Stephens, David Campbell, Jameson Lee, Jane Arc, Eamon Bisson-Donahue e Aimee Gonzalez.

Nada disso teria sido possível sem Brian Foster, Nan Barber, os revisores técnicos e o restante da incrível equipe da editora O’Reilly. Eu não poderia ter escrito este livro sem vocês.

---

1 N.T.: SRE (Site reliability engineering) (SRE) é uma disciplina criada pelo Google, que incorpora aspectos de engenharia de software, para criar software escaláveis e altamente confiáveis. Fonte: Wikipedia.

CAPÍTULO 1

Microsserviços

Nos últimos anos, o setor de tecnologia tem testemunhado uma rápida mudança na arquitetura aplicada e prática de sistemas distribuídos, o que tem levado os gigantes da indústria (como Netflix, Twitter, Amazon, eBay e Uber) a abandonar a construção de aplicações monolíticas e adotar a arquitetura de microsserviços. Embora os conceitos fundamentais por trás dos microsserviços não sejam novos, a aplicação contemporânea da arquitetura de microsserviços é atual, e sua adoção tem sido motivada em parte pelos desafios de escalabilidade, falta de eficiência, velocidade lenta de desenvolvimento e dificuldades em adotar novas tecnologias que surgem quando sistemas complexos de software estão contidos em e são implantados como uma grande aplicação monolítica.

Adotar a arquitetura de microsserviços, seja a partir do zero ou dividindo uma aplicação monolítica existente em microsserviços desenvolvidos e implantados independentemente, resolve esses problemas. Com a arquitetura de microsserviços, uma aplicação pode ser facilmente escalada tanto horizontalmente quanto verticalmente, a produtividade e a velocidade do desenvolvedor aumentam dramaticamente e tecnologias antigas podem facilmente ser trocadas pelas mais recentes.

Como veremos neste capítulo, a adoção da arquitetura de microsserviços pode ser considerada um passo natural do processo de escalar uma aplicação. A divisão de uma aplicação monolítica em microsserviços é motivada por questões de escalabilidade e eficiência, mas os microsserviços apresentam desafios próprios. Um ecossistema de microsserviços bem-sucedido e escalável precisa de uma infraestrutura estável e sofisticada. Além disso, a estrutura organizacional de uma empresa que está adotando os microsserviços precisa ser radicalmente alterada para suportar a arquitetura de microsserviços, e as estruturas de equipes resultantes podem levar à segregação e dispersão. Os maiores desafios trazidos pela arquitetura de microsserviços, entretanto, são a

necessidade de padronização da arquitetura dos próprios serviços, junto aos requisitos de cada microsserviço para garantir confiança e disponibilidade.

De monólitos a microsserviços

Quase todas as aplicações de software escritas atualmente podem ser divididas em três elementos distintos: um lado *frontend* (ou *lado do cliente*), um lado de *backend* e algum tipo de *armazenamento de dados* (Figura 1.1). As solicitações são feitas à aplicação a partir do lado do cliente, o código de backend faz o trabalho pesado e quaisquer dados relevantes que tiverem de ser armazenados ou acessados (quer temporariamente na memória ou permanentemente em um database) são enviados para o local em que os dados estão armazenados ou obtidos nesse local. Chamaremos isso de *arquitetura de três camadas*.

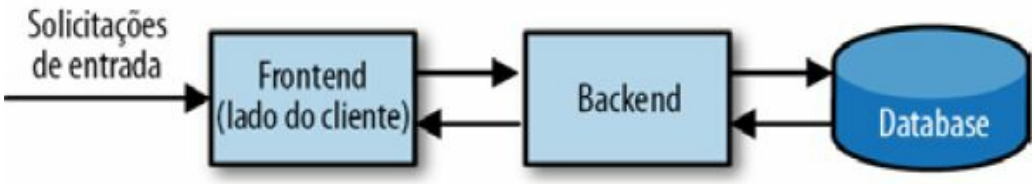


Figura 1.1. Arquitetura de três camadas.

Existem três maneiras diferentes de combinar estes elementos para criar uma aplicação. A maioria das aplicações coloca os dois primeiros elementos em um código-base (ou repositório), onde é armazenado e executado todo código do lado do cliente e de backend como um arquivo executável, com um database separado. Outras aplicações separam todo código frontend do código de backend e os armazena como executáveis lógicos separados, acompanhados por um database externo. Aplicações que não requerem um database externo e armazenam todos os dados na memória tendem a combinar todos os três elementos em um único repositório. Independentemente da forma como esses três elementos são divididos ou combinados, a própria *aplicação* é considerada a soma desses três elementos distintos.

Geralmente as aplicações são projetadas, construídas e executadas dessa forma desde o início de seus ciclos de vida e a arquitetura da aplicação é tipicamente independente do produto oferecido pela empresa ou do propósito da própria aplicação. Esses três elementos de arquitetura que

fazem parte da arquitetura de três camadas estão presentes em todos os sites, todos os aplicativos de smartphone, toda aplicação de backend, de frontend e nas estranhas e gigantescas aplicações corporativas, e são encontrados como uma das permutações descritas.

Nos primeiros estágios, quando uma empresa é jovem, suas aplicações são simples e o número de desenvolvedores que contribuem para o código-base é pequeno; os desenvolvedores tipicamente compartilham o esforço de contribuir para e manter o código-base. À medida que a empresa cresce, mais desenvolvedores são contratados, novos recursos são acrescentados à aplicação, com três consequências significativas.

Primeiro vem um aumento da carga de trabalho (workload) operacional. Em geral, o trabalho operacional é aquele associado à execução e manutenção da aplicação. Isso normalmente leva à contratação de engenheiros operacionais (administradores de sistema, engenheiros "TechOps" e os chamados engenheiros "DevOps") que assumem a maioria das tarefas operacionais, como aquelas relacionadas a hardware, monitoramento e serviços de prontidão.

A segunda, é um simples resultado matemático: acrescentar novos recursos à sua aplicação aumenta o número de linhas de código e a complexidade da própria aplicação.

A terceira é o necessário dimensionamento horizontal e/ou vertical da aplicação. O aumento de tráfego resulta em significativas demandas de escalabilidade e desempenho sobre a aplicação, exigindo que mais servidores hospedem a aplicação. Outros servidores são adicionados, uma cópia da aplicação é instalada em cada servidor, e balanceadores de carga (load balancers) são ativados para que as solicitações sejam distribuídas adequadamente entre os servidores que hospedam a aplicação (veja a Figura 1.2, contendo um elemento de frontend que pode englobar sua própria camada de balanceamento de carga, uma camada de balanceamento de carga no backend e os servidores de backend). O dimensionamento vertical torna-se uma necessidade à medida que a aplicação começa a processar um número maior de tarefas relacionadas à sua diversa gama de recursos, portanto a aplicação é instalada em servidores maiores e mais potentes, capazes de lidar com as demandas de CPU e memória (Figura 1.3).

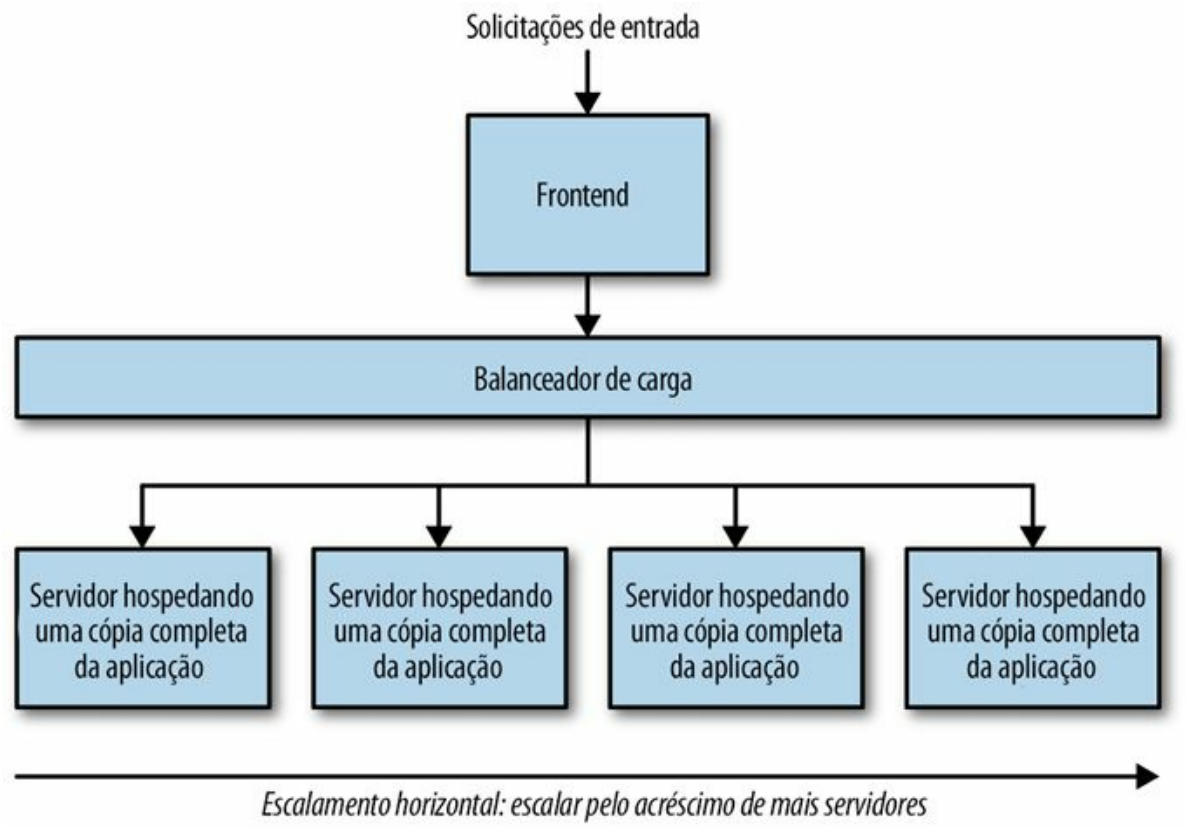


Figura 1.2. Dimensionando uma aplicação horizontalmente.

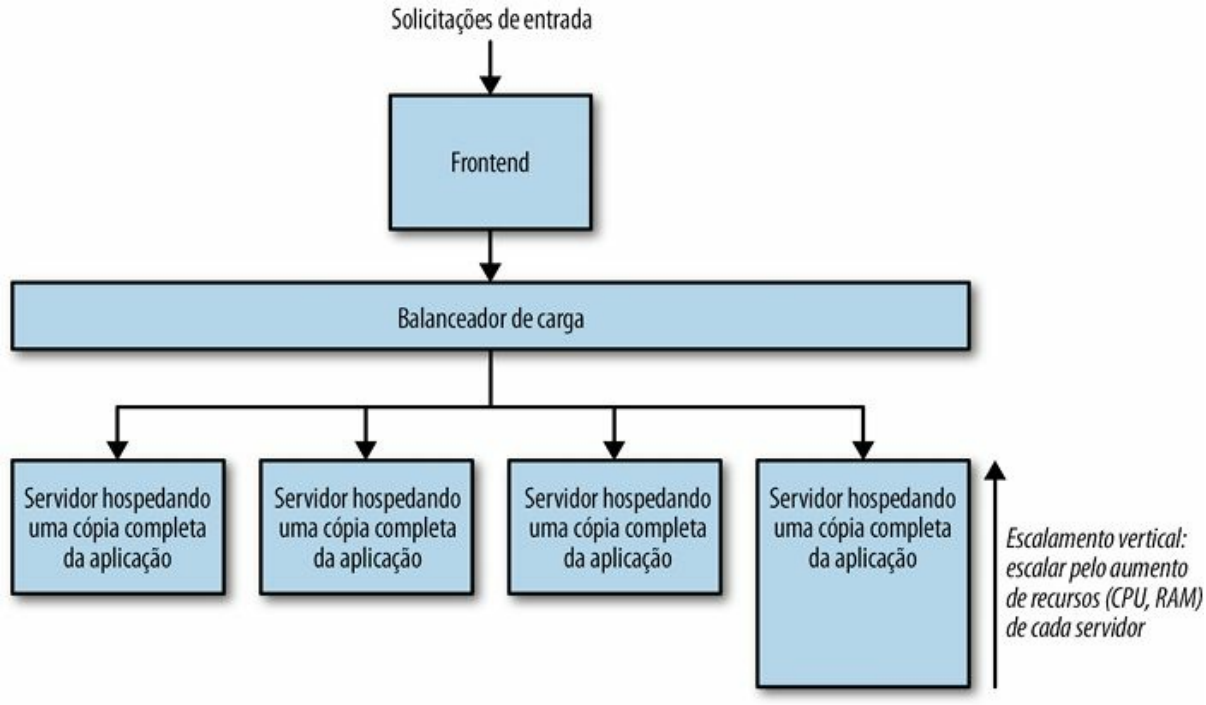


Figura 1.3. Dimensionando uma aplicação verticalmente.

À medida que a empresa cresce e o número de engenheiros ultrapassa a ordem de grandeza das centenas, tudo começa a ficar mais complicado. Graças a todos os recursos, aos patches e às correções adicionadas ao

código-base pelos desenvolvedores, agora a aplicação contém milhares e milhares de linhas de código. A complexidade da aplicação cresce constantemente, e centenas (quando não milhares) de testes precisam ser escritas para garantir que qualquer alteração feita (mesmo uma alteração de uma ou duas linhas) não comprometa a integridade de milhares de linhas de código existentes. O desenvolvimento e a implantação tornam-se um pesadelo, o teste passa a ser um fardo e a implantação das correções mais cruciais é adiada, aumentando rapidamente a defasagem técnica. Aplicações cujo ciclo de vida se encaixa neste padrão (para o bem ou para o mal) são carinhosamente (e adequadamente) conhecidas pela comunidade de software como *monólitos*.

Claro que nem toda aplicação monolítica é ruim ou sofre dos problemas listados, mas são muito raros os monólitos que não apresentam esses problemas em algum ponto de seu ciclo de vida (na minha experiência). A razão pela qual a maioria dos monólitos é suscetível a esses problemas é a natureza de um monólito ser diretamente oposta à *escalabilidade* em termos mais gerais possíveis. Escalabilidade requer *simultaneidade* e *segmentação*: duas condições que são difíceis de ser obtidas com um monólito.

Dimensionando uma aplicação

Vamos examinar a questão em detalhes.

O objetivo de qualquer aplicação de software é processar tarefas de alguma natureza. Independentemente do tipo de tarefa, podemos assumir que, de forma geral, queremos que nossa aplicação processe-as de forma eficiente.

Para processar tarefas de forma eficiente, nossa aplicação precisa ter algum tipo de *simultaneidade*. Isso significa que não podemos ter apenas um processo para fazer todo o trabalho, pois assim esse processo escolheria uma tarefa por vez, concluiria todas as etapas necessárias (ou falharia!) e então passaria para a próxima – isto não é nada eficiente! Para tornar nossa aplicação eficiente, podemos introduzir simultaneidade de modo que cada tarefa possa ser dividida em partes menores.

O que podemos fazer também para processar tarefas de forma eficiente é dividir e conquistar introduzindo *segmentação*, de modo

que cada tarefa não seja apenas dividida em partes pequenas, mas também possa ser processada em paralelo. Se tivermos várias tarefas, poderemos processá-las todas ao mesmo tempo enviando-as para um grupo de workers que possam processá-las em paralelo. Se precisarmos processar mais tarefas, poderemos facilmente dimensionar a aplicação de acordo com a demanda, acrescentando workers adicionais para processar as novas tarefas sem afetar a eficiência de nosso sistema.

Simultaneidade e segmentação são difíceis de suportar quando você tem uma aplicação grande que precisa ser implantada em todos os servidores e precisa processar qualquer tipo de tarefa. Se sua aplicação for minimamente complicada, a única maneira de dimensioná-la com o crescente número de recursos e crescente tráfego é aumentar o hardware no qual a aplicação é implantada.

Para ser realmente eficiente, a melhor maneira de dimensionar uma aplicação é dividi-la em muitas aplicações independentes pequenas para que cada uma execute um tipo de tarefa. É necessário acrescentar outra etapa ao processo geral? Fácil: basta criar uma nova aplicação que execute apenas aquela etapa! Está precisando lidar com mais tráfego? Simples: acrescente mais workers a cada aplicação!

Simultaneidade e segmentação são difíceis de suportar em uma aplicação monolítica, o que impede que a arquitetura de aplicação monolítica seja tão eficiente quanto necessário.

Observamos este padrão surgir em empresas como Amazon, Twitter, Netflix, eBay e Uber: empresas que executam aplicações em milhares, até mesmo em centenas de milhares, de servidores e cujas aplicações tornaram-se monólitos e passaram a enfrentar desafios de escalabilidade. Os desafios que elas enfrentaram foram superados abandonando-se a arquitetura de aplicação monolítica em favor de *microsserviços*.

O conceito básico de um microsserviço é simples: ele é uma pequena aplicação que executa uma única tarefa e o faz com eficiência. Um microsserviço é um pequeno componente que pode ser facilmente substituído, e é desenvolvido e implantado de forma independente. Um microsserviço não sobrevive sozinho, porém – nenhum microsserviço é uma ilha –, é parte de um sistema maior sendo executado e funcionando junto a outros microsserviços para realizar tarefas que normalmente seriam

tratadas por uma grande aplicação autônoma.

O objetivo da arquitetura de microsserviços é construir um conjunto de pequenas aplicações, cada uma responsável por executar uma função (ao contrário da forma tradicional de construir uma aplicação que faz tudo), e permitir que cada microsserviço seja autônomo, independente e autossuficiente. A principal diferença entre uma aplicação monolítica e microsserviços é esta: uma aplicação monolítica (Figura 1.4) contém todos os recursos e funções dentro de uma aplicação e um código-base, todos implantados ao mesmo tempo, com cada servidor hospedando uma cópia completa de toda a aplicação, enquanto um microsserviço (Figura 1.5) contém apenas uma única função ou um recurso e é executado em um *ecossistema de microsserviços* junto a outros microsserviços, cada um executando uma função e/ou um recurso.

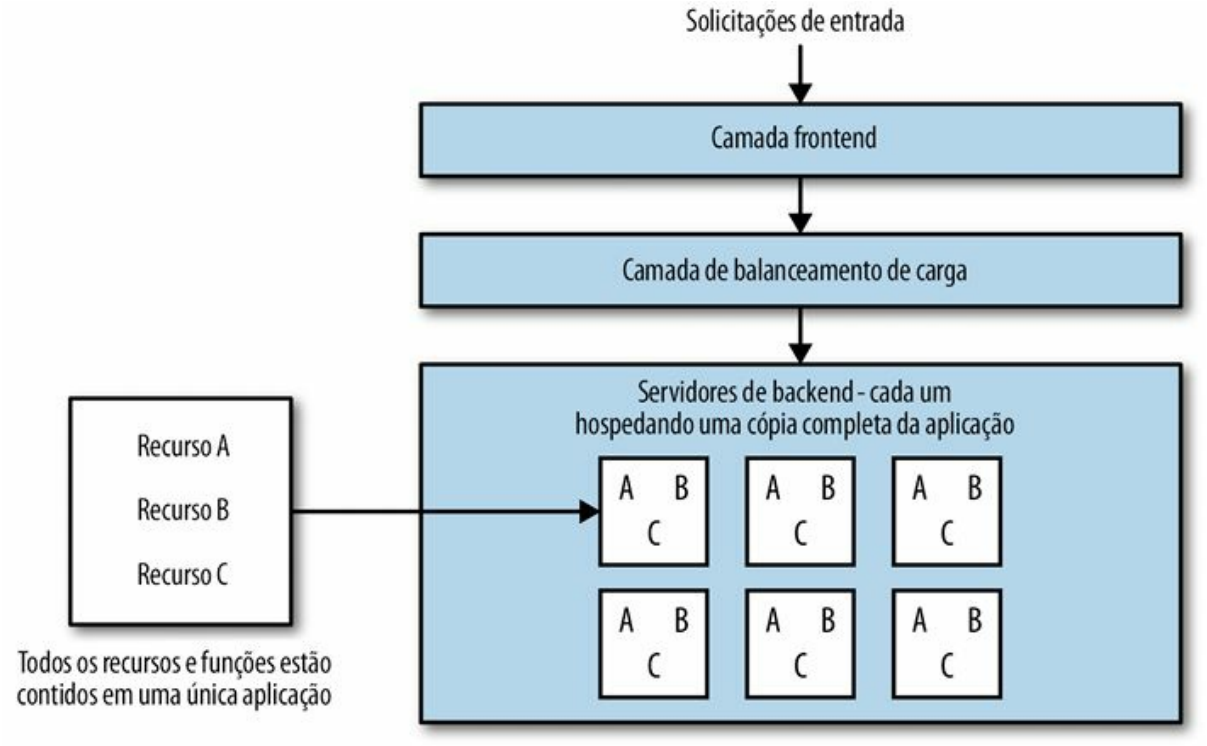


Figura 1.4. Monólito.

Existem muitas vantagens em adotar a arquitetura de microsserviços – incluindo (mas não se limitando a) uma defasagem técnica menor, melhor produtividade e velocidade do desenvolvedor, melhor eficiência de teste, maior escalabilidade e facilidade de implantação –, e empresas que adotam a arquitetura de microsserviços geralmente o fazem depois de terem construído uma aplicação e encontrado desafios organizacionais e de escalabilidade. Elas começam com uma aplicação monolítica e depois

dividem o monólito em microsserviços.

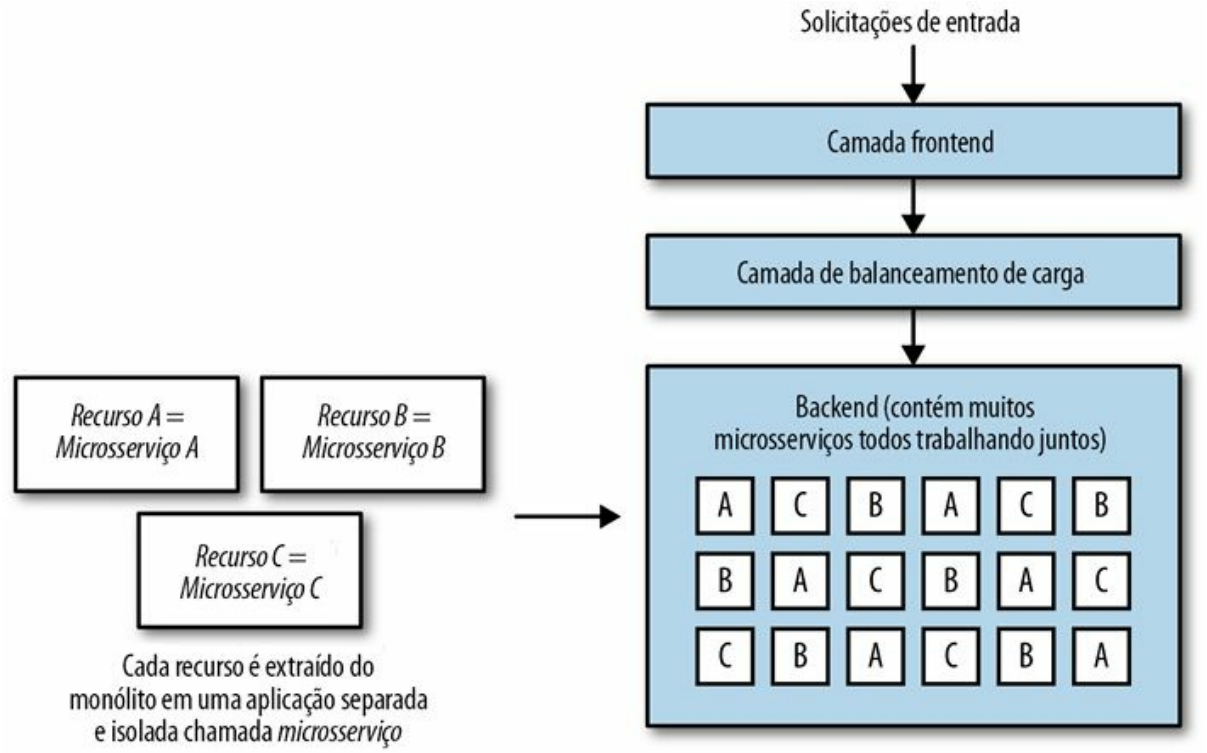


Figura 1.5. Microsserviços.

As dificuldades em dividir um monólito em microsserviços dependem totalmente da complexidade da aplicação monolítica. Uma aplicação monolítica com muitos recursos exigirá muito esforço de arquitetura e cuidadosa deliberação para ser dividida com sucesso em microsserviços, e uma complexidade adicional é introduzida pela necessidade de reorganizar e reestruturar as equipes. A decisão de migrar para microsserviços deve ser sempre um esforço que envolva toda a empresa.

Há várias etapas para se dividir um monólito. A primeira é identificar os componentes que devem ser escritos como serviços independentes. Talvez esta seja a etapa mais difícil de todo o processo, pois, embora existam muitas maneiras corretas de dividir o monólito em componentes de serviços, há muito mais maneiras erradas de fazê-lo. A regra geral para identificar componentes é localizar com precisão as principais funcionalidades gerais do monólito e então dividir essas funcionalidades em pequenos componentes independentes. Os microsserviços devem ser tão simples quanto possível, ou então a empresa correrá o risco de substituir um monólito por vários monólitos menores, que sofrerão dos mesmos problemas à medida que a empresa crescer.

Depois que as funções-chave tiverem sido identificadas e adequadamente transformadas em microsserviços independentes, a estrutura organizacional da empresa precisará ser redefinida de modo que cada microsserviço tenha sua própria equipe de engenharia. Há várias maneiras de fazer isso. O primeiro método de reorganização de uma empresa ao se adotar a arquitetura de microsserviços é dedicar uma equipe para cada microsserviço. O tamanho da equipe será determinado completamente pela complexidade e pela carga de trabalho do microsserviço, e essa equipe deve conter desenvolvedores e engenheiros de confiabilidade suficientes para que o desenvolvimento de recursos e a escala de prontidão do serviço possam ser gerenciados sem sobrecarregar a equipe. O segundo método é atribuir vários serviços a uma equipe e fazer com que essa equipe desenvolva os serviços em paralelo. Isso funciona melhor quando as equipes são organizadas em torno de produtos ou domínios de negócio específicos e são responsáveis por desenvolver qualquer serviço relacionado a esses produtos ou domínios. Se uma empresa escolhe o segundo método de reorganização, é preciso garantir que os desenvolvedores não sejam sobrecarregados e sofram de fadiga operacional ou aquela relacionada às tarefas ou interrupções.

Outra parte importante da adoção de microsserviços é a criação de um *ecossistema de microsserviços*. Geralmente (ou, pelo menos, assim se espera), uma empresa executando uma grande aplicação monolítica tem uma infraestrutura organizacional dedicada que é responsável por projetar, construir e manter a infraestrutura sobre a qual a aplicação é executada. Quando um monólito é dividido em microsserviços, as responsabilidades da infraestrutura organizacional em prover uma plataforma estável para que os microsserviços sejam desenvolvidos e mantidos crescem drasticamente. As equipes de infraestrutura devem fornecer às equipes de microsserviços uma infraestrutura estável que abstraia a maioria das complexidades das interações entre os microsserviços.

Uma vez concluídas estas três etapas – a divisão da aplicação em componentes, a reestruturação das equipes de engenharia para cada microsserviço e o desenvolvimento da infraestrutura organizacional dentro da empresa –, a migração pode começar. Algumas equipes escolhem extrair o código relevante para seu microsserviço diretamente do monólito para um serviço separado e replicar o tráfego do monólito até se

convencerem de que o microsserviço é capaz de oferecer a funcionalidade desejada por conta própria. Outras equipes escolhem construir o serviço a partir do zero e replicar o tráfego ou redirecioná-lo depois que o serviço tiver passado pelos testes adequados. A melhor abordagem para migração depende da funcionalidade do microsserviço, e já vi ambas as abordagens funcionarem igualmente bem na maioria dos casos, mas o essencial para uma migração bem-sucedida é uma execução e um planejamento minuciosos, cuidadosos, meticulosamente documentados, junto à percepção de que uma migração completa de um grande monólito pode levar muitos anos.

Com todo o trabalho envolvido na divisão de um monólito em microsserviços, pode parecer melhor começar com a arquitetura de microsserviços, ignorar todos os difíceis desafios de escalabilidade e evitar o drama da migração para microsserviços. Esta abordagem pode funcionar para algumas empresas, mas gostaria de oferecer alguns conselhos. Empresas pequenas geralmente não têm a infraestrutura necessária em funcionamento para manter microsserviços, mesmo em uma escala muito pequena: uma boa arquitetura de microsserviços requer infraestrutura estável e frequentemente muito complexa. Tal infraestrutura estável requer uma equipe grande e dedicada, cujo custo só pode ser mantido de algum modo por empresas que tenham atingido desafios de escalabilidade que justifiquem a migração para a arquitetura de microsserviços. Empresas pequenas simplesmente não têm capacidade operacional suficiente para manter um ecossistema de microsserviços. Além disso, é incrivelmente difícil identificar as áreas e os componentes essenciais que devem ser transformados em microsserviços quando uma empresa está nos estágios iniciais: as aplicações em empresas novas não têm muitos recursos e nem muitas áreas separadas de funcionalidade que possam ser divididas adequadamente em microsserviços.

Arquitetura de microsserviços

A *arquitetura de um microsserviço* (Figura 1.6) não é muito diferente da arquitetura-padrão de uma aplicação tratada na primeira seção deste capítulo (Figura 1.1). Todo microsserviço terá três componentes: um elemento frontend (lado do cliente), algum código de backend que faça o trabalho pesado e uma maneira de armazenar ou obter dados relevantes.

O elemento frontend (lado do cliente) de um microsserviço não é a típica aplicação de frontend, mas uma API (*interface de programação de aplicação*) com *endpoints* estáticos (pontos de acesso). APIs de microsserviço bem projetadas permitem que os microsserviços interajam de forma fácil e eficaz, enviando solicitações aos endpoint(s) de API relevantes. Por exemplo, um microsserviço responsável pelos dados do cliente pode ter um endpoint do tipo *get_customer_information* (obter informações do cliente) para o qual outros serviços podem enviar solicitações para obter informações sobre clientes, um endpoint do tipo *update_customer_information* (atualizar informações do cliente) para o qual outros serviços podem enviar solicitações para atualizar as informações de um cliente específico e um endpoint *delete_customer_information* (apagar informações do cliente) que os serviços podem usar para apagar as informações de um cliente.

Figura 1.6. Elementos da arquitetura de um microsserviço.

Esses endpoints são separados apenas na arquitetura e na teoria, mas não na prática, pois eles convivem lado a lado como parte do código de backend que processa todas as solicitações. Para nosso exemplo de um microsserviço que é responsável pelos dados do cliente, uma solicitação enviada para o endpoint *get_customer_information* dispararia uma tarefa que processa a solicitação de entrada, determina quaisquer filtros ou opções específicos aplicados à solicitação, obtém as informações de um database, formata as informações e as devolve para o cliente (microsserviço) que as solicitou.

A maioria dos microsserviços armazena algum tipo de dado, seja na memória (talvez usando um cache) ou em um database externo. Se os dados relevantes forem armazenados na memória, não será preciso fazer uma chamada de rede extra para um database externo e o microsserviço poderá facilmente retornar quaisquer dados relevantes para um cliente. Se os dados forem armazenados em um database externo, o microsserviço

precisará fazer outra solicitação para o database, aguardar uma resposta e então continuar a processar a tarefa.

Esta arquitetura é necessária para que os microsserviços funcionem bem em conjunto. O paradigma da arquitetura de microsserviços requer que um conjunto de microsserviços funcione junto para executar as ações que antes existiam na forma de uma grande aplicação, portanto há certos elementos dessa arquitetura que precisam ser padronizados em toda a organização para que um conjunto de microsserviços interaja de forma bem-sucedida e eficiente.

Os endpoints da API dos microsserviços devem ser padronizados em toda a organização. Isso não quer dizer que todos os microsserviços devam ter os mesmos endpoints específicos, mas que o tipo de endpoint deve ser o mesmo. Dois tipos muito comuns de endpoints de API para microsserviços são REST ou Apache Thrift, e já vi alguns microsserviços que têm ambos os tipos de endpoints (embora isso seja raro e torne o monitoramento complicado, por isso não o recomendo particularmente). A escolha do tipo de endpoint é um reflexo do funcionamento interno do próprio microsserviço e também ditará sua arquitetura: é difícil construir um microsserviço assíncrono que se comunique via HTTP usando endpoints do tipo REST, por exemplo, o que levaria à necessidade de adicionar aos serviços um endpoint baseado em troca de mensagens também.

Os microsserviços interagem entre si via RPCs (*remote procedure calls*), que são chamadas por meio de redes projetadas para se parecerem com e se comportarem exatamente como chamadas locais de procedimentos. Os protocolos usados dependem das escolhas de arquitetura e suporte organizacional, assim como dos endpoints usados. Um microsserviço com endpoints do tipo REST, por exemplo, provavelmente irá interagir com outros microsserviços via HTTP, enquanto um microsserviço com endpoints do tipo Thrift pode se comunicar com outros microsserviços via HTTP ou por meio de uma solução interna mais personalizada.

Evite atribuir versões aos microsserviços e endpoints

Um microsserviço não é uma biblioteca (ele não é carregado na memória no momento da compilação ou durante a execução), mas uma aplicação de software independente. Devido à natureza acelerada do desenvolvimento de um microsserviço, atribuir versões aos

> microsserviços pode facilmente se tornar um pesadelo organizacional, com os desenvolvedores de serviços para os clientes incorporando versões específicas (obsoletas e sem manutenção) de um microsserviço a seus próprios códigos. Os microsserviços devem ser tratados como coisas vivas e em constante mutação, e não como versões estáticas ou bibliotecas. Atribuir versões aos endpoints de uma API é outro antipadrão que deveria ser evitado pelas mesmas razões.

Qualquer tipo de endpoint e qualquer protocolo usado para comunicação com outros microsserviços terão benefícios e contrapartidas (“trade-offs”). As decisões de arquitetura não devem ser tomadas pelo desenvolvedor individual que está construindo um microsserviço, mas devem fazer parte do projeto de arquitetura do ecossistema de microsserviços como um todo (trataremos disso na próxima seção).

Escrever um microsserviço dá ao desenvolvedor bastante liberdade: além das escolhas organizacionais relativas aos endpoints de API e protocolos de comunicação, os desenvolvedores são livres para escrever a lógica interna de seus microsserviços da forma que desejarem. Eles podem usar qualquer linguagem de programação – o código pode ser escrito em Go, em Java, em Erlang, em Haskell – desde que se tratem com atenção os endpoints e os protocolos de comunicação. Desenvolver um microsserviço não é tão diferente de desenvolver uma aplicação autônoma. Existem algumas ressalvas, como veremos na seção final deste capítulo (“Desafios organizacionais”), pois a liberdade do desenvolvedor em relação à escolha da linguagem tem um custo elevado para a organização de engenharia de software.

Desta forma, um microsserviço pode ser tratado pelos outros como uma caixa-preta: você insere alguma informação enviando uma solicitação para um de seus endpoints e obtém uma saída. Se você obtém o que deseja e precisa de um microsserviço em um tempo razoável e sem erros absurdos, é sinal de que ele cumpriu o seu papel e que não é preciso entender nada além dos endpoints que precisam ser acessados nem verificar se o serviço está funcionando corretamente ou não.

Nossa discussão sobre as especificidades da arquitetura de microsserviços termina aqui – não porque tenhamos esgotado todo o assunto, mas porque cada um dos próximos capítulos deste livro é dedicado a levar os microsserviços até este estado ideal de caixa-preta.

Ecossistema de microsserviços

Microsserviços não vivem isolados. O ambiente no qual os microsserviços são construídos, executados e interagem é onde eles *vivem*. As complexidades do ambiente de microsserviços de grande escala são comparáveis às complexidades ecológicas de uma floresta tropical, um deserto ou um oceano, e considerar este ambiente como um ecossistema – um *ecossistema de microsserviços* – é benéfico quando se adota a arquitetura de microsserviços.

Em ecossistemas de microsserviços bem projetados e sustentáveis, os microsserviços são separados de toda a infraestrutura. Eles são separados do hardware, separados das redes, separados do pipeline de build e deployment, separados da descoberta de serviço (service discovery) e do balanceamento de carga. Isso tudo faz parte da infraestrutura do ecossistema de microsserviços, e construir, padronizar e manter esta infraestrutura de maneira estável, escalável, tolerante a falhas e confiável são fatores essenciais para uma operação bem-sucedida dos microsserviços.

A infraestrutura precisa sustentar o ecossistema de microsserviços. O objetivo de todo engenheiro e arquiteto de infraestrutura deve ser o de remover as questões operacionais de baixo nível do desenvolvimento de microsserviços e construir uma infraestrutura estável que possa crescer e sobre a qual os desenvolvedores possam facilmente construir e executar microsserviços. Desenvolver um microsserviço dentro de um ecossistema de microsserviços estável deve ser parecido com desenvolver uma pequena aplicação autônoma. Isso requer uma infraestrutura muito sofisticada e de alta qualidade.

O ecossistema de microsserviços pode ser dividido em quatro camadas (Figura 1.7), embora as fronteiras entre elas não sejam sempre claras: alguns elementos da infraestrutura tocarão todas as partes da pilha. As três camadas inferiores são as camadas de infraestrutura: na parte inferior da pilha temos a camada de hardware e sobre ela a camada de comunicação (que chega até a quarta camada), seguida da plataforma de aplicação. A quarta camada (na parte superior) é onde vivem todos os microsserviços individuais.

Figura 1.7. Modelo de quatro camadas do ecossistema de microsserviços.

Camada 1: Hardware

Na parte inferior do ecossistema de microsserviços temos a *camada de hardware*. Trata-se das máquinas propriamente ditas, os computadores reais e físicos nos quais todas as ferramentas internas e os microsserviços são executados. Estes servidores estão localizados em racks dentro de datacenters, são refrigerados por caros sistemas HVAC e alimentados por eletricidade. Muitos tipos diferentes de servidores podem estar presentes: alguns são otimizados para databases; outros, para processar tarefas que exigem muita CPU. Estes servidores podem ser de propriedade da própria empresa ou "alugados" dos chamados provedores de serviços de nuvem, como a Elastic Compute Cloud (AWS EC2) da Amazon Web Services, o Google Cloud Platform (GCP) ou o Microsoft Azure.

A escolha de um hardware específico é determinada pelos proprietários dos servidores. Se sua empresa tem datacenters próprios, a escolha do hardware é sua e você pode otimizar a escolha do servidor de acordo com suas necessidades específicas. Se os servidores estão na nuvem (que é o cenário mais comum), sua escolha é limitada pelo hardware que é oferecido pelo provedor. Escolher entre *servidores físicos* e um *provedor na nuvem* (ou provedores) não é uma decisão fácil, e é preciso levar em conta o custo, a disponibilidade, a confiabilidade e as despesas operacionais.

Gerenciar estes servidores é parte da camada de hardware. Cada servidor precisa ter um *sistema operacional* instalado, e o sistema operacional deve ser padronizado entre todos os servidores. Não há uma resposta certa sobre

qual sistema operacional um ecossistema de microsserviços deve usar: a resposta a esta questão depende totalmente das aplicações que você irá construir, das linguagens nas quais elas serão escritas e das bibliotecas e ferramentas que seus microsserviços exigem. A maioria dos ecossistemas de microsserviços roda em alguma variante do Linux, geralmente o CentOS, o Debian ou o Ubuntu, mas uma empresa .NET irá, obviamente, escolher um sistema diferente. Abstrações adicionais podem ser construídas e empilhadas acima do hardware: isolamento de recursos e abstração de recursos (como aquelas oferecidas por tecnologias como Docker e Apache Mesos) também pertencem a esta camada, assim como os databases (dedicados ou compartilhados).

Instalar um sistema operacional e *fornecer* o hardware é a primeira camada acima dos próprios servidores. Cada servidor deve ser fornecido e configurado, e depois que o sistema operacional for instalado, uma ferramenta de *gerenciamento de configuração* (como Ansible, Chef ou Puppet) deve ser usada para instalar todas as aplicações e fazer todas as configurações necessárias.

Os servidores precisam de um adequado *monitoramento em nível de servidor* (usando algo como Nagios) e *logging em nível de servidor* para que, caso algo aconteça (falha no disco, falha na rede ou se a utilização da CPU ultrapassar o limite), os problemas com os servidores possam ser facilmente diagnosticados, mitigados e resolvidos. O monitoramento no âmbito de servidor é tratado com mais detalhes no Capítulo 6, *Monitoramento*.

Resumo da camada 1: a camada de hardware

A camada de hardware (camada 1) do ecossistema de microsserviços contém:

- os servidores físicos (de propriedade da empresa ou alugados de provedores de serviços de nuvem)
- databases (dedicados e/ou compartilhados)
- o sistema operacional
- isolamento e abstração de recursos
- gerenciamento de configuração
- monitoramento em nível de servidor

• logging em nível de servidor

Camada 2: Comunicação

A segunda camada do ecossistema de microsserviços é a *camada de comunicação*. A camada de comunicação se espalha por todas as outras camadas do ecossistema (incluindo as camadas de plataforma de aplicação e microsserviços), pois é nela que toda a comunicação entre os serviços é tratada; as fronteiras entre a camada de comunicação e qualquer outra camada do ecossistema de microsserviços são pouco definidas. Embora as fronteiras não sejam claras, os elementos *são* claros: a segunda camada de um ecossistema de microsserviços sempre contém a rede, o DNS, os RPCs e os endpoints de API, a descoberta de serviço (service discovery), o registro de serviço (service registry) e o balanceamento de carga.

Discutir os elementos de rede e DNS da camada de comunicação está fora do escopo deste livro, portanto nesta seção nos concentraremos nos elementos RPCs, nos endpoints de API, na descoberta de serviço, no registro de serviço e no balanceamento de carga.

RPCs, endpoints e troca de mensagens

Microsserviços interagem uns com os outros através da rede usando RPCs (*remote procedure calls*) ou *troca de mensagens* para os *endpoints de API* de outros microsserviços (ou, como veremos no caso da troca de mensagens, para um broker de mensagem que roteará a mensagem adequadamente). A ideia básica é: usando um protocolo especificado, um microsserviço enviará pela rede alguns dados em um formato padronizado para outro serviço (talvez para o endpoint de API de outro microsserviço) ou para um broker (intermediário) de mensagem que garante que os dados sejam enviados para o endpoint da API de outro microsserviço.

Existem vários paradigmas de comunicação entre microsserviços. O primeiro é o mais comum: *HTTP+REST/THRIFT*. No HTTP+REST/THRIFT, os serviços se comunicam entre si pela rede usando HTTP (*Hypertext Transfer Protocol*) e enviando solicitações para e recebendo respostas de endpoints específicos do tipo REST (*representational state transfer*) (usando vários métodos, como GET, POST etc.) ou endpoints específicos do tipo *Apache Thrift* (ou ambos).

Geralmente os dados são formatados e enviados como *JSON* (ou *buffers de protocolo*) via HTTP.

HTTP+REST é a forma mais conveniente de comunicação entre microsserviços. Este método não apresenta surpresas, é fácil de configurar e é o mais estável e confiável – principalmente, pois é difícil implementá-lo incorretamente. A desvantagem de adotar este paradigma é que ele é, por necessidade, síncrono (bloqueante).

O segundo paradigma de comunicação é a *troca de mensagens (messaging)*. A troca de mensagens é assíncrona (não bloqueante), mas é um pouco mais complicada. A troca de mensagens funciona da seguinte maneira: um microsserviço envia dados (uma *mensagem*) pela rede (HTTP ou outro protocolo) para um *broker de mensagem*, que roteará a comunicação para outros microsserviços.

Existem vários tipos de troca de mensagem, sendo que os dois mais populares são *publicar–assinar* (pubsub) e *solicitação–resposta*. Nos modelos pubsub, clientes *assinam* um *tópico* e recebem uma mensagem sempre que um *editor publica* uma mensagem naquele tópico. Modelos solicitação-resposta são mais diretos, quando um cliente envia uma *solicitação* para um serviço (ou broker de mensagem), que *responde* com as informações solicitadas. Existem algumas tecnologias de troca de mensagens que são uma combinação única de ambos os modelos, como o Apache Kafka. Celery e Redis (ou Celery com RabbitMQ) podem ser usados para troca de mensagens (e processamento de tarefa) para microsserviços escritos em Python: o Celery processa as tarefas e/ou mensagens usando Redis ou RabbitMQ como broker.

A troca de mensagens apresenta várias desvantagens significativas que precisam ser mitigadas. A troca de mensagens pode ser tão escalável (se não, mais escalável) quanto as soluções HTTP+REST se ela for arquitetada para escalabilidade desde o início. Essencialmente, não é tão fácil alterar e atualizar a troca de mensagens, e sua natureza centralizada (embora pareça um benefício) pode fazer com que suas filas e os brokers se tornem pontos de falha de todo o ecossistema. A natureza assíncrona da troca de mensagens pode levar a condições de corrida e loops infinitos se não estiver preparada para lidar com o problema. Se um sistema de troca de mensagens for implementado com proteções contra esses problemas, ele pode tornar-se tão estável e eficiente quanto uma solução síncrona.

Descoberta de serviços, registro de serviços e balanceamento de carga

Em uma arquitetura monolítica, o tráfego só precisa ser enviado para uma aplicação e distribuído adequadamente para os servidores que estiverem hospedando a aplicação. Em uma arquitetura de microsserviços, o tráfego precisa ser roteado adequadamente para um grande número de diferentes aplicações e então distribuído corretamente para os servidores que estiverem hospedando cada microsserviço específico. Para que isso possa ser feito de forma eficiente e eficaz, a arquitetura de microsserviços requer que três tecnologias sejam implementadas na camada de comunicação: *descoberta de serviços*, *registro de serviços* e *balanceamento de carga*.

Em geral, quando um microsserviço A precisa fazer uma solicitação para outro microsserviço B, o microsserviço A precisa saber o endereço IP e a porta de uma instância específica, na qual o microsserviço B está hospedado. Mais especificamente, a camada de comunicação entre os microsserviços precisa conhecer os endereços IP e as portas desses microsserviços para que as solicitações entre eles possam ser roteadas adequadamente. Isso é possível graças à descoberta de serviço (como etcd, Consul, Hyperbahn ou ZooKeeper), que garante que as solicitações sejam roteadas exatamente para onde elas devem ser enviadas e que (muito importante) elas sejam roteadas apenas para instâncias saudáveis. A descoberta de serviço requer um registro de serviços, que é um database que registra todas as portas e os IPs de todos os microsserviços do ecossistema.

 Dimensionamento dinâmico e portas atribuídas

Na arquitetura de microsserviços, portas e IPs podem (e o fazem) mudar o tempo todo, especialmente à medida que os microsserviços são escalados e reimplantados (especialmente com uma camada de abstração de hardware como o Apache Mesos). Uma maneira de abordar a descoberta e o roteamento é atribuir portas estáticas (tanto no frontend quanto no backend) a cada microsserviço.

A menos que você tenha cada microsserviço hospedado em uma única instância (o que é altamente improvável), você precisará de *balanceamento de carga* ativo em várias partes da camada de comunicação em todo o ecossistema de microsserviços. O balanceamento de carga funciona, em

um nível muito alto, da seguinte maneira: se você tiver dez diferentes instâncias hospedando um microsserviço, o software (e/ou hardware) de balanceamento de carga garantirá que o tráfego seja distribuído (balanceado) por todas as instâncias. O balanceamento de carga será necessário em todo local do ecossistema para o qual uma solicitação for enviada para uma aplicação, o que significa que qualquer grande ecossistema de microsserviços conterá muitas camadas de balanceamento de carga. Os balanceadores de carga mais comuns para este propósito são o Elastic Load Balancer da Amazon Web Services, o Netflix Eureka, o HAProxy e o Nginx.

Resumo da camada 2: a camada de comunicação

A camada de comunicação (camada 2) do ecossistema de microsserviços contém:

- rede
- DNS
- Remote Procedure Calls (RPCs)
- endpoints
- troca de mensagens
- descoberta de serviço
- registro de serviço
- balanceamento de carga

Camada 3: a plataforma de aplicação

A *plataforma de aplicação* é a terceira camada do ecossistema de microsserviços e contém todas as ferramentas e os serviços internos que são independentes dos microsserviços. Esta camada é repleta de ferramentas e serviços centralizados e válidos para todo o ecossistema. Elas precisam ser construídas de modo que as equipes de desenvolvimento de microsserviços não precisem projetar, construir ou manter nada além de seus próprios microsserviços.

Uma boa plataforma de aplicação é aquela que contém *ferramentas internas de autosserviço* para desenvolvedores, um *processo de desenvolvimento* padronizado, um *sistema de build e release* centralizado e

automatizado, *testes automatizados*, uma *solução de deployment* padronizada e centralizada e *logging e monitoramento em nível de microsserviço* centralizados. Muitos dos detalhes desses elementos são tratados nos capítulos posteriores, mas trataremos de vários deles brevemente aqui para fornecer uma introdução aos conceitos básicos.

Ferramentas internas de desenvolvimento do tipo autosserviço

Muitos elementos podem ser categorizados como *ferramentas internas de desenvolvimento do tipo autosserviço*, e quais delas se encaixam nesta categoria depende não apenas das necessidades dos desenvolvedores, mas também do nível de abstração e sofisticação da infraestrutura e do ecossistema como um todo. A chave para determinar quais ferramentas precisam ser construídas é primeiro dividir as esferas de responsabilidade e então determinar quais tarefas os desenvolvedores devem ser capazes de realizar para projetar, construir e manter seus serviços.

Em uma empresa que adotou a arquitetura de microsserviços, as responsabilidades precisam ser cuidadosamente delegadas a diferentes equipes de engenharia. Uma maneira fácil de fazê-lo é criar uma suborganização de engenharia de software para cada camada do ecossistema de microsserviços, junto a outras equipes que conectem cada camada. Cada uma dessas organizações de engenharia de software, funcionando de forma semi-independente, será responsável por tudo dentro de sua camada: equipes TechOps serão responsáveis pela camada 1, equipes de infraestrutura serão responsáveis pela camada 2, equipes de plataforma de aplicação serão responsáveis pela camada 3 e equipes de microsserviço serão responsáveis pela camada 4 (claro que esta é uma visão muito simplificada, mas dá para ter uma ideia geral).

Nesse esquema organizacional, sempre que um engenheiro trabalhando em um dos níveis superiores precisar instalar, configurar ou utilizar algo de um dos níveis inferiores, deve haver uma ferramenta de autosserviço disponível para uso. Por exemplo, a equipe trabalhando em troca de mensagens para o ecossistema deve construir uma ferramenta de autosserviço de modo que, se um desenvolvedor em uma equipe de microsserviço precisar configurar uma troca de mensagens (messaging) para seu serviço, ela possa facilmente configurar a troca de mensagens sem ter de entender todas as complexidades do sistema de troca de mensagens.

Existem muitas razões para disponibilizar estas ferramentas de autosserviço centralizadas para cada camada. Em um ecossistema de microsserviços diverso, o engenheiro médio em qualquer equipe não terá nenhum (ou muito pouco) conhecimento sobre como os serviços e sistemas em outras equipes funcionam, e simplesmente não há como ele se tornar um expert em cada serviço e sistema enquanto trabalha em seu próprio – simplesmente isso não é viável. Cada desenvolvedor individual conhecerá quase nada, exceto aquilo que envolve seu próprio serviço, mas, juntos, todos os desenvolvedores trabalhando dentro do ecossistema conhecerão tudo coletivamente. Em vez de tentar mostrar a cada desenvolvedor as complexidades de cada ferramenta ou serviço dentro do ecossistema, é preciso construir interfaces de usuário sustentáveis e fáceis de usar para cada parte do ecossistema, e então ensinar e treinar como usá-las. Transforme tudo em uma caixa-preta e registre exatamente como ela funciona e como usá-la.

A segunda razão para construir essas ferramentas e construí-las bem é que, falando honestamente, não queremos que um desenvolvedor de outra equipe seja capaz de fazer alterações significativas em seu serviço ou sistema, especialmente alterações que possam causar uma interrupção. Isto é válido e obrigatório especialmente para os serviços e sistemas que pertencem às camadas inferiores (camadas 1, 2 e 3). Permitir que não especialistas façam alterações nestas camadas ou exigir (ou pior, esperar) que eles se tornem experts nessas áreas é uma receita de desastre. Um exemplo de uma área em que isso pode causar muitos problemas é o gerenciamento de configuração: permitir que desenvolvedores nas equipes de microsserviço façam alterações nas configurações do sistema sem terem a expertise para isso pode levar e levará a interrupções de produção em grande escala se uma alteração feita afetar algo além do próprio serviço.

Ciclo de desenvolvimento

Quando desenvolvedores alteram microsserviços ou criam novos, o desenvolvimento pode se tornar mais eficaz quando otimizamos e padronizamos o processo de desenvolvimento e o automatizamos o máximo possível. Os detalhes da padronização do processo de um desenvolvimento estável e confiável são tratados no Capítulo 4, *Escalabilidade e desempenho*, mas existem muitos elementos que precisam estar funcionando na terceira camada de um ecossistema de

microsserviços para que seja possível um desenvolvimento estável e confiável.

O primeiro requisito é um *sistema de controle de versão* centralizado, no qual todo o código possa ser armazenado, rastreado, ter uma versão atribuída a ele e pesquisado. Geralmente isso é feito por meio de algo como o GitHub ou um repositório git ou svn auto-hospedado e vinculado a algum tipo de ferramenta de colaboração, como Phabrictor; essas ferramentas facilitam a manutenção e a revisão do código.

O segundo requisito é um *ambiente de desenvolvimento* estável e eficiente. Ambientes de desenvolvimento são famosos por sua dificuldade de implementação em ecossistema de microsserviços devido às complicadas dependências que cada microsserviço tem em relação aos outros serviços, mas eles são absolutamente essenciais. Algumas organizações de engenharia de software preferem quando todo o desenvolvimento é feito localmente (no laptop de um desenvolvedor), mas isso pode levar a implantações ruins, pois o desenvolvedor não tem uma imagem precisa de como as alterações de seu código irão funcionar no mundo da produção. A maneira mais estável e confiável de projetar um ambiente de desenvolvimento é criar um espelho do ambiente de produção (um que não seja uma réplica (“staging”), nem pré-release e nem produção) contendo todas as intrincadas cadeias de dependência.

Teste, build, empacotamento e liberação

As *etapas de teste, build, empacotamento (package) e liberação (release)* entre o desenvolvimento e a implantação devem ser padronizadas e centralizadas o máximo possível. Depois do ciclo de desenvolvimento, quando qualquer alteração de código tiver sido submetida ao repositório, todos os testes necessários devem ser feitos e novas versões devem ser automaticamente construídas e empacotadas. Uma ferramenta de *integração contínua* existe exatamente para isso, e as soluções existentes (como Jenkins) são muito avançadas e fáceis de configurar. Estas ferramentas facilitam a automação de todo o processo, deixando pouquíssimo espaço para erro humano.

Pipeline de deployment

A *pipeline de deployment* é o processo pelo qual um novo código é

instalado nos servidores de produção depois do ciclo de desenvolvimento e das etapas de teste, build, empacotamento e liberação. A implantação pode tornar-se rapidamente muito complicada em um ecossistema de microsserviços, onde centenas de implantações por dia não são incomuns. Construir ferramentas de implantação e padronizar as práticas de implantação para todas as equipes de desenvolvimento é frequentemente necessário. Os princípios de construção de pipelines de implantação estáveis e confiáveis (prontos para produção) são tratados em detalhes no Capítulo 3, *Estabilidade e confiabilidade*.

Logging e monitoramento

Todos os microsserviços devem manter um *logging em nível de microsserviço* de todas as solicitações feitas ao microsserviço (incluindo todas as informações relevantes e importantes) e suas respostas. Devido à natureza acelerada do desenvolvimento de microsserviços, geralmente é impossível reproduzir os bugs no código, pois é impossível reconstruir o estado do sistema no momento da falha. Um bom logging em nível de microsserviço oferece aos desenvolvedores as informações necessárias para que eles entendam completamente o estado de seu serviço em certo momento do passado ou presente. O *monitoramento em nível de microsserviço* de todas as *principais métricas* dos microsserviços é essencial por razões semelhantes: um monitoramento preciso e em tempo real permite que os desenvolvedores sempre conheçam a saúde e o status de seu serviço. O logging e o monitoramento em nível de microsserviço são tratados em mais detalhes no Capítulo 6, *Monitoramento*.

Resumo da camada 3: a camada da plataforma de aplicação

A camada da plataforma de aplicação (camada 3) do ecossistema de microsserviços contém:

- ferramentas internas de desenvolvimento do tipo autosserviço
- ambiente de desenvolvimento
- ferramentas de teste, empacotamento, build e liberação
- pipeline de deployment
- logging em nível de microsserviço
- monitoramento em nível de microsserviço

Camada 4: Microsserviços

No topo do ecossistema de microsserviços encontra-se a *camada de microsserviços* (*camada 4*). Essa camada é onde vivem os microsserviços – e tudo que for específico a eles —, completamente separados das camadas inferiores de infraestrutura. Aqui eles são separados do hardware, da implantação, da descoberta de serviço, do balanceamento de carga e da comunicação. As únicas coisas que não são separadas da camada de microsserviços são as configurações específicas de cada serviço para uso das ferramentas.

É uma prática comum na engenharia de software centralizar todas as configurações da aplicação para que as configurações de uma ferramenta ou um conjunto de ferramentas específicas (como o gerenciamento da configuração, o isolamento de recursos ou as ferramentas de implantação) fiquem todas armazenadas com a própria ferramenta. Por exemplo, as configurações de implantação personalizada para aplicações são em geral armazenadas com o código da ferramenta de implantação e não com o código da aplicação. Esta prática funciona bem para a arquitetura monolítica e até mesmo para pequenos ecossistemas de microsserviços, mas em um ecossistema de microsserviços muito grande contendo centenas de microsserviços e dúzias de ferramentas internas (cada uma com suas próprias configurações personalizadas) esta prática torna-se confusa: os desenvolvedores das equipes de microsserviços precisam fazer alterações nos códigos das ferramentas nas camadas inferiores e, muitas vezes, esquecem onde certas configurações estão (ou que elas existem). Para minimizar este problema, todas as configurações específicas do microsserviço podem ficar no repositório do microsserviço e devem ser acessadas pelas ferramentas e pelos sistemas das camadas inferiores.

Resumo da camada 4: a camada dos microsserviços

A camada dos microsserviços (camada 4) do ecossistema de microsserviços contém:

- os microsserviços
- todas as configurações específicas dos microsserviços

Desafios organizacionais

A adoção da arquitetura de microsserviços resolve os desafios mais prementes apresentados pela arquitetura de aplicação monolítica. Os microsserviços não são assolados pelos mesmos desafios de escalabilidade, pela falta de eficiência ou pelas dificuldades de adotar novas tecnologias: eles são otimizados para escalabilidade, eficiência e velocidade de desenvolvimento. Em uma indústria na qual novas tecnologias rapidamente ganham impulso de mercado, o puro custo organizacional de manter e tentar melhorar uma aplicação monolítica pesada simplesmente não é prático. Com isto em mente, é difícil imaginar por que alguém relutaria em dividir um monólito em microsserviços, por que alguém ficaria receoso em construir um ecossistema de microsserviços a partir do zero.

Os microsserviços parecem uma solução mágica (e um tanto óbvia), mas sabemos que não é bem assim. Em *The Mythical Man-Month (O Mítico Homem-mês)*, Frederick Brooks explicou por que não existem balas de prata na engenharia de software, uma ideia que ele resumiu da seguinte forma: “Não há um único desenvolvimento, em tecnologia ou técnicas de gerenciamento, que sozinho prometa até mesmo uma melhoria de uma ordem de grandeza dentro de uma década em termos de produtividade, confiabilidade e simplicidade.”

Quando somos apresentados a uma tecnologia que promete oferecer melhorias drásticas, precisamos procurar as contrapartidas. Os microsserviços prometem maior escalabilidade e maior eficiência, mas sabemos que isso terá um custo para alguma parte do sistema geral.

Existem quatro contrapartidas especialmente significativas que surgem com a arquitetura de microsserviços. A primeira é a mudança da estrutura organizacional que tende a um isolamento e uma comunicação deficiente entre as equipes – uma consequência do reverso da *Lei de Conway*. A segunda é o aumento dramático da *dispersão técnica*, que é incrivelmente custosa não apenas para toda a organização, mas que também apresenta custos significativos para cada engenheiro. A terceira contrapartida é a maior *capacidade de falha do sistema*. A quarta é a *competição por recursos de engenharia e infraestrutura*.

Lei de Conway Reversa

A ideia por trás da *Lei de Conway* (em homenagem ao programador Melvin Conway em 1968) é: a arquitetura de um sistema será determinada pelas estruturas de comunicação e organizacionais da empresa. O reverso da Lei de Conway (que chamaremos de *Lei de Conway Reversa*) também é válido e é especialmente relevante para o ecossistema de microsserviços: a estrutura organizacional de uma empresa é determinada pela arquitetura de seu produto. Mais de quarenta anos após a Lei de Conway ter sido introduzida pela primeira vez, tanto ela quanto a lei reversa ainda parecem válidas. A estrutura organizacional da Microsoft, se ela fosse esboçada como a arquitetura de um sistema, se parece muito com a arquitetura de seus produtos – o mesmo vale para o Google, a Amazon e todas as demais grandes empresas de tecnologia. Empresas que adotam a arquitetura de microsserviços nunca serão uma exceção a esta regra.

A arquitetura de microsserviços consiste em um grande número de pequenos microsserviços independentes e isolados. A Lei de Conway Reversa exige que a estrutura organizacional de toda empresa que usa arquitetura de microsserviços seja composta de um grande número de equipes muito pequenas, isoladas e independentes. A estrutura de equipes resultante inevitavelmente leva aos fenômenos de segregação e dispersão, problemas que pioram sempre que o ecossistema de microsserviços se torna mais sofisticado, mais complexo, mais simultâneo e mais eficiente.

A Lei de Conway Reversa também significa que os desenvolvedores serão, em muitos aspectos, como microsserviços: eles serão capazes de fazer uma tarefa e (espera-se) fazer essa tarefa muito bem, mas eles estarão isolados (em termos de responsabilidade, conhecimento do domínio e experiência) do resto do ecossistema. Quando considerados juntos, todos os desenvolvedores trabalhando *coletivamente* dentro de um ecossistema de microsserviços saberão tudo que há para saber sobre ele, mas individualmente eles serão extremamente especializados, conhecendo apenas as partes do ecossistema pelas quais eles são responsáveis.

Isso traz um inevitável problema organizacional: embora os microsserviços devam ser desenvolvidos em isolamento (resultando em equipes isoladas e segregadas), eles não vivem em isolamento e precisam interagir entre si de forma transparente para que o produto como um todo funcione. Isso requer que estas equipes isoladas e funcionando

independentemente trabalhem bem próximas e com frequência – algo que é difícil de conseguir, dado que os objetivos e projetos da maioria das equipes (codificados em seus objetivos e resultados-chave da equipe, ou OKRs da sigla em inglês) são específicos para um microsserviço particular no qual eles estejam trabalhando.

Existe também um grande déficit de comunicação entre as equipes de microsserviços e as equipes de infraestrutura que precisa ser superado. As equipes de plataforma de aplicação, por exemplo, precisam construir serviços e ferramentas de plataforma que todas as equipes de microsserviços usarão, mas conhecer os requisitos e as necessidades de centenas de equipes de microsserviços antes de construir um pequeno projeto pode levar meses (até mesmo anos). Fazer com que os desenvolvedores e as equipes de infraestrutura trabalhem juntos não é uma tarefa fácil.

Há um problema relacionado, resultante da Lei de Conway Reversa, um problema que é raramente encontrado em empresas com arquitetura monolítica: a dificuldade de dirigir uma organização operacional. Com um monólito, uma organização operacional pode facilmente ser montada e estar de prontidão para a aplicação, mas isso é muito difícil de conseguir com uma arquitetura de microsserviços, pois exigiria que todo microsserviço tivesse uma equipe de desenvolvimento *e* uma equipe operacional. Consequentemente, as equipes de desenvolvimento de microsserviços precisam ser responsáveis pelos deveres e tarefas operacionais associados aos seus microsserviços. Não existe uma organização operacional separada para assumir os serviços de prontidão e nem uma organização operacional responsável pelo monitoramento: os desenvolvedores precisarão estar de prontidão para seus serviços.

Dispersão técnica

A segunda contrapartida, *dispersão técnica (technical sprawl)*, está relacionada à primeira. Se por um lado a Lei de Conway e seu reverso preveem dispersão e segregação organizacionais para microsserviços, um segundo tipo de dispersão (relacionado a tecnologias, ferramentas e itens similares) também é inevitável em uma arquitetura de microsserviços. A dispersão técnica pode se manifestar de muitas formas diferentes. Cobriremos algumas das mais comuns aqui.

É fácil ver por que a arquitetura de microsserviços resulta em dispersão técnica se considerarmos um grande ecossistema de microsserviços, um que contenha mil microsserviços. Suponha que cada microsserviço tenha uma equipe de desenvolvimento com seis profissionais, e cada desenvolvedor use seu conjunto de ferramentas favoritas, bibliotecas favoritas e programe na linguagem de sua preferência. Cada uma dessas equipes de desenvolvimento tem sua maneira própria de implantação, sua métrica específica para monitorar e gerar alertas, suas próprias bibliotecas externas e dependências internas, seus scripts personalizados para serem executados nas máquinas de produção, e assim por diante.

Se você tiver mil equipes assim, significa que dentro de um sistema existem mil maneiras de realizar determinada tarefa. Haverá mil formas de implantação, mil bibliotecas a serem mantidas, mil formas diferentes de gerar alertas, de monitorar e testar a aplicação e de lidar com as interrupções. A única maneira de reduzir a dispersão técnica é por meio da padronização em todos os níveis do ecossistema de microsserviços.

Existe outro tipo de dispersão técnica associada à linguagem escolhida. Os microsserviços carregam a fama de prometer maior liberdade do programador, liberdade para escolher as linguagens e bibliotecas que desejar. Isso é possível em princípio e pode ser verdadeiro na prática, mas, à medida que um ecossistema de microsserviços cresce, esta liberdade geralmente torna-se impraticável, custosa e perigosa. Para entender como isso pode se tornar um problema, considere o seguinte cenário. Suponha que tenhamos um ecossistema de microsserviços contendo duzentos serviços e imagine que alguns desses microsserviços sejam escritos em Python, outros em JavaScript, alguns em Haskell, alguns em Go, e outros ainda em Ruby, Java e C++. Para cada ferramenta interna, cada sistema e serviço dentro de cada camada do ecossistema, as bibliotecas terão de ser escritas para cada uma dessas linguagens.

Pense um pouco na enorme quantidade de manutenção e desenvolvimento que precisarão ser feitos para que cada linguagem receba o suporte necessário; é uma quantidade extraordinária, e poucas organizações de engenharia de software poderiam dedicar os recursos de engenharia necessários para que isso acontecesse. É mais realista escolher um pequeno número de linguagens suportadas e garantir que todas as bibliotecas e ferramentas sejam compatíveis com essas linguagens em vez

de tentar suportar um grande número de linguagens.

O último tipo de dispersão técnica tratado aqui é o déficit técnico, que geralmente se refere a trabalho que precisa ser feito, pois algo foi implementado rapidamente, mas não da melhor forma ou da forma ideal. Uma vez que equipes de desenvolvimento de microsserviços podem criar novos recursos rapidamente, o déficit técnico geralmente cresce silenciosamente em segundo plano. Quando acontecem interrupções, quando algo quebra, qualquer trabalho resultante de uma análise do incidente raramente será a melhor solução geral: no que diz respeito às equipes de desenvolvimento de microsserviços, qualquer solução que conserte o problema rapidamente é boa o suficiente, e soluções melhores são adiadas.

Outras maneiras de falhar

Os microsserviços são sistemas grandes, complexos e distribuídos com muitas pequenas partes independentes que estão mudando constantemente. A realidade de trabalhar com sistemas complexos deste tipo é que componentes individuais falharão e falharão com frequência, e falharão de uma maneira que ninguém pode prever. Este é o terceiro tipo de contrapartida: a arquitetura de microsserviços introduz novas maneiras de seu sistema falhar.

Existem maneiras de se preparar para falhas, para mitigar falhas quando elas ocorrem, e testar os limites e as fronteiras entre os componentes individuais e o ecossistema geral, que serão tratados no Capítulo 5, *Tolerância a falhas e prontidão para catástrofes*. Entretanto é importante entender que não importa quantos testes de resiliência você executar, não importa quantos cenários de falhas e catástrofes você cobrir, não é possível escapar do fato de que o sistema *irá* falhar. Existe um limite naquilo que pode ser feito para se preparar para quando incidentes acontecerem.

Competição por recursos

Assim como outros ecossistemas no mundo natural, a competição por recursos é feroz no ecossistema de microsserviços. Cada organização de engenharia de software contém recursos finitos: ela tem recursos finitos de engenharia (equipes, desenvolvedores) e recursos finitos de hardware e infraestrutura (máquinas físicas, hardware na nuvem, armazenamento em

database etc.) e cada recurso custa à empresa muito dinheiro.

Quando seu ecossistema de microsserviços tem um grande número de microsserviços e uma grande e sofisticada plataforma de aplicação, a competição entre equipes pelos recursos de hardware e infraestrutura é inevitável: cada serviço e cada ferramenta serão apresentados como igualmente importantes; suas necessidades de crescimento serão vistas como algo da mais alta prioridade.

Da mesma forma, quando as equipes da plataforma de aplicação solicitarem as especificações e as necessidades das equipes de microsserviços para que elas possam projetar seus sistemas e suas ferramentas adequadamente, cada equipe de desenvolvimento de microsserviços argumentará que suas necessidades são as mais importantes e que ficarão desapontados (e potencialmente muito frustrados) se elas não forem incluídas. Este tipo de competição por recursos de engenharia pode causar ressentimento entre as equipes.

O último tipo de competição por recursos é talvez o mais óbvio: a competição entre gerentes, entre equipes e entre diferentes departamentos/organizações de engenharia de software pela quantidade de contratações de profissionais. Mesmo com o crescimento do número de formados em ciências da computação e o aumento dos centros de treinamento de desenvolvedores, é difícil encontrar desenvolvedores realmente bons, e eles representam um dos recursos mais escassos e insubstituíveis. Quando existirem centenas ou milhares de equipes que poderiam ter um engenheiro extra ou dois, cada equipe irá insistir em afirmar que precisa de um engenheiro extra mais do que qualquer outra equipe.

Não há como evitar a competição por recursos, embora existam maneiras de atenuar um pouco a competição. A forma mais eficaz parece ser organizar ou categorizar as equipes em termos de sua importância e criticidade para o negócio como um todo, e então fornecer às equipes acesso aos recursos com base em sua prioridade ou importância. Há desvantagens, pois isso tende a resultar em equipes de ferramentas de desenvolvimento com poucos profissionais e projetos cuja importância esteja em definir o futuro (como a adoção de novas tecnologias de infraestrutura) sendo abandonados.

CAPÍTULO 2

Disponibilidade de produção

Se por um lado a adoção da arquitetura de microsserviços proporciona uma considerável liberdade para os desenvolvedores, garantir a disponibilidade de todo o ecossistema de microsserviços requer que os microsserviços individuais atinjam altos padrões de arquitetura, operacionais e organizacionais. Este capítulo trata dos desafios da padronização de microsserviços, introduz a disponibilidade como objetivo da padronização, apresenta os oito padrões de disponibilidade de produção e propõe estratégias para implementar a padronização da disponibilidade de produção em uma organização de engenharia de software.

Desafios da padronização de microsserviços

A arquitetura de uma aplicação monolítica geralmente é determinada no início de seu ciclo de vida. Para muitas aplicações, a arquitetura é determinada no momento em que uma empresa inicia suas atividades. À medida que o negócio cresce e a aplicação precisa ser redimensionada, os desenvolvedores que estão acrescentando novos recursos geralmente se encontram limitados e condicionados pelas escolhas feitas quando a aplicação foi concebida. Eles são limitados pela escolha da linguagem, pelas bibliotecas que podem usar, pelas ferramentas de desenvolvimento com as quais podem trabalhar e pela necessidade de um extensivo teste de regressão para garantir que todo novo recurso adicionado não irá atrapalhar ou comprometer a integridade da aplicação. Todo retrabalho que ocorrer na aplicação monolítica autônoma continua sendo essencialmente limitado pelas decisões iniciais de arquitetura: as condições iniciais determinam exclusivamente o futuro da aplicação.

A adoção da arquitetura de microsserviços proporciona uma considerável liberdade para os desenvolvedores. Eles não estarão mais amarrados às decisões de arquitetura do passado, mas poderão projetar seus serviços da

forma que desejarem e terão total liberdade para escolher a linguagem, o database, as ferramentas de desenvolvimento, e assim por diante. A mensagem que os desenvolvedores escutam quando adotam a arquitetura de microsserviços é geralmente a seguinte: construa uma aplicação que realize uma única tarefa e a faça *extraordinariamente bem*; faça o que for preciso, da forma que desejar – certifique-se apenas de que a aplicação funcione.

Embora esta idealização romântica do desenvolvimento de microsserviços seja verdadeira a princípio, nem todos os microsserviços são criados iguais – e nem deveriam. Cada microsserviço é parte de um ecossistema de microsserviços, e cadeias complexas de dependências são inevitáveis. Quando você tem cem, mil ou mesmo 10 mil microsserviços, cada um deles desempenha um pequeno papel em um sistema muito grande. Os serviços precisam interagir entre si de forma transparente e – o que é mais importante – nenhum serviço ou conjunto de serviços deve comprometer a integridade do sistema ou o produto do qual fazem parte. Se quisermos que o sistema ou o produto como um todo seja bom, é preciso cumprir certos padrões, e consequentemente cada uma de suas partes deve também cumprir os mesmos padrões.

É relativamente simples determinar os padrões e fornecer os requisitos a uma equipe de microsserviços se nos concentrarmos nas necessidades daquela equipe específica e no papel que seu serviço deve desempenhar. Podemos dizer “seu microserviço precisa fazer *x*, *y* e *z*, e para fazer *x*, *y* e *z* bem, você deve garantir o cumprimento deste conjunto *S* de requisitos”, fornecendo a cada equipe um conjunto de requisitos que seja relevante para seu serviço e apenas para seu serviço. Infelizmente esta abordagem não é escalável e não leva em conta o importante fato de que um microsserviço é apenas uma parte muito pequena de um quebra-cabeça absurdamente grande e distribuído. Precisamos definir padrões e requisitos para nossos microsserviços, e eles precisam ser genéricos o suficiente para ser aplicados a todos os microsserviços, porém específicos o suficiente também para ser quantificáveis e produzir resultados mensuráveis. É aí que entra o conceito de *disponibilidade de produção*.

Disponibilidade: o objetivo da padronização

Dentro de ecossistemas de microsserviços, acordos de nível de serviço

(SLAs ou service-level agreements) referentes à disponibilidade de um serviço são o método mais usado de avaliar o sucesso de um serviço: se um serviço tem alta disponibilidade (isto é, tem um downtime, ou tempo de indisponibilidade, muito pequeno), então podemos dizer com razoável confiança (e algumas ressalvas) que o serviço está cumprindo seu objetivo.

Calcular e medir a disponibilidade é fácil. É preciso calcular apenas três quantidades mensuráveis: *uptime* (a quantidade de tempo durante a qual o microsserviço funcionou corretamente), *downtime* (a quantidade de tempo durante a qual o microsserviço *não* funcionou corretamente) e o tempo total durante o qual um serviço esteve operacional (a soma do uptime com o downtime). A disponibilidade então é o uptime dividido pelo tempo total em que um serviço esteve operacional (uptime + downtime).

Por mais útil que seja, a disponibilidade não é, por si só, um princípio de padronização de microsserviço, mas seu objetivo. Ela não pode ser um princípio de padronização, pois não oferece uma orientação sobre como projetar, construir ou executar o microsserviço: pedir aos desenvolvedores para tornarem seu microsserviço mais disponível sem dizer como (ou *por que*) fazê-lo é inútil. A disponibilidade sozinha não fornece passos concretos e aplicáveis, mas, como veremos nas seções seguintes, há medidas concretas e aplicáveis que podem ser tomadas no sentido de atingir o objetivo de construir um microsserviço disponível.

Calculando a disponibilidade

A disponibilidade é medida na chamada notação dos noves, que corresponde à porcentagem de tempo em que um serviço está disponível. Por exemplo, um serviço que está disponível 99% do tempo é chamado de “disponibilidade de dois noves”.

Esta notação é útil, pois fornece uma quantidade específica de downtime que um serviço pode ter. Se seu serviço precisa ter uma disponibilidade de quatro noves, então ele pode ficar parado 52,56 minutos por ano, o que significa 4,38 minutos por mês ou 1,01 minuto por semana e 8,66 segundos por dia.

Aqui estão os cálculos de disponibilidade e downtime para uma disponibilidade entre 99% e 99,999%:

Disponibilidade 99%: (dois noves)

- 3,65 dias/ano (de downtime permitido)
- 7,20 horas/mês
- 1,68 horas/semana
- 14,4 minutos/dia

Disponibilidade 99,9% (três noves):

- 8,76 horas/ano
- 43,8 minutos/mês
- 10,1 minutos/semana
- 1,44 minuto/dia

Disponibilidade 99,99% (quatro noves):

- 52,56 minutos/ano
- 4,38 minutos/mês
- 1,01 minuto/semana
- 8,66 segundos/dia

Disponibilidade 99,999% (cinco noves):

- 5,26 minutos/ano
- 25,9 segundos/mês
- 6,05 segundos/semana
- 864,3 milissegundos/dia

Padrões de disponibilidade de produção

A ideia básica por trás da disponibilidade de produção é: uma aplicação ou um serviço pronto para produção é aquele que podemos confiar que atenderá o tráfego de produção. Quando nos referimos a uma aplicação ou um microsserviço como “pronto para produção”, nós depositamos uma grande confiança nele: nós acreditamos que ele irá se comportar de modo razoável, terá um desempenho confiável, executará sua tarefa e o fará com qualidade e pouco tempo de downtime. A disponibilidade de produção é essencial para a padronização do microsserviço e essencial para obtermos disponibilidade em todo o ecossistema de microsserviços.

Entretanto a ideia da disponibilidade de produção como definida anteriormente não é útil o suficiente por si só para servir como a definição ampla que necessitamos, e sem outras explicações o conceito não é muito

útil. Precisamos saber exatamente quais requisitos todo serviço deve cumprir para ser considerado pronto para produção e capaz de atender ao tráfego de produção de modo adequado e confiável – uma confiança que não pode ser dada de graça, mas que precisa ser conquistada. Os próprios requisitos precisam ser princípios verdadeiros para todo microsserviço, toda aplicação e todo sistema distribuído: padronização sem princípios não tem sentido.

Existe um conjunto de oito princípios que, quando adotados em conjunto, atendem a estes critérios. Cada um desses princípios é quantificável, resulta em um conjunto de requisitos acionáveis e produz resultados mensuráveis. Eles são *estabilidade*, *confiabilidade*, *escalabilidade*, *tolerância a falhas*, *prontidão para catástrofes*, *desempenho*, *monitoramento* e *documentação*. A força motora por trás de cada um desses princípios é que, juntos, eles contribuem para e impulsionam a *disponibilidade* de um microsserviço.

A disponibilidade é, em certos aspectos, uma propriedade resultante de um microsserviço pronto para produção. Ela é a decorrência de se construir um microsserviço escalável, confiável, tolerante a falhas, de alto desempenho, monitorado, documentado e preparado para catástrofes. Cada um desses princípios individualmente não é suficiente para garantir a disponibilidade, mas juntos, sim: construir um microsserviço com esses princípios como requisitos operacionais e de arquitetura garante um sistema altamente disponível ao qual é possível confiar o tráfego de produção.

Estabilidade

Com a introdução da arquitetura de microsserviços, os desenvolvedores têm liberdade para desenvolver e implantar serviços a uma velocidade muito alta. Novos recursos podem ser acrescentados e implantados diariamente, erros podem ser rapidamente corrigidos, tecnologias antigas podem ser trocadas pelas mais recentes, microsserviços desatualizados podem ser reescritos e as versões antigas podem ser descontinuadas e desativadas. Esta maior velocidade resulta em maior instabilidade, e em ecossistemas de microsserviços a maioria das interrupções pode geralmente ser rastreada numa implantação ruim contendo código defeituoso ou outros erros sérios. Para garantir a disponibilidade

precisamos estar atentos a esta instabilidade decorrente de maior velocidade de desenvolvimento e da constante evolução do ecossistema de microsserviços.

A estabilidade permite obter disponibilidade fornecendo maneiras de lidar de forma responsável com as alterações feitas nos microsserviços. Um microsserviço estável é aquele cujo desenvolvimento, implantação, acréscimo de novas tecnologias e a desativação e descontinuidade não resultam em instabilidade dentro de um ecossistema maior de microsserviços. Podemos determinar os requisitos de estabilidade para cada microsserviço visando a atenuar os efeitos colaterais negativos que podem acompanhar cada mudança.

Para atenuar quaisquer problemas decorrentes do ciclo de desenvolvimento, é possível usar procedimentos de desenvolvimento estável. Para neutralizar qualquer instabilidade introduzida pela implantação, podemos garantir que nossos microsserviços sejam implantados cuidadosamente com adequados lançamentos (roll outs) do tipo staging (replicação do ambiente de produção), pré-release (um pequeno grupo de 2% a 5% de servidores de produção) e produção. Para evitar que a introdução de novas tecnologias e a descontinuação e a desativação de antigos microsserviços comprometam a disponibilidade de outros serviços, podemos aplicar procedimentos estáveis de introdução e descontinuação.

Requisitos de estabilidade

Os requisitos para construir um microsserviço estável são:

- um ciclo de desenvolvimento estável
- um processo de implantação estável
- procedimentos estáveis de introdução e descontinuação

Os detalhes dos requisitos de estabilidade são abordados no *Capítulo 3 – Estabilidade e confiabilidade.*

Confiabilidade

A estabilidade sozinha não é suficiente para garantir a disponibilidade de um microsserviço: o serviço também precisa ser *confiável*. Um

microsserviço confiável é aquele no qual seus clientes, suas dependências e o ecossistema de microsserviços como um todo podem confiar. Um microsserviço confiável é aquele que realmente conquistou a confiança essencial e necessária para que atenda ao tráfego de produção.

Se por um lado a estabilidade está relacionada a atenuar os efeitos colaterais negativos que surgem com as mudanças e a confiabilidade está relacionada à confiança, as duas qualidades estão ligadas de maneira indissociável. Todo requisito de estabilidade também traz um requisito de confiabilidade: por exemplo, os desenvolvedores deveriam buscar não apenas processos estáveis de implantação, mas também garantir que cada implantação seja confiável do ponto de vista de seus clientes ou suas dependências.

A confiança resultante pode ser dividida em vários requisitos, do mesmo modo que determinamos os requisitos para a estabilidade. Por exemplo, podemos tornar nossos processos de implantação confiáveis garantindo que nossos testes de integração sejam abrangentes e nossas fases de implantação do tipo staging e pré-release sejam bem-sucedidas de modo que seja possível garantir que cada alteração introduzida na produção não contenha erros que possam comprometer seus clientes e suas dependências.

Ao agregar confiabilidade aos nossos microsserviços, estamos protegendo sua disponibilidade. Podemos armazenar dados em cache para que estes fiquem prontamente disponíveis para os serviços do cliente, ajudando-os a proteger seus SLAs ao tornar nossos próprios serviços altamente disponíveis. Para proteger nosso próprio SLA contra quaisquer problemas de disponibilidade de nossas dependências, podemos implementar o cache defensivo.

O último requisito de confiabilidade está relacionado ao roteamento e à descoberta. A disponibilidade requer que a comunicação e o roteamento entre diferentes serviços sejam confiáveis: verificações de saúde do sistema devem ser precisas, solicitações e respostas devem chegar a seus destinos, e os erros devem ser tratados de forma cuidadosa e adequada.

Requisitos de confiabilidade

Os requisitos para construir um microsserviço confiável são:

- um processo confiável de implantação
- planejar, atenuar e proteger-se contra falhas e dependências
- roteamento e descoberta confiáveis

Os detalhes dos requisitos de confiabilidade de uma aplicação pronta para produção são abordados no *Capítulo 3 – Estabilidade e confiabilidade.*

Escalabilidade

O tráfego do microsserviço raramente é estático ou constante, e uma das marcas de um microsserviço bem-sucedido (e de um ecossistema de microsserviços bem-sucedido) é um crescimento constante do tráfego. Os microsserviços precisam estar preparados para este crescimento, eles precisam comportá-lo facilmente e ser capazes de crescer ativamente com ele. Um microsserviço que não consegue acompanhar este crescimento experimenta uma crescente latência, baixa disponibilidade e, em casos extremos, um aumento drástico no número de incidentes e interrupções. A *escalabilidade* é essencial para a disponibilidade, o que faz dela nosso terceiro padrão de disponibilidade de produção.

Um microsserviço escalável é aquele que consegue tratar um grande número de tarefas ou solicitações ao mesmo tempo. Para garantir que um microsserviço seja escalável, precisamos conhecer (1) sua escala de crescimento qualitativo (por exemplo, se o crescimento se dá por visualizações de página, ou “page views”, ou pedidos dos clientes) e (2) sua escala de crescimento quantitativo (por exemplo, quantas solicitações por segundo ele consegue tratar). Uma vez conhecida a escala de crescimento, podemos planejar visando a futuras necessidades de capacidade e identificar os gargalos e os requisitos de recursos.

A forma como um microsserviço trata o tráfego também deve ser escalável. Ele deve estar preparado para rajadas de tráfego, tratá-las com atenção e evitar que elas derrubem o serviço completamente. É mais fácil falar do que fazer, mas sem um tratamento escalável do tráfego os desenvolvedores podem (e irão) se deparar com um ecossistema quebrado de microsserviços.

Uma complexidade adicional é introduzida pelo restante do ecossistema de microsserviços. É preciso se preparar para o inevitável tráfego adicional e

o crescimento dos clientes do serviço. Da mesma forma, quaisquer dependências do serviço devem ser alertadas quando forem esperados crescimentos de tráfego. A comunicação e a colaboração entre equipes são essenciais para escalabilidade: comunicar-se regularmente com os clientes e suas dependências sobre os requisitos de escalabilidade de um serviço, o status e quaisquer gargalos garante que quaisquer serviços que dependam um do outro estejam preparados para o crescimento e potenciais armadilhas.

Por fim, porém não menos importante, a maneira como um microsserviço armazena e lida com os dados também precisa ser escalável. Construir uma solução escalável de armazenamento ajuda bastante a garantir a disponibilidade de um microsserviço e é um dos fatores mais essenciais de um sistema verdadeiramente pronto para produção.

Requisitos de escalabilidade

Os requisitos para construir um microsserviço escalável são:

- escalas de crescimento quantitativas e qualitativas bem definidas
- identificação de gargalos e requisitos de recursos
- planejamento de capacidade cuidadoso e preciso
- tratamento escalável do tráfego
- o escalamento das dependências
- armazenamento escalável dos dados

Os detalhes dos requisitos de escalabilidade da disponibilidade de produção são abordados no *Capítulo 4 – Escalabilidade e desempenho.*

Tolerância a falhas e prontidão para catástrofes

Mesmo o mais simples dos microsserviços é um sistema bastante complexo. Como sabemos muito bem, sistemas complexos falham, eles falham com frequência, e qualquer cenário potencial de falha pode e irá acontecer em algum momento do tempo de vida de um microsserviço. Os microsserviços não vivem isolados, mas dentro das cadeias de dependência como parte de um ecossistema de microsserviços maior e incrivelmente complexo. A complexidade cresce linearmente com o número de

microsserviços no ecossistema geral, e garantir a disponibilidade não apenas de um microsserviço individual, mas do ecossistema como um todo, requer a adoção de outro padrão de disponibilidade de produção por parte de cada microsserviço. Todo microsserviço dentro do ecossistema precisa ser *tolerante a falhas* e estar *preparado para qualquer catástrofe*.

Um microsserviço tolerante a falhas e preparado para catástrofes é capaz de resistir tanto a falhas internas quanto externas. Falhas internas são aquelas que o microsserviço causa a si mesmo: por exemplo, erros de código que não são pegos pelos testes adequados levam a implantações ruins, causando interrupções que afetam todo o ecossistema. Catástrofes externas, como interrupções de datacenters e/ou um gerenciamento de configuração inadequado em todo o ecossistema, leva a interrupções que afetam a disponibilidade de todos os microsserviços e de toda a organização.

É possível se preparar adequadamente (embora não exaustivamente) para cenários de falhas e potenciais catástrofes. Identificar falhas e cenários de catástrofe é o primeiro requisito para construir um microsserviço tolerante a falhas e preparado para produção. Uma vez identificados esses cenários, começa o trabalho pesado de criar uma estratégia e um planejamento para quando eles ocorrerem. Isso deve acontecer em todos os níveis do ecossistema de microsserviços, e quaisquer estratégias compartilhadas devem ser comunicadas para toda a organização, para que a mitigação seja padronizada e previsível.

A padronização da mitigação e a resolução de falhas no âmbito organizacional significam que os incidentes e as interrupções de microsserviços individuais, dos componentes da infraestrutura ou do ecossistema como um todo precisam ser formatados em procedimentos cuidadosamente executados e facilmente compreensíveis. Os procedimentos de resposta a incidentes precisam ser tratados de forma coordenada, planejada e comunicada minuciosamente. Se os incidentes e as interrupções forem tratados dessa forma e a estrutura da resposta a incidentes for bem definida, as organizações poderão evitar longos períodos de downtime e proteger a disponibilidade dos microsserviços. Se cada desenvolvedor souber exatamente o que deve fazer em caso de interrupção, souber como mitigar e resolver problemas de forma rápida e adequada, e souber como escalar se um problema estiver além de sua

capacidade ou de seu controle, então os tempos de mitigação e resolução cairão drasticamente.

Tornar as falhas e catástrofes previsíveis significa um passo além de identificar os cenários de falha e catástrofe e se planejar para enfrentá-los. Significa forçar os microsserviços, a infraestrutura e o ecossistema a falhar de todas as formas conhecidas para testar a disponibilidade de todo o sistema. Isso é feito por meio de vários tipos de teste de resiliência. O teste do código (incluindo testes de unidades, testes de regressão e testes de integração) é o primeiro passo para testar a resiliência. O segundo passo é o teste de carga, no qual os microsserviços e os componentes da infraestrutura são testados em sua capacidade de lidar com mudanças drásticas de tráfego. O último, mais intenso e mais relevante tipo de teste de resiliência é o teste de caos, no qual os cenários de falhas são executados (tanto de forma programada quanto aleatória) nos serviços de produção para garantir que os microsserviços e os componentes da infraestrutura estejam realmente preparados para todos os cenários conhecidos de falhas.

Requisitos da tolerância a falhas e prontidão para catástrofes

Os requisitos para construir um microsserviço tolerante a falhas e preparado para qualquer catástrofe são:

- identificar e se planejar para os potenciais cenários de catástrofe e falhas;
- identificar e resolver os pontos únicos de falhas;
- estratégias de detecção e correção de falhas em funcionamento;
- teste de resiliência por meio de teste do código, teste de carga e teste de caos;
- o tráfego deve ser gerenciado cuidadosamente em preparação para falhas;
- incidentes e interrupções devem ser tratados de forma adequada e produtiva.

Os detalhes dos requisitos de tolerância a falhas e prontidão para catástrofes são abordados no *Capítulo 5 – Tolerância a falhas e prontidão para catástrofes*.

Desempenho

No contexto do ecossistema de microsserviços, escalabilidade (que já tratamos brevemente neste capítulo) está relacionada à quantidade de solicitações que um microsserviço consegue tratar. Nosso próximo princípio de disponibilidade de produção – *desempenho* – se refere a quão bem o microsserviço trata essas solicitações. Um microsserviço de alto desempenho trata as solicitações rapidamente, processa as tarefas de forma eficiente e usa adequadamente os recursos (como hardware e outros componentes de infraestrutura).

Um microsserviço que emite um grande número de custosas chamadas de rede, por exemplo, não apresenta bom desempenho. O mesmo vale para um microsserviço que processa e trata as tarefas simultaneamente nos casos em que um processamento assíncrono (não bloqueante) de tarefas aumentaria o desempenho e a disponibilidade do serviço. Identificar e eliminar esses problemas de desempenho é um rígido requisito de disponibilidade de produção.

Do mesmo modo, dedicar um grande número de recursos (como CPU) a um microsserviço que não os utiliza é ineficiente. A ineficiência reduz o desempenho: se isso não ficar claro no âmbito de microsserviços em todos os casos, as consequências se tornarão custosas e prejudiciais no âmbito do ecossistema. Recursos de hardware subutilizados afetam o resultado final, e o hardware não é barato. Existe uma diferença sutil entre subutilização e um adequado planejamento de capacidade, portanto os dois precisam ser planejados e compreendidos de forma conjunta para que a disponibilidade do microsserviço não seja comprometida e o custo da subutilização seja razoável.

Requisitos de desempenho

Os requisitos para construir um microsserviço de alto desempenho são:

- SLAs (service-level agreements) adequados de disponibilidade
- adequado tratamento e processamento de tarefas
- utilização eficiente de recursos

Os detalhes dos requisitos de desempenho para uma disponibilidade de produção são abordados no *Capítulo 4 – Escalabilidade e desempenho.*

Monitoramento

Outro princípio necessário para garantir a disponibilidade de um microsserviço é seu adequado *monitoramento*. Um bom monitoramento tem três componentes: um adequado logging (registro) de todas as informações importantes e relevantes; interfaces gráficas úteis (dashboards de controle) que sejam facilmente compreendidas por todo desenvolvedor na empresa e que reflitam com precisão a saúde dos serviços; e um sistema de alerta eficaz e acionável sobre as principais métricas.

O logging pertence a cada microsserviço e começa no código-base dele. Determinar com precisão quais informações registrar difere para cada serviço, mas o objetivo do logging é bem simples: quando ocorre um erro – mesmo um erro em uma implantação antiga –, o logging é necessário para determinar o que exatamente deu errado e onde as coisas começaram a descambar. Em ecossistemas de microsserviços não se recomenda atribuir versões aos microsserviços, portanto não haverá uma versão precisa à qual se referir para encontrar os erros ou problemas. O código é revisado frequentemente; implantações ocorrem muitas vezes por semana, recursos são adicionados constantemente e dependências estão sempre mudando, mas os logs permanecem os mesmos, preservando as informações necessárias para identificar qualquer problema. Certifique-se de que seus logs contenham as informações necessárias para detectar possíveis problemas.

Todas as principais métricas (como utilização de hardware, conexões ao database, respostas e tempos médios de respostas, e o status de endpoints de API) devem ser exibidas graficamente em tempo real em um dashboard de fácil acesso. Dashboards são um importante componente para construir um microsserviço pronto para produção e bem monitorado: eles ajudam a avaliar a saúde de um microsserviço com uma rápida olhada e permitem que desenvolvedores detectem padrões estranhos e anomalias que talvez não sejam extremas o suficiente para disparar os limites de alerta. Quando usados com sabedoria, os dashboards permitem que desenvolvedores detectem se um microsserviço está ou não funcionando corretamente por meio de uma simples olhada, mas não é necessário que os desenvolvedores observem os dashboards para detectar incidentes e interrupções, e o

retorno a versões anteriores estáveis deve ser totalmente automatizado.

A detecção real de falhas é feita por meio de alertas. Todas as principais métricas devem gerar alertas, incluindo (no mínimo) utilização de CPU e RAM, o número de descritores de arquivos (file descriptors) , o número de conexões ao database, o SLA do serviço, solicitações (requests) e respostas (responses), o status dos endpoints da API, erros e exceções, a saúde das dependências do serviço, informações sobre qualquer database e o número de tarefas que estão sendo processadas (se aplicável).

É preciso configurar limites do tipo normal, alerta e crítico para cada uma dessas métricas, e qualquer desvio da norma (isto é, quando os limites de alerta ou crítico forem atingidos) deve disparar um alerta para os desenvolvedores que estão de prontidão para o serviço. Os limites devem ser indicativos: altos o suficiente para evitar ruído, mas baixos o suficiente também para detectar qualquer problema real.

Os alertas devem ser úteis e acionáveis. Um alerta não acionável não é um alerta útil e um desperdício de horas de engenharia. Cada alerta acionável – isto é, *todo* alerta – deve ser acompanhado de um roteiro. Por exemplo, se um alerta for disparado em um número alto de exceções de certo tipo, então é preciso haver um roteiro contendo as estratégias de mitigação que qualquer desenvolvedor de prontidão possa usar enquanto tenta resolver o problema.

Requisitos de monitoramento

Os requisitos para construir um microsserviço adequadamente monitorado são:

- logging e rastreamento adequados em toda a pilha
- dashboards bem projetados que sejam fáceis de entender e reflitam com precisão a saúde do serviço
- alertas eficazes e acionáveis acompanhados de roteiros
- implementar e manter uma rotação das equipes de prontidão

Os detalhes dos requisitos de monitoramento são abordados no *Capítulo 6 – Monitoramento.*

Documentação

A arquitetura de microsserviços tem o potencial de causar maior defasagem técnica – uma das principais contrapartidas de se adotar os microsserviços. Como regra, a defasagem técnica tende a crescer *com* a velocidade do desenvolvedor: quanto mais rapidamente um serviço pode gerar iterações, ser alterado e implantado, mais frequentemente atalhos e patches são ativados. Clareza e estrutura organizacionais relativas à *documentação* e *compreensão* de um microsserviço contornam esta defasagem técnica e evitam muito a confusão, a falta de percepção e compreensão da arquitetura que tendem a acompanhar essa defasagem.

Reduzir a defasagem técnica não é a única razão para tornar uma boa documentação um dos princípios de disponibilidade de produção: fazê-lo seria como uma consideração *a posteriori* (uma importante consideração *a posteriori*, mas uma consideração *a posteriori* mesmo assim). Não, assim como os demais padrões de disponibilidade de produção, a documentação e sua contraparte (compreensão) influenciam de forma direta e mensurável a disponibilidade de um microsserviço.

Para observar a validade deste argumento, podemos analisar como as equipes de desenvolvedores trabalham juntas e compartilham seu conhecimento e a compreensão de um microserviço. Você mesmo pode fazê-lo colocando uma de suas equipes de desenvolvimento em uma sala em frente a um quadro branco e pedindo para eles esboçarem a arquitetura e todos os detalhes importantes do serviço. Garanto que você irá se surpreender com o resultado deste exercício e provavelmente descobrirá que o conhecimento e a compreensão do serviço não são coesos ou coerentes em todo o grupo. Um desenvolvedor conhecerá um aspecto sobre a aplicação que ninguém mais conhece, enquanto um segundo desenvolvedor terá uma compreensão tão diferente do microsserviço que você se perguntará se eles estão contribuindo para o mesmo código-base. Quando chegar o momento de as alterações de código serem revisadas, tecnologias trocadas ou recursos acrescentados, a falta de alinhamento do conhecimento e da compreensão resultará em um projeto e/ou uma evolução de microsserviços que não são prontos para produção, contendo sérios defeitos que prejudicam a capacidade do serviço de atender com confiabilidade o tráfego de produção.

Esta confusão e os problemas causados por ela podem ser evitados facilmente e com sucesso se exigirmos que cada microsserviço siga um

conjunto rigorosamente padronizado de requisitos de documentação. A documentação precisa conter todo o conhecimento essencial (fatos) sobre um microsserviço, incluindo um diagrama da arquitetura, um guia de integração e de desenvolvimento, detalhes sobre o fluxo de solicitações e quaisquer endpoints de API, e um roteiro de prontidão para cada alerta do serviço.

A compreensão de um microsserviço pode ser alcançada de várias formas. A primeira é fazendo o exercício que acabei de mencionar: coloque a equipe de desenvolvimento em uma sala de conferência e peça que seus componentes esbocem o diagrama de arquitetura do serviço. Graças à velocidade sempre maior de desenvolvimento, os microsserviços mudam radicalmente em diferentes momentos de seu ciclo de vida. Ao tornar estas revisões de arquitetura parte do processo de cada equipe e programá-las regularmente, você pode garantir que o conhecimento e a compreensão sobre quaisquer alterações no microsserviço sejam disseminados para toda a equipe.

Para abordar o segundo aspecto da compreensão de um microsserviço, precisamos subir um nível de abstração e considerar os próprios padrões de disponibilidade de produção. Uma grande parte da compreensão de um microserviço é capturada determinando se um microsserviço está pronto para produção ou não e qual seu status em relação aos padrões de disponibilidade de produção e seus requisitos individuais. Isso pode ser feito de muitas formas; uma das quais é executar auditorias para determinar se um microsserviço atende aos requisitos e então criar um roteiro para o serviço, detalhando como levá-lo até um estado de disponibilidade para produção. A verificação dos requisitos também pode ser automatizada para toda a organização. Trataremos de outros aspectos desta questão em mais detalhes na próxima seção, sobre a implementação de padrões de disponibilidade de produção em uma organização que adotou a arquitetura de microsserviços.

Requisitos de documentação

Os requisitos para construir um microsserviço bem documentado são:

- documentação completa, atualizada e centralizada contendo todas as informações relevantes e essenciais sobre o microsserviço
- compreensão organizacional nos níveis de desenvolvedor, equipe e

ecossistema

Os detalhes dos requisitos de documentação são abordados no *Capítulo 7 – Documentação e compreensão.*

Implementando a disponibilidade de produção

Temos agora um conjunto de padrões que se aplicam a todo microsserviço em qualquer ecossistema de microsserviços, cada um com seu conjunto próprio de requisitos específicos. Qualquer microsserviço que atenda a esses requisitos é confiável em termos de suprir o tráfego de produção e garantir um alto nível de disponibilidade.

Agora que temos os padrões de disponibilidade de produção, a pergunta que permanece é *como* podemos implementá-los em um ecossistema de microsserviços especializado e do mundo real. Passar dos princípios para a prática e aplicar a teoria a aplicações do mundo real demanda um nível significativo de dificuldade. Entretanto o poder desses padrões de disponibilidade de produção e dos requisitos que eles impõem está em sua incrível aplicabilidade e sua rígida granularidade: eles são genéricos o suficiente para ser aplicados a qualquer ecossistema e específicos o suficiente para fornecer estratégias concretas para implementação.

A padronização requer uma aceitação por parte de todos os níveis da organização e precisa ser adotada e estimulada tanto de cima para baixo quanto de baixo para cima. Nos níveis executivo e de liderança (gerencial e técnico), estes princípios precisam ser estimulados e suportados como requisitos de arquitetura para a organização de engenharia de software. No nível mais básico, dentro das equipes individuais de desenvolvimento, a padronização precisa ser abraçada e implementada. O importante é que a padronização precisa ser vista e comunicada não como um obstáculo ou gargalo para o desenvolvimento e a implantação, mas como um guia para o desenvolvimento e a implantação de aplicações prontas para produção.

Muitos desenvolvedores podem resistir à padronização. Afinal de contas, eles podem argumentar, o motivo para adotar a arquitetura de microsserviços não é oferecer mais velocidade de desenvolvimento, liberdade e produtividade? Responder a essas objeções não significa negar que a adoção da arquitetura de microsserviços traz liberdade e velocidade

para as equipes de desenvolvimento, mas concordar e salientar que é *exatamente* por isso que os padrões de disponibilidade de produção precisam ser adotados. A velocidade e a produtividade do desenvolvedor são paralisadas sempre que uma interrupção derruba o serviço, sempre que uma má implantação compromete a disponibilidade dos clientes e das dependências de um microsserviço, sempre que uma falha que *poderia ter sido evitada com testes adequados de resiliência* derruba todo o ecossistema de microserviços. Se há algo que aprendemos nos últimos cinquenta anos sobre desenvolvimento de software é que a padronização traz liberdade e reduz a entropia. Como diz Brooks em *The Mythical Man-Month* (O mítico homem-mês), talvez a maior coletânea de ensaios sobre a prática da engenharia de software, “a forma é libertadora”.

Uma vez que a organização de engenharia de software adotou e concordou em seguir os padrões de disponibilidade de produção, o próximo passo é avaliar e elaborar sobre cada um dos requisitos do padrão. Os requisitos apresentados aqui e detalhados ao longo do livro são muito gerais e precisam levar em conta o contexto, os detalhes específicos da organização e as estratégias de implementação. O que precisa ser feito é analisar cada padrão de disponibilidade de produção e seus requisitos e descobrir como cada requisito pode ser implementado na organização de engenharia de software. Por exemplo, se o ecossistema de microsserviços da organização tiver uma ferramenta de implantação do tipo autosserviço, então implementar um processo de implantação estável e confiável precisa ser comunicado em termos da ferramenta interna de implantação e de como ela funciona. Este exercício também pode resultar na reconstrução das ferramentas internas e/ou na adição de recursos a elas.

A real implementação dos requisitos e a determinação de que um dado microsserviço atenda a esses requisitos podem ser feitas pelos próprios desenvolvedores, pelos líderes de equipe, pela área de gerenciamento ou pelos engenheiros operacionais (de sistema, DevOps ou de confiabilidade). Tanto no Uber quanto nas várias empresas que adotaram a padronização da disponibilidade de produção, a implementação e a aplicação dos padrões de disponibilidade de produção são motivadas pelas organizações de engenharia de confiabilidade (SRE). Geralmente, SREs são responsáveis pela disponibilidade dos serviços, portanto aplicar esses padrões em todo o ecossistema de microsserviços se encaixa bem às

responsabilidades existentes. Isso não quer dizer que os desenvolvedores ou as equipes de desenvolvimento não tenham responsabilidade por garantir que seus serviços sejam prontos para produção; ao contrário, SREs informam, motivam e aplicam a disponibilidade de produção dentro do ecossistema de microsserviços, e a responsabilidade pela implementação recai sobre os SREs integrados às equipes de desenvolvimento e os próprios desenvolvedores.

Construir e manter um ecossistema de microsserviços prontos para produção não é um desafio fácil de empreender, mas as recompensas são ótimas e o impacto pode ser visto claramente na maior disponibilidade de cada microsserviço. Implementar padrões de disponibilidade de produção e seus requisitos fornece resultados mensuráveis e significa que as equipes de desenvolvimento podem trabalhar sabendo que os serviços dos quais dependem são confiáveis, estáveis, tolerantes a falhas, de alto desempenho, monitorados, documentados e preparados para qualquer catástrofe.

CAPÍTULO 3

Estabilidade e confiabilidade

Um microsserviço pronto para produção é estável e confiável. Tanto os microsserviços individuais quanto o ecossistema de microsserviços como um todo estão constantemente mudando e evoluindo, e quaisquer esforços feitos para aumentar a estabilidade e a confiabilidade de um microsserviço já são meio caminho andado em direção a garantir a saúde e a disponibilidade do ecossistema como um todo. Neste capítulo são exploradas diferentes maneiras de construir e executar um microsserviço estável e confiável, incluindo padronizar o processo de desenvolvimento, construir abrangentes pipelines de deployment, compreender as dependências e proteger-se contra suas falhas, construir um roteamento e uma descoberta estáveis e confiáveis e estabelecer os procedimentos adequados de descontinuação e desativação para microsserviços antigos e desatualizados e/ou seus endpoints.

Princípios para construir microsserviços estáveis e confiáveis

A própria arquitetura de microsserviços é propícia a um desenvolvimento acelerado. A liberdade oferecida pelos microsserviços significa que o ecossistema estará em um estado de contínua mudança, nunca estático, sempre evoluindo. Recursos serão acrescentados todos os dias; novas versões serão implantadas várias vezes por dia e antigas tecnologias serão trocadas por novas e melhores num ritmo surpreendente. Esta liberdade e a flexibilidade resultam em uma inovação real e tangível, mas implicam um alto custo.

Inovação, mais velocidade e produtividade do desenvolvedor, rápido avanço tecnológico e o ecossistema de microsserviço em constante mutação podem rapidamente estagnar se qualquer parte do ecossistema de microsserviços se tornar instável ou não confiável. Em alguns casos, basta

implantar uma versão defeituosa ou com erros de um microsserviço crítico para derrubar todo o negócio.

Um microsserviço *estável* é aquele para o qual o desenvolvimento, a implantação (deployment), a adoção de novas tecnologias e a descontinuação ou desativação de outros serviços não resultam em instabilidade de todo o amplo ecossistema de microsserviços. Isso requer adotar medidas para se proteger contra as consequências negativas que podem ser introduzidas por esses tipos de mudanças. Um microsserviço *confiável* é aquele no qual outros microsserviços e o ecossistema como um todo podem confiar. A estabilidade anda lado a lado com a confiabilidade, pois cada requisito de estabilidade traz consigo um requisito de confiabilidade (e vice-versa): por exemplo, processos estáveis de implantação são acompanhados pelo requisito de que cada nova implantação não comprometa a confiabilidade do microsserviço do ponto de vista de um de seus clientes ou suas dependências.

Há várias coisas que podem ser feitas para garantir que um microsserviço seja estável e confiável. É possível implementar um *ciclo de desenvolvimento* padronizado para se proteger contra práticas ruins de desenvolvimento. O processo de *implantação* pode ser concebido de modo que as alterações no código sejam forçadas a passar por vários estágios antes de ser implantadas em todos os servidores de produção. É possível se proteger contra falhas de *dependências*. Verificações de saúde, roteamento adequado e “circuit breakers” (mecanismos de proteção) podem ser implementados nos canais de *roteamento e descoberta* para tratar padrões anômalos de tráfego. Finalmente, os microsserviços e seus endpoints podem ser *descontinuados* e/ou *desativados* sem causar quaisquer falhas em outros microsserviços.

Um serviço pronto para produção é estável e confiável quando:

- ele tem um ciclo de desenvolvimento padronizado;
- seu código passa por minuciosos testes lint, de unidade, de integração e fim a fim;
- seu processo de teste, empacotamento, versão e liberação é completamente automatizado;
- ele tem um pipeline de deployment padronizado, contendo as fases

de staging (réplica), pré-release e produção;

- seus clientes são conhecidos;
- suas dependências são conhecidas e podem ser feitos backups e há alternativas, fallbacks (contingência) e cache em operação em caso de falhas;
- ele contém um sistema de roteamento e faz descobertas estáveis e confiáveis.

Ciclo de desenvolvimento

A estabilidade e a confiabilidade de um microsserviço começam com o desenvolvedor individual que cria código para o serviço. A maioria das interrupções e das falhas dos microsserviços é causada por erros introduzidos no código que não foram detectados na fase de desenvolvimento, em nenhum dos testes ou em nenhuma das etapas do processo de implantação. Mitigar e resolver essas interrupções e falhas geralmente implicam simplesmente retornar à mais recente versão estável, revertendo qualquer versão que contivesse o erro e reimplantando uma nova (e sem erros) versão do código.


O verdadeiro custo de um desenvolvimento instável e não confiável

Um ecossistema de microsserviços não é terra de ninguém. Toda interrupção, todo incidente e todo erro podem e irão custar milhares (se não milhões) de dólares para a empresa em horas de engenharia e receitas perdidas. É preciso ter salvaguardas em operação durante o ciclo de desenvolvimento (e, como veremos, no pipeline de desenvolvimento) para detectar todos os erros antes de eles chegarem à produção.

Um ciclo de desenvolvimento estável e confiável passa por várias etapas (Figura 3.1).

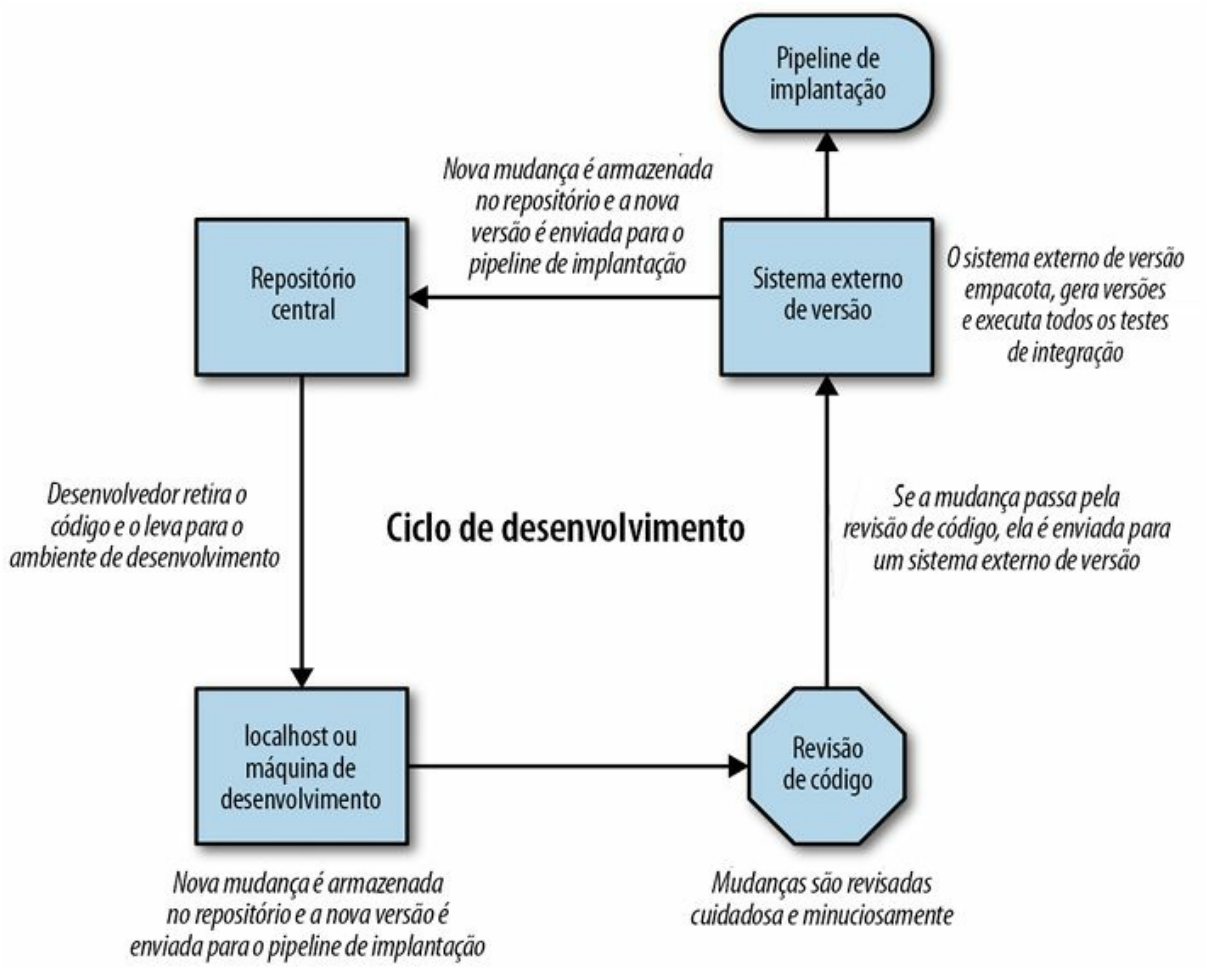


Figura 3.1. O ciclo de desenvolvimento

Primeiro, o desenvolvedor faz uma alteração no código. Geralmente isso começa com a retirada de uma cópia do código de um repositório central (normalmente usando git ou svn), a criação de um ramo individual onde ele fará as alterações, o acréscimo de suas alterações a seu ramo e a execução de quaisquer testes de unidade e integração. Este estágio do desenvolvimento pode ocorrer em qualquer lugar: no próprio local, no laptop de um desenvolvedor ou em um servidor num ambiente de desenvolvimento. Um ambiente confiável de desenvolvimento – que reflita com precisão o mundo da produção – é essencial, especialmente se testar o serviço em questão exigir enviar solicitações para outros microsserviços ou ler ou escrever dados em um database.

Depois que o código tiver sido enviado ao repositório central, o segundo passo consiste em ter as alterações cuidadosa e minuciosamente revisadas por outros engenheiros da equipe. Se todos os revisores aprovarem as alterações, e todos os testes lint, de unidade e de integração tiverem sido realizados em uma nova versão, a alteração poderá ser adicionada ao

repositório (veja o Capítulo 5 – *Tolerância a falhas e prontidão para catástrofes*, para mais informações sobre testes lint, de unidade e de integração). Só então a nova alteração pode ser introduzida no pipeline de deployment.

 Testes antes da revisão do código

Uma maneira de garantir que todos os erros sejam detectados antes de chegarem à produção é executar todos os testes lint, de unidade, de integração e fim a fim *antes* da fase de revisão do código. Isso é possível fazendo com que os desenvolvedores trabalhem em um ramo separado, iniciando todos os testes nesse ramo assim que o desenvolvedor submeter o código para revisão, e então permitindo que ele chegue à fase de revisão do código (ou apenas permitindo que ele seja construído) *depois* de ele ter passado com sucesso por todos os testes.

Como mencionado na seção sobre a camada 4 do ecossistema de microsserviços no Capítulo 1, *Microserviços*, muita coisa acontece entre o ciclo de desenvolvimento e o pipeline de deployment. A nova versão precisa ser empacotada (packaged), construída (built) e completamente testada antes de chegar ao primeiro estágio do pipeline de deployment.

Pipeline de deployment

Existe grande chance de erro humano em ecossistemas de microsserviços, especialmente quando se trata de práticas de implantação, e (como mencionei anteriormente) a maioria das interrupções em sistemas de produção de grande escala é causada por implantações ruins. Considere a dispersão organizacional que acompanha a adoção da arquitetura de microsserviços e o que isso acarreta no processo de implantação: temos, no mínimo, dezenas (se não centenas ou milhares) de equipes independentes e isoladas que estão implantando alterações em seus microsserviços segundo sua própria programação e frequentemente sem coordenação entre clientes e dependências. Se algo der errado, se um erro for introduzido na produção ou se um serviço ficar temporariamente indisponível durante a implantação, então todo o ecossistema poderá ser negativamente afetado. Para garantir que erros ocorram com menos frequência e que quaisquer falhas sejam detectadas antes de ser implantadas em todos os servidores de produção, introduzir um *pipeline de deployment* padronizado por toda a

organização de engenharia de software pode ajudar a garantir a estabilidade e a confiabilidade em todo o ecossistema.

Estou me referindo ao processo de implantação aqui como um “pipeline”, pois as implantações mais confiáveis são as que precisaram passar por um conjunto de testes antes de chegar aos servidores de produção. Podemos definir três estágios ou fases isoladas neste pipeline (Figura 3.2): primeiro, podemos testar uma nova versão em um ambiente do tipo *staging*; segundo, se essa versão passar pela fase de staging, poderemos implantá-la em um pequeno ambiente do tipo *pré-release (fase canary)*, no qual ela atenderá entre 5% e 10% do tráfego de produção; e terceiro, se ela passar pela fase de pré-release, poderemos implantá-la lentamente nos servidores de *produção* até que ela esteja implantada em todos os servidores.

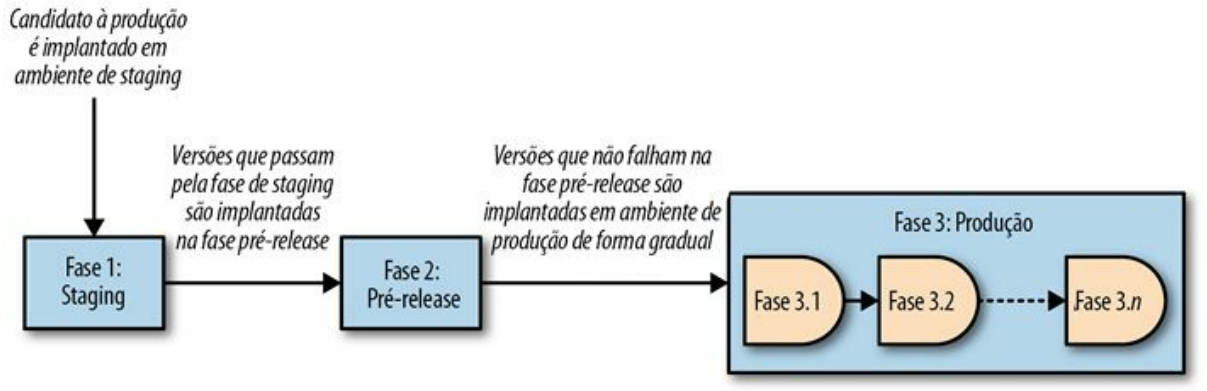


Figura 3.2. Estágios de um pipeline de deployment estável e confiável.

Staging

Qualquer nova versão pode primeiro ser implantada em um ambiente do tipo *staging*. Um ambiente desse tipo deve ser uma cópia exata do ambiente de produção: ele é um reflexo do estado do mundo real, mas sem o tráfego real. Ambientes do tipo staging geralmente não são executados na mesma escala que a produção (isto é, eles normalmente não são executados com o mesmo número de servidores que a produção, um fenômeno também conhecido como *paridade de servidor*), pois manter dois ecossistemas separados pode representar um grande custo de hardware para a empresa. Entretanto algumas organizações de engenharia de software podem determinar que a única maneira de replicar com precisão o ambiente de produção de forma estável e confiável é construir um ambiente staging idêntico com paridade de servidor.

Para a maioria das organizações de engenharia de software determinar a capacidade de hardware e a escala do ambiente staging como uma porcentagem do ambiente de produção geralmente é preciso o suficiente. A capacidade necessária de staging pode ser determinada pelo método que usaremos para testar o microsserviço na fase de staging. Para testar no ambiente staging, temos várias opções: podemos submeter o microsserviço a um tráfego simulado (ou gravado); podemos testá-lo manualmente acessando seus endpoints e avaliando suas respostas; podemos executar testes automatizados de unidade, de integração e outros testes especializados; ou podemos testar cada nova versão com qualquer combinação desses métodos.

 Trate os ambientes de staging e de produção como implantações separadas do mesmo serviço

Você pode ser tentado a executar as versões de staging e de produção como serviços separados e armazená-las em repositórios separados. Isso *pode* ser feito com sucesso, mas requer que as mudanças sejam sincronizadas em ambos os serviços e repositórios, incluindo mudanças de configuração (que geralmente são esquecidas). É muito mais fácil tratar as versões de staging e produção como “implantações” ou “fases” separadas do mesmo microsserviço.

Embora ambientes staging *sejam* ambientes de teste, eles diferem da fase de desenvolvimento e do ambiente de desenvolvimento, pois a versão implantada no ambiente staging é uma versão *candidata à produção*. Uma candidata à produção já deve ter passado com sucesso pelos testes lint, testes de unidade, testes de integração e revisão de código antes de ser implantada em um ambiente staging.

Implantar em um ambiente staging deve ser tratado pelos desenvolvedores com a mesma seriedade e o mesmo cuidado que se teria ao implantar em um ambiente de produção. Se uma versão for implantada com sucesso em um ambiente staging, ela poderá ser automaticamente implantada em ambientes do tipo pré-release, que *serão* submetidos a tráfego de produção.

Configurar ambientes staging em um ecossistema de microsserviços pode ser difícil por causa das complexidades introduzidas pelas dependências. Se seu microsserviço depende de nove outros microsserviços, então ele se baseia nessas dependências para oferecer respostas precisas quando

solicitações são enviadas e leituras ou gravações são feitas nos databases relevantes. Como consequência dessas complexidades, o sucesso de um ambiente staging depende da forma como ele é padronizado em toda a empresa.

Staging total

Há várias formas de configurar a fase de staging do pipeline de deployment. A primeira é o *staging total* (Figura 3.3), no qual um ecossistema separado do tipo staging é executado como uma cópia completa de todo o ecossistema de produção (embora não necessariamente com a paridade de servidor). O staging total usa a mesma infraestrutura básica da produção, mas existem várias diferenças importantes. Ambientes staging dos serviços são, no mínimo, acessíveis a outros serviços por meio de portas de frontend e backend específicas. O importante é que ambientes staging em um ecossistema do tipo staging total se comuniquem *apenas com os ambientes staging de outros serviços* e nunca enviem solicitações ou recebam respostas de serviços de produção (o que significa que enviar tráfego para portas de produção no ambiente staging está fora de questão).

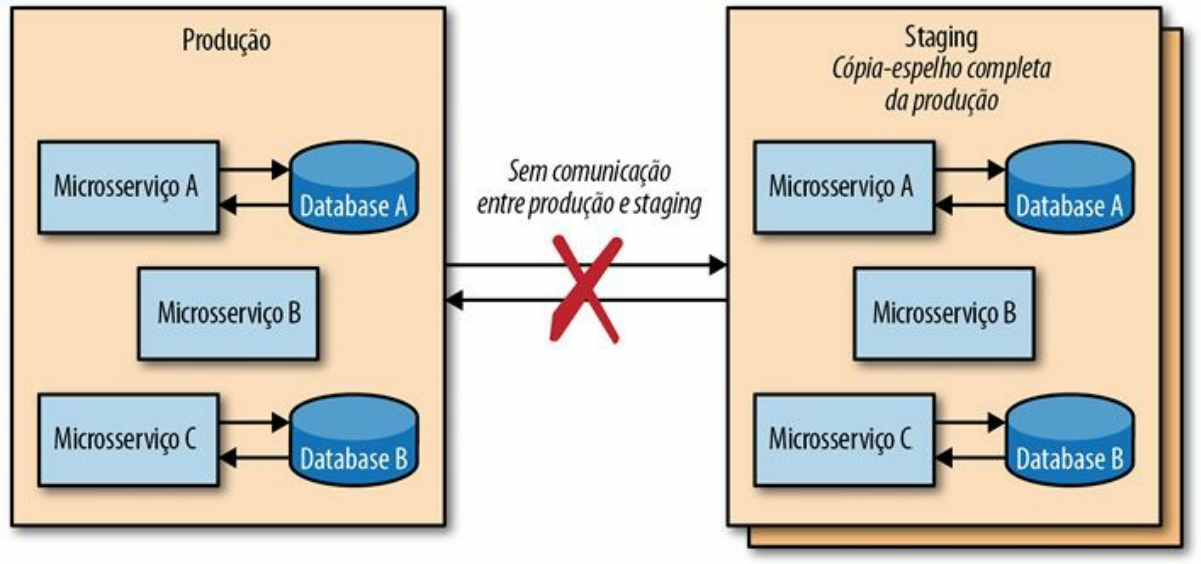


Figura 3.3. Staging total.

O staging total requer que todo microsserviço tenha um ambiente staging totalmente funcional, com o qual outros microsserviços possam se comunicar quando novas versões forem implantadas. A comunicação com outros microsserviços dentro do ecossistema do tipo staging pode ser feita por meio da escrita de testes que são iniciados quando uma nova versão é

implantada no ambiente staging ou, como mencionado, submetendo o serviço que está sendo implantado a um antigo tráfego de produção gravado, juntamente com todas as dependências upstream e downstream.

O staging total também requer um tratamento cuidadoso dos dados de teste: ambientes de staging *nunca* devem ter acesso de escrita a quaisquer databases de produção, e também não se recomenda fornecer acesso de leitura aos databases de produção. Uma vez que o staging total tiver sido projetado para ser uma cópia integral da produção, o ambiente staging de cada microsserviço deverá conter um database de teste separado com acesso de leitura e escrita.


Riscos do staging total

É preciso tomar cuidado ao implementar e implantar ambientes do tipo staging total, pois novas versões dos serviços quase sempre se comunicarão com outras novas versões de quaisquer dependências upstream e downstream – isso pode não ser um reflexo preciso do mundo real. Organizações de engenharia de software talvez precisem solicitar que suas equipes coordenem e/ou programem implantações em ambientes staging para evitar que a implantação de um serviço afete negativamente o ambiente staging para todos os outros serviços relacionados.

Staging parcial

O segundo tipo de ambiente do tipo staging é conhecido como *staging parcial*. Como o nome sugere, ele não é uma cópia completa do ambiente de produção. Em vez disso, cada microsserviço tem seu próprio ambiente staging, que é um pool de servidores com (no mínimo) portas de frontend e de backend específicas; e quando novas versões são introduzidas na fase de staging, elas se comunicam com os clientes upstream e as dependências downstream que estão rodando no ambiente de produção (Figura 3.4).

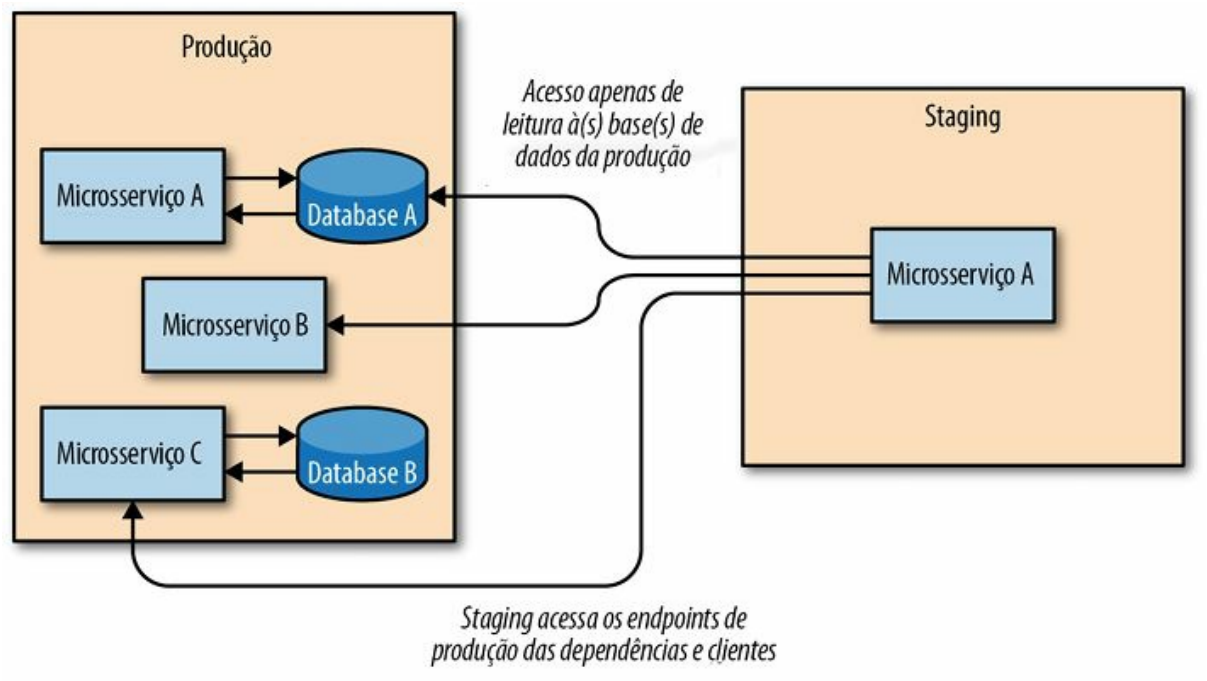


Figura 3.4. Staging parcial

Implantações em ambientes de staging parcial devem acessar todos os endpoints de produção dos clientes e dependências de um microsserviço para replicar o estado do mundo real o mais próximo possível. Testes específicos de staging precisarão ser escritos e executados para realizar isso, e todo novo recurso adicionado deve provavelmente ser acompanhado de pelo menos um teste adicional de staging para garantir que ele será testado completamente.


Riscos do staging parcial

Uma vez que microsserviços com ambientes de staging parcial se comuniquem com microsserviços de produção, será preciso extremo cuidado. Mesmo que o staging parcial seja limitado a solicitações apenas de leitura, os serviços de produção podem facilmente ser derrubados em virtude de implantações ruins em ambiente staging que enviem solicitações ruins e/ou sobrecarreguem os serviços de produção com muitas solicitações.

Esses tipos de ambientes staging também devem ter acesso restrito apenas de leitura ao database: um ambiente staging nunca escreve em um database de produção. Entretanto alguns microsserviços podem usar muitas operações de escrita, e testar a funcionalidade de escrita de uma nova versão será essencial. A forma mais comum de fazê-lo é marcar qualquer dado escrito por um ambiente staging como *dados de teste* (isto é conhecido como *locação de teste – test tenancy*), mas a maneira mais

segura é escrever em um database de teste separado, já que dar acesso de escrita em um ambiente staging ainda corre o risco de alterar dados do mundo real. Veja a Tabela 3.1 para comparar os ambientes de staging total e parcial.

Tabela 3.1 – Ambientes de staging total x parcial

	Staging total	Staging parcial
Cópia completa do ambiente de produção	Sim	Não
Portas de staging frontend e de backend separadas	Sim	Sim
Acesso a serviços de produção	Não	Sim
Acesso de leitura aos databases de produção	Não	Sim
Acesso de escrita aos databases de produção	Não	Sim
Requer rollbacks automatizados	Não	Sim

Ambientes de staging (total ou parcial) devem ter dashboards de controle, monitoramento e logging como os ambientes de produção – e todos devem ser configurados de forma idêntica aos seus equivalentes do ambiente de produção do microsserviço (veja o Capítulo 6, *Monitoramento*). Os gráficos para todas as principais métricas podem ficar no mesmo dashboard das métricas de produção, embora as equipes optem por separar os dashboards para cada parte do processo de implantação: um dashboard de staging, um dashboard de pré-release e um dashboard de produção. Dependendo de como os dashboards forem configurados, será melhor manter todos os gráficos de todas as implantações em um dashboard e organizá-los por implantação (ou métrica). Independentemente de como uma equipe decida configurar seus dashboards, o objetivo de criar dashboards de disponibilidade de produção bons e úteis não deve ser esquecido: os dashboards de um microsserviço pronto para produção devem facilitar que alguém de fora determine rapidamente a saúde e o status do serviço.

Monitorar e gravar logs para o ambiente staging deve ser idêntico a monitorar e gravar logs para a implantação de staging e de produção, de modo que quaisquer falhas nos testes e erros em novas versões implantadas na fase de staging sejam detectadas antes de passarem para a próxima fase do pipeline de deployment. É extremamente útil configurar alertas e logs para que eles sejam diferenciados e separados por tipo de implantação, garantindo que quaisquer alertas disparados por falhas ou erros especifiquem qual ambiente estará passando pelo problema, tornando

a depuração, a mitigação e a resolução de erros ou falhas bastante fáceis e diretas.

O propósito de um ambiente de staging é detectar quaisquer erros introduzidos pelas alterações no código antes que afetem o tráfego de produção. Quando um erro é introduzido pelo código, geralmente é detectado no ambiente de staging (se este for configurado corretamente). Rollbacks automatizados de implantações com defeito são uma necessidade para ambientes de staging parcial (embora não sejam necessários para ambientes de staging total). Estabelecer quando retornar a uma versão anterior deve ser determinado por vários limites nas métricas principais do microsserviço.

Uma vez que o staging parcial requeira a interação com microsserviços sendo executados em produção, os erros introduzidos pelas novas versões implantadas em um ambiente de staging parcial poderão derrubar outros microsserviços que estiverem sendo executados em produção. Se não houver rollbacks automatizados em funcionamento, a mitigação e a resolução desses problemas deverão ser realizadas manualmente. Quaisquer passos do processo de implantação que precisem de intervenção manual são pontos de falha não apenas para o próprio microsserviço, mas para todo o ecossistema de microsserviços.

A última pergunta que uma equipe de microsserviços deve responder durante a configuração de um ambiente de staging é por quanto tempo uma nova versão deve ser executada em staging antes de poder ser implantada no ambiente de pré-release (e, em seguida, no ambiente de produção). A resposta a esta pergunta é determinada pelos testes específicos da fase de staging: uma nova versão está pronta para avançar para a próxima fase do processo de implantação assim que tiver passado por todos os testes sem falhas.

Pré-release (fase canary)

Assim que uma nova versão tiver sido implantada com sucesso no ambiente de staging e passado por todos os testes necessários, ela pode ser implantada no próximo estágio do pipeline de deployment: o ambiente *de pré-release (ou canary em inglês)*. O nome inusitado deste ambiente vem de uma tática usada por mineiros de carvão: eles levavam canários para dentro das minas de carvão para monitorar os níveis de monóxido de

carbono no ar; se o canário morresse, eles sabiam que o nível de gás tóxico no ar estava alto e saíam imediatamente das minas[1]. Enviar uma nova versão para um ambiente de pré-release tem a mesma finalidade: implante a versão em um pequeno grupo de servidores com tráfego de produção (em torno de 5% a 10% da capacidade de produção) e, se ela sobreviver, instale-a no restante dos servidores de produção.


Distribuição de tráfego de pré-release

Se o serviço de produção for implantado em vários datacenters, diversas regiões ou em fornecedores de serviços de nuvem diferentes, então o conjunto de servidores de pré-release deverá conter servidores em cada um deles para que eles sejam uma amostra precisa da produção.

Uma vez que um ambiente de pré-release atende ao tráfego de produção, ele deve ser considerado parte da produção. Ele deve ter as mesmas portas de backend e de frontend, e os servidores de pré-release devem ser escolhidos aleatoriamente dentro do conjunto de servidores de produção para garantir uma amostragem precisa do tráfego de produção. Os servidores de pré-release podem (e devem) ter acesso total aos serviços de produção: eles devem acessar todos os endpoints de produção das dependências upstream e downstream, e eles devem ter acesso de leitura e escrita a quaisquer databases (se aplicável).

Da mesma forma que o ambiente de staging, os dashboards, o monitoramento e o registro de logs devem ser os mesmos para a fase de pré-release e para a produção. Alertas e logs devem ser diferenciados e identificados como originários da implantação de pré-release para que os desenvolvedores possam facilmente mitigar, depurar e resolver quaisquer problemas.


Portas separadas para ambiente de pré-release e produção

Alocar portas de frontend e backend separadas para pré-release e produção de modo que o tráfego possa ser direcionado deliberadamente pode parecer uma boa ideia, mas infelizmente separar o tráfego desta maneira anula a finalidade dos ambientes de pré-release: amostrar aleatoriamente o tráfego de produção em um pequeno conjunto de servidores para testar uma nova versão.

Rollbacks automatizados são uma necessidade absoluta para ambientes de

pré-release: se ocorrer algum erro conhecido, o sistema de implantação precisa automaticamente retornar à última versão estável conhecida. Lembre-se de que os servidores de pré-release atendem ao tráfego de produção, e quaisquer problemas que surgirem afetarão o mundo real.

Por quanto tempo uma nova versão deve permanecer nos servidores de pré-release até que os desenvolvedores estejam convencidos de que ela está pronta para produção? Podem ser minutos, horas ou mesmo dias, e a resposta é determinada pelos padrões de tráfego do microsserviço. O tráfego de todo microsserviço apresenta algum tipo de padrão, não importa o quão estranho seu microsserviço ou negócio seja. Uma nova versão só deve sair do estágio de pré-release da implantação quando um ciclo de tráfego completo tiver sido concluído. A definição de um "ciclo de tráfego" precisa ser padronizada em toda a organização de engenharia de software, mas a duração e os requisitos do ciclo de tráfego talvez precisem ser criados conforme cada serviço.

Produção

Produção é o mundo real. Quando uma versão passou com sucesso pelo ciclo de desenvolvimento, sobreviveu ao ambiente de staging e às minas de carvão da fase canário (pré-release), ela está pronta para ser implantada no ambiente de produção. Neste ponto do pipeline de deployment – o último passo – a equipe de desenvolvimento deve estar totalmente confiante na nova versão. Quaisquer erros no código devem ter sido descobertos, mitigados e resolvidos antes de se chegar a este ponto.

Toda versão que chegou ao estágio de produção deve ser completamente estável e confiável. Uma versão sendo implantada na produção já deve ter sido completamente testada e uma versão *nunca* deve ser implantada na produção sem ter passado pelas fases de staging e pré-release sem erros. A implantação no ambiente de produção pode ser feita de uma vez só depois que a versão tiver passado pela fase de pré-release ou pode ser gradualmente implantada em estágios: os desenvolvedores podem escolher implantar no ambiente de produção por porcentagem do hardware (por exemplo, primeiro em 25% de todos os servidores, depois 50%, então 75% e finalmente 100%) ou por datacenter, região, país ou qualquer combinação desses critérios.

Impondo uma implantação estável e confiável

Quando uma nova candidata à produção tiver passado pelo processo de desenvolvimento, tiver sobrevivido ao ambiente de staging e sido implantada com sucesso na fase de pré-release, as chances de ela causar uma severa interrupção são muito pequenas, pois a maioria dos erros no código já terá sido detectada antes de a candidata à produção ser implantada na produção. É exatamente por isso que ter um pipeline abrangente de implantação é essencial para construir um microsserviço estável e confiável.

Para alguns desenvolvedores, o atraso introduzido pelo pipeline de deployment pode parecer um fardo desnecessário, pois atrasa a implantação imediata de suas alterações de código e/ou novos recursos no ambiente de produção. Na realidade, o atraso introduzido pelas fases do pipeline de deployment é muito pequeno e facilmente personalizável, mas é preciso impor um processo padronizado de implantação para garantir a confiabilidade. Implantar um microsserviço várias vezes por dia pode (e irá) comprometer a estabilidade e a confiabilidade do microsserviço e quaisquer outros serviços em sua complexa cadeia de dependências: um microsserviço que muda de hora em hora raramente é estável ou confiável.

Os desenvolvedores podem ser tentados a pular as fases de staging e pré-release do processo de implantação e instalar uma correção diretamente na produção se, por exemplo, um erro grave for descoberto na produção. Embora isso resolva o problema rapidamente, *possivelmente* evite que a empresa perca receita e impeça que as dependências sofram interrupções, só deve-se permitir que os desenvolvedores implantem diretamente no ambiente de produção no caso de interrupções mais graves. Sem essas restrições em vigor, há sempre a infeliz possibilidade de fazer mau uso do processo e implantar diretamente na produção; para a maioria dos desenvolvedores, cada mudança no código, cada implantação é importante e pode parecer importante o suficiente para pular as fases de staging e pré-release, comprometendo a estabilidade e a confiabilidade de todo o ecossistema de microsserviços. Quando ocorrem erros, as equipes de desenvolvimento devem ser encorajadas a sempre retornar à última versão estável do microsserviço, trazendo-o de volta para um estado conhecido (e estável), que pode ser executado em produção sem qualquer problema enquanto a equipe trabalha para descobrir a causa da falha ocorrida.


Correções de emergência são um antipadrão

Quando um pipeline de deployment está em funcionamento, nunca deve haver uma implantação direta no ambiente de produção, exceto em caso de emergência, mas mesmo isso deve ser desencorajado. Pular as fases iniciais do pipeline de deployment geralmente introduz novos erros na produção, já que as correções de emergência no código correm o risco de não estar adequadamente testadas. Em vez de implantar uma correção diretamente na produção, os desenvolvedores devem retornar à última versão estável, se possível.

Uma implantação estável e confiável não se limita apenas a seguir o pipeline de deployment, e há vários casos em que impedir que um particular microsserviço seja implantado pode aumentar a disponibilidade no ecossistema.

Se um serviço não cumpre seus SLAs (veja o Capítulo 2, *Disponibilidade de produção*), todas as implantações podem ser adiadas se a cota de downtime do serviço tiver se esgotado. Por exemplo, se um serviço tiver um SLA que prometa uma disponibilidade de 99,99% (o que permite 4,38 minutos de downtime por mês), mas tiver ficado indisponível por 12 minutos em um mês, então novas implantações desse microsserviço poderão ser bloqueadas pelos próximos três meses, garantindo que ele cumpra seu SLA. Se um serviço falha no teste de carga (veja o Capítulo 5, *Tolerância a falhas e prontidão para catástrofes*), então a implantação em ambiente de produção pode ser bloqueada até que o serviço seja capaz de passar com sucesso por quaisquer testes de carga necessários. Para serviços que são críticos para o negócio, cujas interrupções impediriam que a empresa funcionasse corretamente, às vezes é necessário bloquear a implantação se eles não atenderem aos critérios de disponibilidade de produção estabelecidos pela organização de engenharia de software.

Dependências

A adoção da arquitetura de microsserviços às vezes é motivada pela ideia de que os microsserviços podem ser construídos e executados isoladamente, como componentes totalmente independentes e substituíveis de um sistema maior. Isso é verdade, a princípio, mas no mundo real todo microsserviço tem *dependências* tanto upstream quanto downstream. Todo microsserviço recebe solicitações de *clientes* (outros microsserviços) que contam com esse serviço para funcionar da forma esperada e cumprir seus

SLAs, assim como dependências downstream (outros serviços) dos quais ele depende para executar sua tarefa.

Construir e executar microsserviços prontos para produção são ações que exigem que os desenvolvedores se preparem para falhas de dependência, as mitiguem e se protejam contra elas. Entender as dependências de um serviço e se preparar para suas falhas são dos aspectos mais importantes da construção de um microsserviço estável e confiável.

Para entender a importância disso, vamos considerar um exemplo de microsserviço chamado *recebimento-transmissor*, cujo SLA é quatro noves (promessa de 99,99% de disponibilidade para os clientes upstream). O *recebimento-transmissor* depende de vários outros microsserviços, incluindo um microsserviço chamado *clientes* (que trata todas as informações do cliente) e um chamado *pedidos* (que trata informações sobre os pedidos feitos por cada cliente). Tanto o microsserviço *clientes* quanto o *pedidos* dependem de outros microsserviços: o microsserviço *clientes* depende de outro microsserviço que chamaremos de *clientes-dependência*, e o *pedidos* depende de um que chamaremos de *pedidos-dependência*. As chances de que *clientes-dependência* e *pedidos-dependência* tenham suas próprias dependências são muito grandes, portanto o gráfico de dependência do microsserviço *recebimento-transmissor* rapidamente se torna muito, muito complicado.

Uma vez que o microsserviço *recebimento-transmissor* quer proteger seu SLA e oferece um uptime de 99,99% para todos os seus clientes, sua equipe precisa garantir que os SLAs de todas as dependências downstream sejam estritamente obedecidos. Se o SLA do microsserviço *recebimento-transmissor* depender de o microsserviço *clientes* estar disponível 99,99% do tempo, mas o real uptime de *clientes* for apenas de 89,99% do tempo, a disponibilidade de *recebimento-transmissor* ficará comprometida e agora será apenas de 89,98%. Cada uma das dependências de *recebimento-transmissor* poderá sofrer o mesmo impacto sobre sua disponibilidade se qualquer uma das dependências na cadeia de dependências não cumprir seus SLAs.

Um microsserviço estável e confiável precisa mitigar falhas de dependência desse tipo (e, sim, não cumprir um SLA é uma falha!). Isso pode ser feito por meio de backups, fallbacks, cache e/ou alternativas para cada dependência caso elas apresentem falhas.

Antes de se preparar e mitigar as falhas de dependência, as dependências de um microsserviço precisam ser conhecidas, documentadas e rastreadas. Qualquer dependência que possa prejudicar o SLA de um microsserviço precisa ser incluída no diagrama e na documentação da arquitetura do microsserviço (veja o Capítulo 7, *Documentação e compreensão*) e deve ser incluída no dashboard de controle do serviço (veja Capítulo 6, *Monitoramento*). Além disso, todas as dependências devem ser rastreadas criando-se automaticamente gráficos de dependência para cada serviço, o que pode ser feito implementando-se um sistema de rastreamento distribuído em todos os microsserviços da organização.

Assim que todas as dependências forem conhecidas e rastreadas, o próximo passo será configurar backups, alternativas, fallbacks ou cache para cada dependência. A maneira correta de fazê-lo depende completamente das necessidades do serviço. Por exemplo, se a funcionalidade de uma dependência pode ser satisfeita acessando-se o endpoint de outro serviço, então uma falha da dependência primária deve ser tratada pelo microsserviço de modo que as solicitações sejam enviadas para a alternativa. Se as solicitações que precisam ser enviadas para a dependência puderem permanecer em uma fila quando a dependência estiver indisponível, então será preciso implementar uma fila. Outra maneira de tratar falhas de dependência é colocar em funcionamento um cache para a dependência dentro do serviço: salve dados relevantes em cache para que quaisquer falhas sejam tratadas de forma elegante.

O tipo de cache usado com mais frequência nestes casos é o cache LRU (*Least Recently Used* – Menos Usado Recentemente), no qual dados relevantes são mantidos em uma fila e quaisquer dados não usados são apagados quando a fila do cache esgota sua capacidade. Caches LRU são fáceis de implementar (em geral, uma única linha de código para cada instância), eficientes (não é preciso fazer custosas chamadas de rede), de alto desempenho (os dados estão disponíveis imediatamente) e são eficientes em mitigar quaisquer falhas de dependência. Isso é conhecido como *cache defensivo* e é útil para proteger um microsserviço contra falhas de suas dependências: salve em cache as informações que seu microsserviço recebe de suas dependências, e se as dependências ficarem inacessíveis, a disponibilidade de seu microsserviço não será afetada. Implementar cache defensivo não é necessário para cada dependência, mas

se uma dependência ou um conjunto de dependências específicas não for confiável, o cache defensivo evitará que seu microsserviço seja prejudicado.

Roteamento e descoberta

Outro aspecto de construir microsserviços estáveis e confiáveis é garantir que a comunicação e a interação entre microsserviços sejam estáveis e confiáveis, o que significa que a camada 2 (a camada de comunicação) do ecossistema de microsserviços (veja o Capítulo 1, *Microserviços*) deve ser construída para funcionar de modo a se proteger contra padrões prejudiciais de tráfego e manter a confiança por todo o ecossistema. As partes da camada de comunicação relevantes para a estabilidade e a confiabilidade (além da própria rede) são a descoberta de serviços (service discovery), o registro de serviços (service registry) e o balanceamento de carga.

A saúde de um microsserviço, tanto no âmbito do servidor quanto no âmbito do serviço como um todo, deve sempre ser conhecida. Isso significa que *verificações de saúde* devem ser feitas constantemente para que uma solicitação nunca seja enviada para um servidor ou um serviço com problemas. Fazer verificações de saúde em um canal separado (não usado para comunicação geral do microsserviço) é a forma mais fácil de garantir que as verificações de saúde nunca sejam comprometidas por algo como uma rede congestionada. Codificar respostas “200 OK” em um endpoint */health* para verificações de saúde não é ideal para todo microsserviço, embora seja suficiente para a maioria. Respostas codificadas não dizem muita coisa, exceto que o microsserviço foi iniciado no servidor de forma parcialmente bem-sucedida: qualquer endpoint */health* de um microsserviço deve dar uma resposta útil e precisa.

Se uma instância de um serviço em um servidor não for saudável, os balanceadores de carga não deverão mais rotear tráfego para ela. Se um microsserviço como um todo não for saudável (com as verificações de saúde falhando em certa porcentagem de servidores ou em todos os servidores em produção), então o tráfego não deverá mais ser roteado para aquele microsserviço particular até que os problemas que causaram a falha nas verificações de saúde sejam resolvidos.

Entretanto, verificações de saúde não devem ser o único fator que determina se um serviço é saudável ou não. Um grande número de exceções não tratadas também deve fazer com que um serviço seja considerado não saudável, e *circuit breakers* devem ser ativados para essas falhas de modo que se um serviço apresentar uma quantidade anormal de erros, as solicitações deixarão de ser enviadas para o serviço até que o problema seja resolvido. O essencial em um roteamento e numa descoberta estáveis e confiáveis é preservar o ecossistema de microsserviços evitando que maus atores atendam ao tráfego de produção e aceitem solicitações de outros microsserviços.

Descontinuação e desativação

Uma causa geralmente esquecida e ignorada de instabilidade e não confiabilidade em ecossistemas de microsserviços é a *descontinuação ou desativação* de um microsserviço ou de um de seus endpoints de API. Quando um microsserviço não está mais em uso ou não é mais suportado por uma equipe de desenvolvimento, sua desativação deve ser executada com atenção para garantir que nenhum cliente seja comprometido. A descontinuação de um ou mais endpoints de API de um microsserviço é ainda mais comum: quando novos recursos são adicionados ou antigos são removidos, os endpoints geralmente mudam, exigindo que as equipes de cliente sejam atualizadas e quaisquer solicitações feitas para os antigos endpoints sejam redirecionadas para novos endpoints (ou removidas totalmente).

Na maioria dos ecossistemas de microsserviços a descontinuação e a desativação são mais um problema sociológico dentro da organização de engenharia de software do que um problema técnico, o que torna a questão ainda mais difícil de tratar. Quando um microsserviço está prestes a ser desativado, sua equipe de desenvolvimento precisa lembrar-se de alertar todos os serviços de cliente e aconselhá-los sobre como lidar com a perda de sua dependência. Se o microsserviço sendo desativado for substituído por outro microsserviço novo ou se a funcionalidade do microsserviço for construída em outro microsserviço existente, então a equipe deverá ajudar todos os clientes a atualizar seus microsserviços para enviar solicitações aos novos endpoints. A descontinuação de um endpoint segue um processo semelhante: os clientes precisam ser alertados e informados sobre o novo

endpoint ou aconselhados sobre como lidar com a perda definitiva do endpoint. Tanto na descontinuação quanto na desativação, o monitoramento desempenha um papel crítico: os endpoints precisarão ser monitorados de perto *antes* de o serviço ou o endpoint ser completamente desativado e/ou descontinuado para verificar se há solicitações que ainda estão sendo enviadas para o serviço ou o endpoint desatualizado.

Por outro lado, não conseguir descontinuar um endpoint ou desativar um microsserviço adequadamente também pode ter efeitos desastrosos no ecossistema de microsserviços. Isso acontece com mais frequência do que os desenvolvedores gostariam de admitir. Em um ecossistema contendo centenas ou milhares de microsserviços, os desenvolvedores são frequentemente deslocados entre equipes, as prioridades mudam e tanto os microsserviços quanto as tecnologias são substituídos por novos e melhores o tempo todo. Quando esses antigos microsserviços ou tecnologias continuam sendo executados sem qualquer (ou muito) envolvimento, supervisão ou monitoramento, as falhas passam despercebidas e qualquer falha percebida pode demorar a ser resolvida. Se um microsserviço for largado à própria sorte, ele correrá o risco de comprometer seus clientes em caso de uma interrupção – tais microsserviços devem ser desativados em vez de abandonados.

Nada é mais prejudicial para um microsserviço do que a completa perda de uma de suas dependências. Nada causa mais instabilidade e falta de confiabilidade do que a repentina e inesperada falha de uma de suas dependências, mesmo que a falha tenha sido prevista por outra equipe. A importância de uma descontinuação e desativação estáveis e confiáveis não pode ser honestamente enfatizada o suficiente.

Avalie seu microsserviço

Agora que você tem um melhor entendimento da estabilidade e da confiabilidade, use a seguinte lista de perguntas para avaliar a disponibilidade de produção de seu(s) microsserviço(s) e do ecossistema de microsserviços. As perguntas são organizadas por tópico e correspondem às seções dentro deste capítulo.

Ciclo de desenvolvimento

- O microsserviço tem um repositório central, no qual todo o código está armazenado?
- Os desenvolvedores trabalham em um ambiente de desenvolvimento que reflete com precisão o estado da produção (por exemplo, que reflete com precisão o mundo real)?
- Existem testes adequados do tipo lint, de unidade, de integração e fim a fim em operação para o microsserviço?
- Existem procedimentos e políticas de revisão de código em vigor?
- O processo de teste, empacotamento, build e liberação (release) é automatizado?

Pipeline de deployment

- O ecossistema de microsserviços tem um pipeline de deployment padronizado?
- Existe uma fase de staging total ou parcial no pipeline de deployment?
- Qual o acesso que o ambiente de staging tem aos serviços de produção?
- Existe uma fase de pré-release no pipeline de deployment?
- As implantações são executadas na fase de pré-release por um período de tempo longo o suficiente para detectar quaisquer falhas?
- A fase de pré-release trata com precisão uma amostra aleatória do tráfego de produção?
- As portas do microsserviço são as mesmas para o ambiente de pré-release e de produção?
- As implantações no ambiente de produção são feitas todas ao mesmo tempo ou gradualmente?
- Existe um procedimento em vigor para pular as fases de staging e pré-release em caso de emergência?

Dependências

- Quais são as dependências deste microsserviço?
- Quais são seus clientes?
- Como este microsserviço mitiga as falhas de dependência?
- Existem backups, alternativas, fallbacks ou cache defensivo para cada

dependência?

Roteamento e descoberta

- As verificações de saúde do microsserviço são confiáveis?
- As verificações de saúde refletem com precisão a saúde do microsserviço?
- As verificações de saúde são realizadas em um canal separado dentro da camada de comunicação?
- Existem circuit breakers ativados para evitar que microsserviços não saudáveis façam solicitações?
- Existem circuit breakers ativados para evitar que o tráfego de produção seja enviado para servidores e microsserviços não saudáveis?

Descontinuação e desativação

- Existem procedimentos em funcionamento para desativar um microsserviço?
- Existem procedimentos em funcionamento para descontinuar endpoints de API de um microsserviço?

---

1 Os canários, por serem mais sensíveis que os humanos, conseguem detectar essas alterações mais facilmente.

CAPÍTULO 4

Escalabilidade e desempenho

Um microsserviço pronto para produção é escalável e de alto desempenho. Um microsserviço escalável e de alto desempenho é orientado por eficiência, e não apenas é capaz de tratar um grande número de tarefas ou solicitações ao mesmo tempo, mas também pode tratá-las eficientemente e está preparado para o crescimento futuro do número de tarefas e solicitações. Neste capítulo são abordados os componentes essenciais da escalabilidade e do desempenho do microsserviço, incluindo a compreensão de itens como escalas de crescimento qualitativo e quantitativo, eficiência de hardware, identificação de requisitos e gargalos de recursos, percepção e planejamento de capacidade, o tratamento escalável de tráfego, o dimensionamento das dependências, o tratamento e o processamento de tarefas e o armazenamento escalável de dados.

Princípios de escalabilidade e desempenho de microsserviços

A eficiência é de suma importância para a arquitetura de sistemas distribuídos de larga escala do mundo real, e os ecossistemas de microsserviços não são exceção a esta regra. É fácil quantificar a eficiência de um único sistema (como uma aplicação monolítica), mas é incrivelmente difícil avaliar a eficiência e alcançar maior eficiência em um grande ecossistema de microsserviços, no qual as tarefas são fragmentadas entre centenas (se não milhares) de pequenos serviços. Ela também é condicionada pela leis da arquitetura de computadores e de sistemas distribuídos, que colocam limites na eficiência de sistemas distribuídos complexos e de larga escala: quanto mais distribuído o seu sistema e maior o número de microsserviços em operação dentro desse sistema, a eficiência de um microsserviço fará menos diferença no sistema como um todo. A padronização dos princípios que aumentarão a eficiência geral se

torna uma necessidade. Dois de nossos padrões de disponibilidade de produção – *escalabilidade* e *desempenho* – ajudam a alcançar esta eficiência geral e aumentar a disponibilidade do ecossistema de microsserviços.

Escalabilidade e desempenho estão especialmente interligados graças aos efeitos que têm sobre a eficiência de cada microsserviço e do ecossistema como um todo. Como vimos no Capítulo 1, *Microsserviços*, para construir uma aplicação escalável, o projeto precisa levar em consideração a simultaneidade e o particionamento: a simultaneidade permite que cada tarefa seja dividida em partes menores, enquanto o particionamento é essencial para permitir que essas partes menores sejam processadas em paralelo. Portanto, enquanto a *escalabilidade* está relacionada à forma como dividimos e conquistamos o processamento de tarefas, o *desempenho* é a medida da eficiência da aplicação que processa essas tarefas.

Em um ecossistema de microsserviços crescente e em expansão, no qual o tráfego aumenta constantemente, cada microsserviço precisa ser capaz de crescer com todo o sistema sem sofrer de problemas de desempenho. Para garantir que nossos microsserviços sejam escaláveis e de alto desempenho, precisamos estabelecer condições para cada um deles. Precisamos entender sua *escala de crescimento*, tanto quantitativo quanto qualitativo, para nos prepararmos para o crescimento esperado. Precisamos usar nossos *recursos de hardware eficientemente*, estarmos cientes dos *requisitos e gargalos de recursos* e efetuar um adequado *planejamento de capacidade*. Precisamos garantir que as *dependências de um microsserviço cresçam* com ele. Precisamos *gerenciar o tráfego* de forma escalável e com alto desempenho. Precisamos *tratar e processar tarefas* com alto desempenho. E por fim, porém não menos importante, precisamos *armazenar dados de forma escalável*.

Um serviço pronto para produção é escalável e de alto desempenho quando:

- suas escalas de crescimento qualitativo e quantitativo são conhecidas.
- ele usa os recursos de hardware de forma eficiente.
- seus gargalos e requisitos de recursos foram identificados.

- o planejamento de capacidade é automatizado e executado de forma programada.
- suas dependências crescem com ele.
- ele cresce com seus clientes.
- seus padrões de tráfego são compreendidos.
- o tráfego pode ser redirecionado em caso de falhas.
- ele é escrito em uma linguagem de programação que permite que ele seja escalável e apresente alto desempenho.
- ele trata e processa tarefas de maneira eficiente.
- ele trata e armazena dados de forma escalável e eficiente.

Conhecendo a escala de crescimento

Determinar *como* um microsserviço cresce (em um nível muito alto) é o primeiro passo para entender como construir e manter um microsserviço escalável. Existem dois aspectos relacionados à *escala de crescimento* de um microsserviço, e ambos desempenham papéis importantes na compreensão e no planejamento da escalabilidade de um serviço. O primeiro é a *escala de crescimento qualitativo*, que resulta da compreensão do local em que o serviço se encaixa dentro do ecossistema de microsserviços como um todo e de quais são as principais métricas de alto nível de negócio que o afetarão. O segundo é a *escala de crescimento quantitativo*, que, como o nome sugere, é uma compreensão bem definida, mensurável e quantitativa de quanto tráfego um microsserviço é capaz de tratar.

A escala de crescimento qualitativa

A tendência natural quando se tenta determinar a escala de crescimento de um microsserviço é expressar a escala de crescimento em termos de *solicitações por segundo* (RPS da sigla em inglês) ou *consultas por segundo* (QPS da sigla em inglês) que o serviço pode suportar e então prever quantas RPS/QPS serão enviadas para o serviço no futuro. O termo “solicitações por segundo” geralmente é usado quando se fala de microsserviços e “consultas por segundo” é usado quando se fala de databases ou microsserviços que retornam dados para clientes, embora em

muitos casos eles sejam intercambiáveis. Esta informação é muito importante, porém inútil sem contexto adicional – especificamente sem o contexto de onde o microsserviço se encaixa no quadro geral.

Na maioria dos casos, a informação sobre o número de RPS/QPS que um microsserviço pode suportar é determinada pelo estado do microsserviço no momento em que a escala de crescimento é calculada inicialmente: se a escala de crescimento for calculada apenas considerando os atuais níveis de tráfego e como o microsserviço trata a atual carga de tráfego, qualquer inferência sobre a quantidade de tráfego que o microsserviço poderá suportar no futuro corre o risco de ser equivocada. Existem várias abordagens para contornar este problema, incluindo o teste de carga (testar o microsserviço com cargas maiores de tráfego), que pode apresentar uma imagem mais precisa da escalabilidade do serviço, e a análise do histórico dos dados de tráfego, para ver como o nível de tráfego cresce com o tempo. Porém há algo muito importante faltando, algo que é uma propriedade inerente da arquitetura de microsserviços – especificamente, o fato de que os microsserviços não existem sozinhos, mas fazem parte de um ecossistema maior.

É aí que entra a *escala de crescimento qualitativo*. As escalas de crescimento qualitativo permitem que a escalabilidade de um serviço seja vinculada às métricas de negócio de nível superior: um microsserviço pode, por exemplo, crescer com o número de usuários, com o número de pessoas que abrem uma aplicação de telefone (“eyeballs”) ou com o número de pedidos (para um serviço de entrega de comida). Essas métricas, essas escalas de crescimento qualitativo, não estão vinculadas a um microsserviço individual, mas ao sistema ou produto como um todo. No nível do negócio, a organização terá, na maior parte do tempo, certa ideia de como estas métricas mudam com o tempo. Quando essas métricas de negócio de nível superior são comunicadas às equipes de engenharia, os desenvolvedores podem interpretá-las de acordo com seus respectivos microsserviços: se um de seus microsserviços fizer parte do fluxo de pedido de um serviço de entrega de comida, eles saberão que quaisquer métricas relacionadas ao número de pedidos esperado no futuro informarão que tipo de tráfego seu serviço deve esperar.

Quando pergunto às equipes de desenvolvimento de microsserviço se elas sabem a escala de crescimento de seu serviço, a resposta usual é: “Ele

consegue tratar *x* solicitações por segundo". Minhas perguntas seguintes são sempre voltadas a descobrir onde o serviço em questão se encaixa no produto como um todo: Quando as solicitações são feitas? É uma solicitação por viagem? Uma solicitação sempre que alguém abre a aplicação? Uma solicitação sempre que um novo usuário se inscreve para usar nosso produto? Quando essas perguntas contextuais são respondidas, a escala de crescimento fica clara – e útil. Se o número de solicitações feitas ao serviço estiver diretamente vinculado ao número de pessoas que abrem uma aplicação de smartphone, então o serviço cresce com "eyeballs" e podemos planejar a escalabilidade do serviço prevendo quantas pessoas abrirão a aplicação. Se o número de solicitações feitas ao serviço for determinado pelo número de pessoas que pedem comida em domicílio, então o serviço cresce com as entregas e podemos planejar e prever a escalabilidade de nosso serviço usando métricas de negócio de nível superior sobre quantas futuras entregas estão previstas.

Existem exceções às regras de escalas de crescimento qualitativo, e determinar uma escala adequada de crescimento qualitativo pode se tornar muito complicado quanto mais o serviço estiver nas posições inferiores da pilha. Ferramentas internas tendem a sofrer dessas complicações, e mesmo assim elas tendem a ser tão críticas para o negócio que, se não forem escaláveis, o resto da organização rapidamente enfrentará problemas de escalabilidade. Não é fácil expressar a escala de crescimento de um serviço, como monitoramento ou a plataforma de alertas, em termos de métricas de negócio (usuários, "eyeballs" etc.), portanto organizações de plataforma e/ou infraestrutura precisam determinar escalas precisas de crescimento para seus serviços em termos de seus clientes (desenvolvedores, serviços etc.) e das especificações de seus clientes. Ferramentas internas podem crescer, por exemplo, com o número de implantações, o número de serviços, o número de logs agregados, ou gigabytes de dados. As escalas de crescimento qualitativo das ferramentas são mais complicadas de calcular em consequência da dificuldade inerente em prever esses números, mas elas precisam ser tão diretas e previsíveis quanto as escalas de crescimento dos microsserviços que estão mais próximos do topo da pilha.

Escala de crescimento quantitativo

A segunda parte de conhecer a escala de crescimento é determinar seus aspectos quantitativos, e é quando RPS/QPS e métricas semelhantes entram na jogada. Para determinar a *escala de crescimento quantitativo*, precisamos abordar nossos microsserviços levando em conta a escala de crescimento qualitativo: a escala de crescimento quantitativo é definida traduzindo-se a escala de crescimento qualitativo em uma quantidade mensurável. Por exemplo, se a escala de crescimento qualitativo de nosso microsserviço for medida em “eyeballs” (por exemplo, quantas pessoas abrem uma aplicação de telefone) e cada “eyeball” resultar em duas solicitações para nosso microsserviço e uma transação de database, então nossa escala de crescimento quantitativo será medida em termos de solicitações e transações, resultando em solicitações por segundo e transações por segundo como as duas principais quantidades que determinam nossa escalabilidade.

A importância de escolher escalas de crescimento qualitativo e quantitativo precisas não pode ser enfatizada o suficiente. Como veremos em breve, a escala de crescimento será usada ao se fazer previsões sobre os custos operacionais, as necessidades de hardware e as limitações do serviço.

Uso eficiente de recursos

Quando consideramos a escalabilidade de sistemas distribuídos de grande porte como ecossistemas de microsserviços, uma das abstrações mais úteis que podemos fazer é tratar as propriedades de nossos sistemas de hardware e infraestrutura como *recursos*. CPU, memória, armazenamento de dados e rede são semelhantes a recursos no mundo natural: são finitos, são objetos físicos do mundo real e precisam ser distribuídos e compartilhados entre vários importantes players no ecossistema. Como discutimos brevemente em “Desafios organizacionais”, recursos de hardware são caros, valiosos e às vezes raros, o que leva a uma feroz competição por recursos dentro do ecossistema de microsserviços.

O desafio organizacional da alocação e da distribuição de recursos pode ser amenizado se fornecermos aos microsserviços que são críticos ao negócio uma parte maior dos recursos. As necessidades por recursos podem ser priorizadas se classificarmos os vários microsserviços do ecossistema de acordo com sua importância e seu valor para o negócio

como um todo: se os recursos forem escassos dentro do ecossistema, os serviços mais críticos para o negócio podem receber uma prioridade maior em relação à alocação de recursos.

O desafio técnico da alocação e distribuição de recursos apresenta alguma dificuldade, pois muitas decisões precisam ser tomadas sobre a primeira camada (a camada de hardware) do ecossistema de microsserviços. Os microsserviços podem receber hardware dedicado de modo que apenas um serviço seja executado em cada servidor, mas isso pode ser muito caro e representar um uso ineficiente dos recursos de hardware. Muitas organizações de engenharia de software optam por compartilhar hardware entre vários microsserviços, e cada servidor executará diversos serviços diferentes – uma prática que é, na maioria dos casos, um uso mais eficiente dos recursos de hardware.


Os perigos de recursos de hardware compartilhados

Embora executar muitos microsserviços diferentes em uma máquina (isto é, compartilhar máquinas entre os microsserviços) geralmente represente um uso mais eficiente dos recursos de hardware, é preciso tomar cuidado para garantir que os microsserviços estejam suficientemente isolados e não comprometam o desempenho, a eficiência ou a disponibilidade de seus serviços adjacentes. O empacotamento dentro de um "contêiner" (usando Docker) junto ao isolamento de recursos pode ajudar a evitar que os microsserviços sejam prejudicados por vizinhos malcomportados.

Uma das maneiras mais eficientes de alocar e distribuir recursos de hardware por um ecossistema de microsserviços é abstrair totalmente a noção de um servidor e substituí-la por recursos de hardware usando tecnologias de abstração de recursos como Apache Mesos. Usar este nível de abstração de recursos permite que os recursos sejam alocados dinamicamente, eliminando muitas das armadilhas associadas à alocação e distribuição de recursos em sistemas distribuídos de grande escala como ecossistemas de microsserviços.

Percepção de recursos

Antes que os recursos de hardware possam ser alocados e distribuídos de forma eficiente entre os microsserviços dentro do ecossistema de microsserviços, é importante identificar os *requisitos de recursos* e

gargalos de recursos de cada microsserviço. Requisitos de recursos são os recursos específicos (CPU, RAM etc.) de que cada microsserviço necessita; identificá-los é essencial para executar um serviço escalável. Os gargalos de recursos são as limitações de escalabilidade e desempenho de cada microsserviço individual que dependem das características de seus recursos.

Requisitos de recursos

Os *requisitos de recursos* de um microsserviço são os recursos de hardware de que o microsserviço precisa para ser executado adequadamente, para processar as tarefas com eficiência e para escalar verticalmente e/ou horizontalmente. Os dois recursos de hardware mais importantes e relevantes tendem a ser, como era de esperar, CPU e RAM (em ambientes "multithread", as threads passam a ser o terceiro recurso importante). Determinar os requisitos de recursos de um microsserviço implica quantificar a CPU e a RAM de que *uma instância* do serviço precisa para ser executada. Isso é essencial para abstração de recursos, alocação e distribuição de recursos e para determinar a escalabilidade geral e o desempenho do microsserviço.


Identificando requisitos adicionais de recursos

Se por um lado a CPU e a RAM são os dois requisitos de recursos mais comuns, é importante ficar de olho em outros recursos de que um microsserviço pode precisar dentro do ecossistema. Eles podem ser recursos de hardware, como conexões a databases, ou recursos de plataforma de aplicação, como quotas de logging. Perceber as necessidades de um microsserviço específico ajuda muito a melhorar a escalabilidade e o desempenho.

Calcular os requisitos de recursos específicos de um microsserviço pode ser um processo complicado e demorado, pois há muitos fatores relevantes. O essencial, como mencionei anteriormente, é determinar quais são os requisitos para uma única *instância* do serviço. A forma mais eficaz e eficiente de escalar nosso serviço é dimensioná-lo horizontalmente: se nosso tráfego está prestes a crescer, precisamos adicionar mais alguns servidores e implantar nosso serviço neles. Para saber quantos servidores adicionais são necessários, precisamos saber como se comporta nosso serviço quando é executado em apenas um servidor: quanto tráfego ele consegue tratar? Quanta CPU ele utiliza? Quanta memória? Esses números

irão dizer exatamente quais são os requisitos de recursos de nosso microsserviço.

Gargalos de recursos

Podemos descobrir e quantificar as limitações de desempenho e escalabilidade de nossos microsserviços identificando os *gargalos de recursos*. Um gargalo de recurso é qualquer coisa inerente ao modo como o microsserviço utiliza seus recursos e que limita a escalabilidade da aplicação. Pode ser um gargalo de infraestrutura ou algo dentro da arquitetura do serviço que impeça que ele seja escalável. Por exemplo, o número de conexões abertas com um database de que um microsserviço precisa pode ser um gargalo se ele se aproximar do limite de conexões do database. Outro exemplo de um gargalo comum de recurso é quando os microsserviços precisam ser escalados verticalmente (em vez de escalados horizontalmente, quando mais instâncias/hardwares são adicionados) quando eles experimentam um aumento de tráfego: se a única maneira de escalar um microsserviço é aumentar os recursos de cada instância (mais CPU, mais memória), então os dois princípios de escalabilidade (simultaneidade e particionamento) são abandonados.

Alguns gargalos de recursos são fáceis de identificar. Se seu microsserviço só pode ser escalado para atender ao tráfego crescente por meio da implantação em máquinas com mais CPU e memória, então você tem um gargalo de escalabilidade e precisa refatorar o microsserviço de modo que ele possa ser escalado horizontalmente em vez de verticalmente, usando simultaneidade e particionamento como princípios orientadores.

 As armadilhas do escalamento vertical

O escalamento vertical não é uma forma sustentável ou escalável de projetar microsserviços. Ele pode parecer funcionar bem em situações em que cada microsserviço tem um hardware dedicado, mas não funciona bem com as novas tecnologias de abstração e isolamento de hardware que são usadas na área de tecnologia atualmente, como Docker e Apache Mesos. Sempre otimize pensando em simultaneidade e particionamento se você quiser construir uma aplicação escalável.

Outros gargalos de recurso não são tão óbvios, e a melhor maneira de descobri-los é submeter o serviço a exaustivos testes de carga. Abordaremos os testes de carga em mais detalhes em “Teste de

resiliência”.

Planejamento de capacidade

Um dos mais importantes requisitos para construir um microsserviço escalável é garantir que ele terá acesso aos recursos de hardware necessários à medida que crescer. Usar recursos com eficiência, planejar para o crescimento e projetar do zero um microsserviço para perfeita eficiência e escalabilidade rapidamente se tornam inúteis se recursos de hardware não estiverem disponíveis quando o microsserviço precisar atender a um tráfego de produção maior. Esse desafio é especialmente relevante para microsserviços que são otimizados para escalabilidade horizontal.

Além dos desafios técnicos que acompanham esse potencial problema, empresas de engenharia frequentemente enfrentam questões mais complicadas no âmbito organizacional e de relevância para o negócio: recursos de hardware custam caro, equipes de desenvolvimento individuais e de negócios precisam cumprir metas orçamentárias, e esses orçamentos (que tendem a incluir hardware) precisam ser planejados com antecedência. Para garantir que os microsserviços possam ser escalados adequadamente quando o tráfego aumentar é possível fazer um *planejamento de capacidade* programado. Os princípios do planejamento de capacidade são bem diretos: determinar as necessidades de hardware de cada microsserviço com antecedência, incorporar as necessidades ao orçamento e garantir que o hardware necessário seja reservado.

Para definir as necessidades de hardware de cada serviço, podemos usar as escalas de crescimento (tanto quantitativo quanto qualitativo), as principais métricas de negócio e as previsões de tráfego, os gargalos e requisitos de recursos conhecidos e o histórico de dados sobre o tráfego do microsserviço. É aí que as escalas de crescimento qualitativo e quantitativo são especialmente úteis, porque nos permitem descobrir precisamente como o comportamento de escalabilidade de nossos microsserviços se relaciona com as previsões de negócio de alto nível. Por exemplo, se soubermos que (1) nosso microsserviço cresce com visitantes únicos ao produto como um todo, (2) cada visitante único corresponderá a certo número de solicitações por segundo feitas ao microsserviço e (3) a empresa poderá prever que o produto receberá 20 mil novos visitantes

únicos no trimestre seguinte, então saberemos exatamente quais serão nossas necessidades de capacidade para o trimestre seguinte.

Isso precisa ser incorporado ao orçamento de cada equipe de desenvolvimento, cada organização de engenharia de software e cada empresa. Realizar este exercício de forma programada antes de determinar o orçamento pode ajudar as organizações de engenharia de software a garantir que os recursos de hardware nunca estejam indisponíveis simplesmente porque o orçamento dos recursos não foi concluído ou elaborado. O importante aqui (tanto das perspectivas da engenharia quanto de negócio) é reconhecer o custo de um planejamento inadequado de capacidade: microsserviços que não podem ser escalados corretamente devido à escassez de hardware resultam em uma menor disponibilidade dentro de todo o ecossistema, o que leva a interrupções, o que custa dinheiro à empresa.

Tempo de espera para solicitações de novo hardware

Um problema potencial que normalmente é ignorado pelas equipes de desenvolvimento durante a fase de planejamento de capacidade é que o hardware necessário para o microsserviço pode não existir no momento do planejamento e talvez ele precise ser comprado, instalado e configurado antes de qualquer microsserviço poder usá-lo. Antes de programar um planejamento de capacidade, não se esqueça de descobrir o tempo exato de espera necessário para adquirir novo hardware a fim de evitar longas interrupções em momentos críticos e alguns atrasos no processo.

Uma vez garantidos os recursos de hardware dedicados a cada microsserviço, o planejamento de capacidade estará concluído. Determinar quando e como alocar o hardware depois da fase de planejamento é, claro, responsabilidade de cada organização de engenharia de software e de suas equipes de desenvolvimento, infraestrutura e operações.

O planejamento de capacidade pode ser uma tarefa realmente difícil e manual. Como a maioria das tarefas manuais dentro da engenharia, esta introduz novos modos de falha: cálculos manuais podem estar incorretos e mesmo uma pequena interrupção pode se mostrar desastrosa para serviços críticos para o negócio. Automatizar a maior parte do processo de planejamento de capacidade, separando-o das equipes de desenvolvimento e operações, reduz a chance de potenciais erros e falhas, e uma ótima maneira de conseguir isso é construir e executar uma ferramenta de

planejamento de capacidade do tipo autosserviço dentro da camada da plataforma de aplicação do ecossistema de microsserviços.

Escalamento de dependências

A escalabilidade das dependências de um microsserviço pode apresentar um problema de escalabilidade próprio. Um microsserviço que foi projetado, construído e executado para ser perfeitamente escalável em todos os aspectos ainda enfrentará desafios de escalabilidade se suas dependências não puderem escalar com ele. Se apenas uma dependência crítica for incapaz de escalar com seus clientes, então toda a cadeia de dependências sofrerá. Garantir que todas as dependências irão escalar com o crescimento esperado de um microsserviço é essencial para construir serviços prontos para produção.

Este desafio é relevante para todo microsserviço individual e toda parte do ecossistema de microsserviços, o que significa que as equipes de microsserviço também precisam garantir que seu serviço não seja um gargalo de escalabilidade para seus clientes. Em outras palavras, uma complexidade adicional é introduzida pelo restante do ecossistema de microsserviços. É preciso estar preparado para o inevitável tráfego adicional e o crescimento das necessidades dos clientes de um microsserviço.

 Escalas de crescimento qualitativo e escalabilidade das dependências

Ao lidar com as incrivelmente complexas cadeias de dependência, garantir que todas as equipes de microsserviço vinculem a escalabilidade de seus serviços às métricas de negócio de alto nível (usando a escala de crescimento qualitativo) fará com que todos os serviços estejam adequadamente preparados para o crescimento esperado, mesmo quando a comunicação entre equipes se tornar difícil.

O problema da escalabilidade de dependências é um argumento especialmente forte para a implementação de padrões de escalabilidade e desempenho em todo o ecossistema de microsserviços. A maioria dos microsserviços não existe isoladamente. Quase todo microsserviço é uma pequena parte de cadeias de dependência intricadas e entrelaçadas. Na maioria dos casos, escalar o produto, a organização e o ecossistema de forma eficaz requer que cada peça do sistema escale junto com o restante.

Ter um pequeno número de microsserviços supereficientes, de alto desempenho e escaláveis em um sistema no qual o resto dos serviços não cumpre os mesmos padrões torna a eficiência dos serviços padronizados completamente discutível.

Além da padronização em todo o ecossistema e da exigência para que cada equipe de desenvolvimento de microsserviço atenda aos altos padrões de escalabilidade, é importante que as equipes de desenvolvimento trabalhem juntas por meio das interfaces entre microsserviços para garantir que cada cadeia de dependência escale junto. As equipes de desenvolvimento responsáveis por quaisquer dependências de um microsserviço precisam ser alertadas quando forem esperados crescimentos de tráfego. A comunicação e a colaboração entre equipes são essenciais aqui: comunicar-se regularmente com clientes e dependências sobre os requisitos, o status e os gargalos de escalabilidade do serviço pode ajudar a garantir que quaisquer serviços que dependam uns dos outros estejam preparados para o crescimento e cientes sobre qualquer potencial gargalo de escalabilidade. Uma estratégia que usei para ajudar as equipes a conseguir isso é organizar reuniões de revisão de arquitetura e escalabilidade com equipes cujos serviços dependam uns dos outros. Nessas reuniões, cobrimos a arquitetura de cada serviço e suas limitações de escalabilidade, e então discutimos o que precisava ser feito para escalar todo o conjunto de serviços.

Gerenciamento de tráfego

À medida que os serviços escalam e o número de solicitações que cada serviço precisa tratar cresce, um serviço escalável e de alto desempenho também precisa tratar o tráfego de forma inteligente. Existem vários aspectos relacionados ao gerenciamento de tráfego de forma escalável e de alto desempenho: primeiro, a escala de crescimento (quantitativo e qualitativo) precisa ser usada para prever futuros aumentos (ou reduções) de tráfego; segundo, é preciso compreender bem e estar preparado para os padrões de tráfego; e terceiro, os microsserviços precisam ser capazes de tratar de forma inteligente os aumentos de tráfego, assim como picos repentinos de tráfego.

Já tratamos do primeiro aspecto no início deste capítulo: entender as escalas de crescimento (tanto quantitativo quanto qualitativo) de um

microsserviço permite-nos entender as atuais cargas de tráfego sobre o serviço e também nos prepararmos para um futuro crescimento de tráfego.

Entender os atuais padrões de tráfego ajuda na interação com o serviço no nível mais baixo de várias formas interessantes. Quando os padrões de tráfego são claramente identificados, tanto em termos de solicitações por segundo enviadas para o serviço ao longo do tempo quanto de todas as principais métricas (veja o Capítulo 6, *Monitoramento*, para mais informações sobre as principais métricas), as mudanças no serviço, os downtimes operacionais e as implantações podem ser programadas para evitar horários de pico de tráfego, reduzindo as possíveis futuras interrupções se um erro for implantado e o potencial downtime se o microsserviço for reiniciado ao experimentar uma carga de pico de tráfego. Monitorar atentamente o tráfego em vista de padrões de tráfego e ajustar os limites de monitoramento cuidadosamente de acordo com os padrões de tráfego do microsserviço em questão podem ajudar a detectar quaisquer problemas e incidentes rapidamente, antes que eles causem uma interrupção ou levem a uma menor disponibilidade (os princípios de monitoramento pronto para produção são tratados com mais detalhes no Capítulo 6, *Monitoramento*).

Quando podemos prever o futuro crescimento de tráfego e entender os padrões atuais e anteriores de tráfego suficiente bem para saber como os padrões mudarão com o crescimento esperado, é possível executar testes de carga sobre nossos serviços para garantir que eles se comportarão conforme esperado com cargas de tráfego mais pesadas. Os detalhes do teste de carga são tratados em “Teste de resiliência”.

O terceiro aspecto do gerenciamento de tráfego é quando as coisas se complicam. O modo como um microsserviço trata o tráfego deve ser escalável, o que significa que ele deve estar preparado para drásticas mudanças de tráfego, especialmente picos de tráfego, tratá-las com cuidado e evitar que derrubem o serviço completamente. É mais fácil falar do que fazer, porque mesmo os microsserviços mais bem monitorados, escaláveis e de alto desempenho podem sofrer problemas de monitoramento, logging e outros se o tráfego crescer repentinamente. É preciso estar preparado para esses tipos de picos no nível de infraestrutura, dentro de todos os sistemas de monitoramento e logging, e dentro das equipes de desenvolvimento como parte do pacote de teste de resiliência

do serviço.

Há um aspecto adicional que gostaria de mencionar e que está relacionado ao gerenciamento de tráfego entre e através de vários locais. Muitos ecossistemas de microsserviços não serão implantados em apenas um local, um datacenter ou uma cidade, mas em vários datacenters espalhados pelo país (ou até mesmo pelo mundo). Não é incomum que os próprios datacenters sofram interrupções de grande escala, e quando isso acontece todo o ecossistema de microsserviços pode (e geralmente irá) cair junto com o datacenter. Distribuir e rotear tráfego adequadamente entre datacenters é responsabilidade do nível de infraestrutura (em particular, da camada de comunicação) do ecossistema de microsserviços, mas cada microsserviço precisa estar preparado para redirecionar o tráfego de um datacenter para outro sem que o serviço sofra qualquer redução de disponibilidade.

Tratamento e processamento de tarefas

Todo microsserviço no ecossistema de microsserviços precisará processar tarefas de algum tipo. Isto é, todo microsserviço receberá solicitações de serviços-cliente upstream que precisam de alguma informação do microsserviço ou precisam que o microsserviço calcule ou processe algo e então retorne informações sobre esse cálculo ou processo, e então o microsserviço precisa atender a essa solicitação (geralmente se comunicando com serviços downstream, além de executar alguma atividade própria) e retornar qualquer informação solicitada ou resposta ao cliente que enviou a solicitação.

Limitações da linguagem de programação

Os microsserviços podem processar tarefas e desempenhar o papel que se espera deles de muitas maneiras, e as formas como eles irão executar seus cálculos, interagir com serviços downstream e processar as várias tarefas dependerá da linguagem na qual o serviço é escrito e, consequentemente, da arquitetura do serviço (o que, de muitas formas, é determinado pela linguagem). Por exemplo, um microsserviço escrito em Python tem muitas formas de processar múltiplas tarefas, algumas das quais exigem o uso de frameworks assíncronos (como Tornado), e outras podem utilizar tecnologias de troca de mensagens como RabbitMQ e Celery para

processar as tarefas de modo eficiente. Por esses motivos, a capacidade de um microsserviço de tratar e processar tarefas de forma escalável e com alto desempenho é ditada em parte pela escolha da linguagem.

Tome cuidado com as limitações de escalabilidade e desempenho das linguagens de programação

Muitas linguagens de programação não são otimizadas para os requisitos de escalabilidade e desempenho da arquitetura de microsserviços ou não contêm frameworks escaláveis e de alto desempenho que permitam que os microsserviços processem tarefas de modo eficiente.

Devido às limitações introduzidas pela escolha da linguagem, quando se trata da capacidade de um microsserviço de processar tarefas eficientemente, a escolha da linguagem se torna extremamente importante na arquitetura de microsserviços. Para muitos desenvolvedores, um dos argumentos para a adoção da arquitetura de microsserviços é a capacidade de escrever um microsserviço em qualquer linguagem, e isso em geral é verdade, mas com uma ressalva: é preciso levar em consideração as limitações da linguagem de programação, e qualquer escolha de linguagem deve ser determinada não pelo fato de ela estar na moda ou ser divertida (ou mesmo se ela for a linguagem mais comum com a qual a equipe de desenvolvimento está familiarizada), mas considerando as limitações de escalabilidade e desempenho de cada potencial linguagem como fatores decisivos. Não existe a “melhor” linguagem para escrever um microsserviço, mas *existem* linguagens que são mais adequadas do que outras para certos tipos de microsserviços.

Tratando das solicitações e processando tarefas de modo eficiente

Escolha da linguagem à parte, a padronização da disponibilidade de produção requer que cada microsserviço seja escalável e de alto desempenho, o que significa que os microsserviços precisam ser capazes de tratar e processar um grande número de tarefas ao mesmo tempo, tratar e processar essas tarefas de modo eficiente e estar preparados para um aumento do número de tarefas e solicitações no futuro. Com isso em mente, as equipes de desenvolvimento devem ser capazes de responder a três perguntas básicas sobre seus microsserviços: como seu microsserviço processa tarefas, o quão eficientemente seu microsserviço processa essas

tarefas e como seu microsserviço irá se comportar à medida que o número de solicitações aumentar.

Para garantir a escalabilidade e o desempenho, os microsserviços precisam processar tarefas de modo eficiente. Para fazê-lo, eles precisam adotar a simultaneidade e o particionamento. A simultaneidade requer que o serviço não tenha um único processo que faça todo o trabalho: um processo deste tipo escolhe uma tarefa por vez, conclui as etapas em uma ordem específica e então passa para a próxima tarefa, o que é uma maneira relativamente ineficiente de processar tarefas. Em vez de projetar nosso serviço para usar um único processo, nós podemos introduzir a simultaneidade de modo que cada tarefa seja dividida em partes menores.

 Escreva microsserviços em linguagens de programação que sejam otimizadas para simultaneidade e particionamento

Algumas linguagens são mais adequadas do que outras para um tratamento e processamento mais eficiente (simultâneo e particionado) de tarefas. Ao escrever um novo microsserviço, certifique-se de que a linguagem usada não introduza limitações de escalabilidade e desempenho. Os microsserviços que já são escritos em linguagens com limitações de eficiência podem (e devem) ser reescritos em linguagens mais adequadas, o que é uma tarefa demorada, mas incrivelmente recompensadora, que pode melhorar drasticamente a escalabilidade e o desempenho. Por exemplo, se você estiver otimizando para simultaneidade e particionamento, e quiser usar um framework assíncrono para ajudá-lo nisso, escrever seu serviço em Python (em vez de escrevê-lo na linguagem C++, Java ou Go – três linguagens construídas para simultaneidade e particionamento) irá introduzir muitos gargalos de escalabilidade e desempenho que serão difíceis de mitigar.

Dividindo essas tarefas em partes menores, podemos processá-las de modo mais eficiente usando particionamento – em que cada tarefa não é apenas dividida em pequenas partes, mas também pode ser processada em paralelo. Se tivermos um grande número de pequenas tarefas, podemos processá-las todas ao mesmo tempo enviando-as para um conjunto de workers que podem processá-las em paralelo. Se precisarmos processar mais tarefas, podemos facilmente acompanhar a maior demanda adicionando outros workers para processar as novas tarefas sem afetar a eficiência de nosso sistema. Juntos, a simultaneidade e o particionamento ajudam a garantir que nosso microsserviço seja otimizado tanto para escalabilidade quanto para desempenho.

Armazenamento escalável de dados

Os microsserviços precisam *tratar os dados de modo escalável e com alto desempenho*. A maneira como um microsserviço armazena e trata os dados pode facilmente se tornar a limitação mais significativa que impede sua escalabilidade e desempenho: escolher o database incorreto, o schema incorreto ou um database que não suporta locação de teste (“test tenancy”) pode acabar comprometendo a disponibilidade geral de um microsserviço. Escolher o database correto para um microsserviço é um tópico que, como todos os outros tópicos abordados neste livro, é incrivelmente complexo e só será tratado superficialmente neste capítulo. Nas próximas seções, observaremos os vários aspectos a ser considerados ao se escolher databases em um ecossistema de microsserviços, e então alguns desafios de database que são específicos para a arquitetura de microsserviços.

Escolha do database em ecossistemas de microsserviços

Construir, executar e manter databases em grandes ecossistemas de microsserviços não é uma tarefa fácil. Algumas empresas que adotam a arquitetura de microsserviços optam por permitir que as equipes de desenvolvimento escolham, construam e mantenham suas próprios databases, enquanto outras decidem por, pelo menos, um database que funcione para a maioria dos microsserviços nesta empresa e criam uma equipe separada para executar e manter os databases para que os desenvolvedores se concentrem apenas em seus próprios microsserviços.

Se pensarmos na arquitetura de microsserviços como sendo composta de quatro camadas separadas (veja “Arquitetura de microsserviços” para mais detalhes) e reconhecermos que, graças à Lei Reversa de Conway, as organizações de engenharia de software das empresas que adotam a arquitetura de microsserviços irão espelhar a arquitetura de seu produto, então podemos ver de quem é a responsabilidade por escolher os databases adequados, construí-los, executá-los e mantê-los: ou na camada de plataforma de aplicação, o que permitiria que os databases fossem fornecidos como um serviço para as equipes de microsserviços, ou na camada de microsserviço, onde o database usado por um microsserviço é considerada parte do serviço. Já vi ambas as configurações em funcionamento em várias empresas, e algumas funcionam melhor que outras. Também percebi que uma abordagem funciona particularmente

bem: oferecer databases como um serviço dentro da camada de plataforma de aplicação e então permitir que equipes individuais de desenvolvimento de microsserviços executem seu próprio database se os databases oferecidos como parte da plataforma de aplicação não atenderem às suas necessidades específicas.

Os tipos mais comuns de database são os *relacionais* (SQL, MySQL) e os *NoSQL* (Cassandra, Vertica, MongoDB e do tipo chave-valor como Dynamo, Redis e Riak). Escolher entre um database relacional e um database NoSQL, e então escolher o adequado database específico para as necessidades de um microsserviço depende das respostas a várias perguntas:

- Qual o requisito de transações por segundo de cada microsserviço?
- Que tipo de dados cada microsserviço precisa armazenar?
- Qual o schema exigido por cada microsserviço? E com que frequência ele precisará ser alterado?
- Os microsserviços precisam de uma consistência forte ou eventual?
- Os microsserviços fazem uso intenso de operações de leitura, de escrita ou de ambas?
- O database precisa ser escalado horizontalmente ou verticalmente?

Independentemente do fato de um database ser mantido como parte de uma plataforma de aplicação ou pela equipe individual de desenvolvimento de cada microsserviço, a escolha do database deve ser motivada pelas respostas às perguntas acima. Por exemplo, se o database em questão precisa ser escalado horizontalmente, ou se leituras e escritas devem ocorrer em paralelo, então um database NoSQL deve ser escolhido, já que databases relacionais têm dificuldades com escalamento horizontal e operações paralelas de leitura e escrita.

Desafios de databases na arquitetura de microsserviços

Existem vários desafios relacionados a databases que são específicos da arquitetura de microsserviços. Quando databases são compartilhados entre microsserviços, a competição por recursos entra em jogo e alguns microsserviços podem utilizar mais do que sua parcela justa do armazenamento disponível. Os engenheiros que estão construindo e mantendo databases compartilhados devem projetar suas soluções de

armazenamento de dados para que os databases possam ser facilmente escalados se qualquer um dos microsserviços “inquilinos” precisar de espaço adicional ou estiver correndo o risco de usar todo o espaço disponível.

 Cuidado com as conexões ao database

Alguns databases têm limitações estritas em relação ao número de conexões que podem ser abertas simultaneamente. Certifique-se de que todas as conexões sejam fechadas adequadamente para evitar comprometer tanto a disponibilidade de um serviço quanto a disponibilidade do database para todos os microsserviços que o utilizam.

Outro desafio que microsserviços enfrentam com frequência, especialmente depois de terem construído e padronizado ciclos de desenvolvimento e pipelines de implantação estáveis e confiáveis, é o tratamento de dados de teste dos testes fim a fim, dos testes de carga e de quaisquer escritas de teste feitas na fase de staging. Como mencionado em “O pipeline de deployment”, a fase de staging do pipeline de deployment requer a leitura e/ou escrita em databases. Se o modelo de staging total tiver sido implementado, então a fase de staging terá sua próprio database separado para teste e staging, mas o staging parcial requer acesso de leitura e escrita aos servidores de produção, portanto é preciso tomar muito cuidado para garantir que os dados de teste sejam tratados adequadamente: eles precisam ser claramente marcados como dados de teste (um processo conhecido como *locação de teste*), e então todos os dados de teste precisam ser apagados a intervalos regulares.

Avalie seu microsserviço

Agora que você tem um melhor entendimento da escalabilidade e do desempenho, use a seguinte lista de perguntas para avaliar a disponibilidade de produção de seu(s) microsserviço(s) e do ecossistema de microsserviços. As perguntas são organizadas por tópico e correspondem às seções dentro deste capítulo.

Conhecer a escala de crescimento

- Qual a escala de crescimento qualitativo do microsserviço?
- Qual a escala de crescimento quantitativo do microsserviço?

Uso eficiente de recursos

- O microsserviço está sendo executado em hardware dedicado ou compartilhado?
- Estão sendo usadas tecnologias de abstração e alocação de recursos?

Percepção de recursos

- Quais os requisitos de recursos do microsserviço (CPU, RAM etc.)?
- Quanto tráfego uma instância do microsserviço consegue tratar?
- Quanta CPU uma instância do microsserviço exige?
- Quanta memória uma instância do microsserviço exige?
- Existem outros requisitos de recurso que são específicos deste microsserviço?
- Quais os gargalos de recursos deste microsserviço?
- Este microsserviço precisa ser escalado verticalmente, horizontalmente ou ambos?

Planejamento de capacidade

- O planejamento de capacidade é executado de forma programada?
- Qual o tempo de espera pelo novo hardware?
- Qual a frequência das solicitações de hardware?
- Alguns microsserviços têm prioridade quando as solicitações de hardware são feitas?
- O planejamento de capacidade é automatizado ou manual?

Dimensionamento das dependências

- Quais as dependências deste microsserviço?
- As dependências são escaláveis e de alto desempenho?
- As dependências escalarão com o crescimento esperado deste microsserviço?
- Os donos das dependências estão preparados para o crescimento esperado deste microsserviço?

Gerenciamento de tráfego

- Os padrões de tráfego do microsserviço são bem entendidos?
- As mudanças no serviço são programadas de acordo com os padrões de tráfego?
- As mudanças drásticas nos padrões de tráfego (especialmente picos de tráfego) são tratadas de forma cuidadosa e adequada?
- O tráfego pode ser automaticamente roteado para outros datacenters em caso de falha?

Tratamento e processamento de tarefas

- O microsserviço está escrito em uma linguagem de programação que permite que o serviço seja escalável e de alto desempenho?
- Existem limitações de escalabilidade ou desempenho na forma como o microsserviço trata as solicitações?
- Existem limitações de escalabilidade ou desempenho na forma como o microsserviço processa tarefas?
- Os desenvolvedores na equipe do microsserviço entendem como seu serviço processa tarefas, o quão eficientemente ele processa essas tarefas e como o serviço irá se comportar à medida que o número de tarefas e solicitações aumentar?

Armazenamento escalável de dados

- Este microsserviço trata dados de forma escalável e com alto desempenho?
- Que tipos de dados este microsserviço precisa armazenar?
- Qual o schema necessário para seus dados?
- Quantas transações são necessárias e/ou feitas por segundo?
- Este microsserviço precisa de um desempenho maior de escrita ou leitura?
- Ele faz uso intensivo de operações de leitura, escrita ou ambos?
- O database deste serviço é escalado horizontalmente ou verticalmente? Ele é replicado ou particionado?
- Este microsserviço usa um database dedicado ou compartilhado?

- Como o serviço trata e/ou armazena dados de teste?

CAPÍTULO 5

Tolerância a falhas e preparação para catástrofes

Um microsserviço pronto para produção é tolerante a falhas e preparado para qualquer catástrofe. Microsserviços falharão, eles falharão com frequência, e qualquer potencial cenário de falha pode e irá ocorrer em algum momento da vida útil de um microsserviço. Garantir a disponibilidade em todo o ecossistema de microsserviço requer um cuidadoso planejamento contra falhas, uma preparação para catástrofes e ativamente forçar o microsserviço a falhar em tempo real para garantir que ele possa se recuperar das falhas com elegância.

Este capítulo cobre itens como evitar pontos únicos de falha, cenários comuns de catástrofes e falhas, detecção e reparo de falhas, implementar diferentes tipos de teste de resiliência e maneiras de lidar com incidentes e interrupções no âmbito organizacional quando ocorrerem falhas.

Princípios de construção de microsserviços tolerantes a falhas

A realidade de construir sistemas distribuídos de grande escala é que componentes individuais podem falhar, eles irão falhar e irão falhar com frequência. Nenhum ecossistema de microsserviços é exceção a essa regra. Qualquer possível cenário de falha pode e irá ocorrer em algum momento da vida útil de um microsserviço, e essas falhas podem se tornar piores em virtude das complexas cadeias de dependência dentro de ecossistemas de microsserviços: se um serviço na cadeia de dependência falha, todos os clientes upstream sofrem e a disponibilidade fim a fim de todo o sistema é comprometida.

A única maneira de mitigar falhas catastróficas e evitar comprometer a disponibilidade de todo o sistema é exigir que cada microsserviço dentro

do ecossistema seja *tolerante a falhas* e *preparado para qualquer catástrofe*.

O primeiro passo na construção de um microsserviço tolerante a falhas e preparado para catástrofes é eliminar *pontos únicos de falha*. Nunca deve existir uma parte do ecossistema cuja falha possa parar todo o sistema, e também não deve haver qualquer parte individual dentro da arquitetura de um microsserviço que derrube o microsserviço sempre que ela falhar. Identificar estes pontos únicos de falha, tanto dentro do microsserviço como em uma camada de abstração acima dele, pode evitar a ocorrência de falhas mais evidentes.

Identificar cenários de falha é o próximo passo. Nem toda falha ou catástrofe que acomete um microsserviço é um evidente e óbvio ponto único de falha que pode ser eliminado. Tolerância a falhas e preparação para catástrofes requerem que um microsserviço suporte tanto *falhas internas* (falhas dentro do próprio microsserviço) como *falhas externas* (falhas dentro de outras camadas do ecossistema). Desde a falha em um servidor até a falha de todo um datacenter, de um database até a fila de tarefas distribuídas de um serviço, o número de maneiras pelas quais um microsserviço pode ser derrubado pela falha de uma ou mais de suas partes é enorme, crescendo com a complexidade do próprio microsserviço e do ecossistema de microsserviços como um todo.

Uma vez eliminados os pontos únicos de falha e identificada a maioria (ou todos) dos cenários de falha, o próximo passo é testar essas falhas para ver se o microsserviço pode ou não se recuperar com elegância quando estas ocorrerem e determinar se ele é resiliente ou não. A resiliência de um serviço pode ser testada completamente por meio de *testes de código*, *testes de carga* e *testes de caos*.

Este passo é fundamental: em um complexo ecossistema de microsserviços, simplesmente eliminar a falha não é suficiente – mesmo a melhor estratégia de mitigação pode se mostrar totalmente inútil quando os componentes começam a falhar. A única maneira de construir um microsserviço realmente tolerante a falhas é forçá-lo a falhar em ambiente de produção, causando falhas de forma ativa, repetida e aleatória em cada componente que possa derrubar o sistema.

Nem todas as falhas podem ser previstas, portanto os últimos passos para construir microsserviços tolerantes a falhas e preparados para catástrofes

são de natureza organizacional. As estratégias de detecção e mitigação de falhas precisam estar funcionando e devem ser padronizadas em cada equipe de microsserviço, e qualquer nova falha que um serviço sofrer deve ser adicionada ao conjunto de testes de resiliência para garantir que ela nunca volte a ocorrer. As equipes de microsserviços também precisam ser treinadas para lidar com as falhas de modo adequado: lidar com interrupções e incidentes (independentemente de sua gravidade) deve ser padronizado em toda a organização de engenharia de software.

Um serviço pronto para produção é tolerante a falhas e preparado para qualquer catástrofe quando:

- ele não tem um ponto único de falha;
- todos os cenários de falha e as possíveis catástrofes foram identificados;
- ele é testado para resiliência por meio de testes de código, testes de carga e testes de caos;
- a detecção e a mitigação de falhas foram automatizadas;
- existem procedimentos padronizados de incidentes e interrupções dentro da equipe de desenvolvimento de microsserviços e em toda a organização.

Evitando pontos únicos de falha

O primeiro lugar para procurar possíveis cenários de falhas é dentro da arquitetura de cada microsserviço. Se há uma parte da arquitetura do serviço que possa derrubar todo o microsserviço em caso de falha, nós nos referimos a ela como um *ponto único de falha* do microsserviço. Nenhuma parte da arquitetura de um microsserviço deve ser capaz de derrubar o serviço, mas ela frequentemente derruba. Na verdade, a maioria dos microsserviços do mundo real tem não apenas um *único* ponto de falha, mas *vários* pontos de falha.

Exemplo: o broker de mensagem como um ponto único de falha

Para entender como um ponto único de falha funciona no mundo real

da produção, vamos considerar um microsserviço escrito em Python que usa uma combinação de Redis (como broker de mensagem) e Celery (como processador de tarefas) para processamento distribuído de tarefas.

Vamos supor que os workers do Celery (que estão processando as tarefas) parem por algum motivo e sejam incapazes de concluir seu trabalho. Isso não é necessariamente um ponto de falha, pois o Redis (funcionando como o broker de mensagem) pode processar novamente as tarefas quando os workers voltarem a funcionar. Enquanto os workers estiverem parados, o Redis continuará funcionando e as tarefas se acumularão na fila do Redis, aguardando para ser distribuídas aos workers do Celery assim que eles voltarem a funcionar. Este microsserviço, porém, acolhe uma *grande quantidade* de tráfego (recebendo milhares de solicitações por segundo), e as filas começam a encher, esgotando toda a capacidade da máquina Redis. Rapidamente a caixa Redis fica sem memória e você começa a perder tarefas. Isso parece ruim, mas a situação pode piorar ainda mais, pois seu hardware pode ser compartilhado entre vários microsserviços diferentes, e então todos os outros microsserviços que estiverem usando essa caixa Redis como broker de mensagem perderão todas as suas tarefas.

Este (a máquina Redis neste exemplo) é um ponto único de falha e é um exemplo do mundo real que vi muitas e muitas vezes em minha experiência trabalhando com desenvolvedores para identificar pontos únicos de falha em seus microsserviços.

É fácil identificar pontos de falha dentro de microsserviços quando eles falham realmente, e precisamos repará-los para que o microsserviço volte a funcionar. Aguardar uma falha, porém, não é a melhor abordagem se quisermos que nossos microsserviços sejam tolerantes a falhas e mantenham sua disponibilidade. Uma ótima maneira de descobrir pontos de falha antes que eles sejam responsáveis por uma interrupção é executar revisões de arquitetura com as equipes de desenvolvimento de microsserviço, pedir que os desenvolvedores em cada equipe desenhem a arquitetura de seu microsserviço em um quadro branco e então analisar a arquitetura, perguntando “O que acontece se *esta* parte da arquitetura do microsserviço falhar?” (veja “Compreensão do microsserviço” para ter

acesso a mais detalhes sobre as revisões de arquitetura e a descoberta de pontos únicos de falha).


Nenhum ponto isolado de falha

Devido às complexas cadeias de dependência que existem entre diferentes microsserviços dentro de um ecossistema de microsserviços, um ponto de falha na arquitetura de um microsserviço individual frequentemente é um ponto de falha para toda a cadeia de dependência e, em casos extremos, para todo o ecossistema. Não há pontos de falha isolados dentro de ecossistemas de microsserviços, o que torna a identificação, a mitigação e a eliminação de pontos de falha essenciais para atingir o estado de tolerância a falhas.

Uma vez identificado o único (ou múltiplo) ponto de falha, é preciso que ele seja mitigado e (se possível) eliminado. Se o ponto de falha pode ser completamente eliminado e substituído por algo mais tolerante a falhas, então o problema está resolvido. Infelizmente, nem sempre é possível evitar todos os pontos de falha, e há algumas situações em que não podemos eliminar nem mesmo os pontos de falha mais evidentes e óbvios. Por exemplo, se nossa organização de engenharia de software exigir o uso de certa tecnologia que funcione bem para o restante das equipes de desenvolvimento, mas represente um ponto de falha único para nosso serviço, então talvez não haja uma maneira de eliminá-lo, e nossa única opção neste caso para tornar o serviço tolerante a falhas é encontrar formas de mitigar quaisquer consequências negativas de sua falha.

Cenários de catástrofes e falhas

Se existe algo que sabemos sobre sistemas complexos e arquitetura de sistemas distribuídos de larga escala é que o sistema irá falhar de algum modo, e qualquer falha que poderia ocorrer quase certamente ocorrerá em algum momento da vida útil do sistema.

Microsserviços são sistemas complexos. Eles são parte de sistemas distribuídos de larga escala (ecossistemas de microsserviços) e, portanto, não são exceção a esta regra. Qualquer possível falha e qualquer possível catástrofe quase certamente ocorrerão em algum ponto entre o momento em que a RFC (request for comments ou solicitação de comentários) de um microsserviço for escrita e o momento em que o microsserviço for descontinuado e desativado. Catástrofes ocorrem o tempo todo: racks caem

em datacenters, sistemas de ar-condicionado HVAC quebram, databases de produção são apagados por acidente (sim, isso acontece com mais frequência do que a maioria dos desenvolvedores gostaria de admitir), desastres naturais destroem datacenters inteiros. Qualquer falha que possa acontecer irá acontecer: dependências falharão, servidores individuais falharão, bibliotecas serão corrompidas ou perdidas completamente, o monitoramento falhará, os logs poderão ser e serão perdidos (aparentemente, desaparecendo no ar).

Uma vez identificados, mitigados e (se possível) eliminados quaisquer pontos de falha claramente óbvios na arquitetura de nosso microsserviço, o próximo passo é *identificar quaisquer outros cenários de falhas e potenciais catástrofes* que possam afetar nosso microsserviço. Podemos separar esses tipos de falhas e catástrofes em quatro categorias principais, organizando-os de acordo com sua posição na pilha do ecossistema de microsserviço. Os cenários mais comuns de falhas e catástrofes são falhas de hardware, falhas de infraestrutura (camada de comunicação e camada de plataforma de aplicação), falhas de dependências e falhas internas. Analisaremos com atenção alguns dos possíveis cenários de falhas mais comuns dentro de cada categoria nas próximas seções, mas primeiro trataremos de algumas causas comuns de falhas que afetam todos os níveis do ecossistema de microsserviços.

É preciso observar que as listas de possíveis cenários de falhas apresentadas aqui não são completas. O objetivo aqui é apresentar os cenários mais comuns e encorajar o leitor a definir a que tipos de falhas e catástrofes seu(s) microsserviço(s) e ecossistema(s) de microsserviços podem estar suscetíveis e então (quando necessário) indicar ao leitor outros capítulos neste livro em que alguns dos tópicos relevantes são tratados. A maioria das falhas aqui pode ser evitada adotando-se os padrões de disponibilidade de produção (e implementando seus requisitos correspondentes) encontrados ao longo do livro, portanto só mencionei algumas das falhas e não incluí qualquer falha que é tratada nos outros capítulos.

Falhas comuns em um ecossistema

Existem algumas falhas que acontecem em todos os níveis do ecossistema de microsserviços. Estes tipos de falha são geralmente causados (de uma

forma ou outra) pela falta de padronização em uma organização de engenharia de software, pois tendem a ser de natureza operacional (e não necessariamente técnica). Referir-se a essas falhas como “operacionais” não quer dizer que elas sejam menos importantes ou menos perigosas do que falhas técnicas, e nem quer dizer que resolver essas falhas esteja fora do campo técnico e não seja responsabilidade das equipes de desenvolvimento de microsserviços. Esses tipos de falha tendem a ser os mais sérios, a ter algumas das consequências técnicas mais debilitantes e refletem falta de alinhamento entre as diversas equipes de engenharia em uma empresa. Em meio a esse tipo de falha, as falhas mais comuns são revisões insuficientes de projeto da arquitetura do sistema e do serviço, revisões incompletas do código, processos precários de desenvolvimento e procedimentos instáveis de implantação.

Revisões insuficientes de projeto da arquitetura do sistema e de microsserviços resultam em serviços mal projetados, especialmente em grandes e complexos ecossistemas de microsserviços. A razão para isso é simples: nenhum engenheiro e nenhuma equipe de desenvolvimento de microsserviço conhecem os detalhes da infraestrutura e a complexidade de todos os quatro níveis do ecossistema. Enquanto novos sistemas estão sendo projetados e novos microsserviços são arquitetados, é vital para a futura tolerância a falhas do sistema ou do serviço que engenheiros de cada nível do ecossistema de microsserviços sejam agregados ao processo de projeto para determinar como o sistema ou o serviço deve ser construído e executado em função da complexidade de todo o ecossistema. Entretanto, ainda que isso seja feito adequadamente enquanto o sistema ou o serviço estiver sendo projetado, os ecossistemas de microsserviços evoluem tão rapidamente que a infraestrutura em geral é praticamente irreconhecível depois de um ou dois anos; portanto revisões programadas da arquitetura com especialistas de cada parte da organização podem ajudar a garantir que o sistema ou o microsserviço esteja atualizado e se encaixe corretamente no ecossistema como um todo. Para ter acesso a mais detalhes sobre as revisões de arquitetura, veja o Capítulo 7, *Documentação e compreensão*.

Revisões incompletas de código formam outra fonte comum de falhas. Embora este problema não seja específico da arquitetura de microsserviços, a adoção da arquitetura de microsserviços tende a

exacerbar o problema. Considerando a maior velocidade de desenvolvimento que acompanha os microsserviços, os desenvolvedores são geralmente obrigados a rever todo novo código escrito por seus colegas de equipe várias vezes por dia, além de escrever seu próprio código, participar de reuniões e fazer todo o resto necessário para realizar seu(s) serviço(s). Isso requer uma constante alternância entre contextos, e é fácil perder a atenção aos detalhes dentro do código de outra pessoa quando você mal tem tempo para revisar seu próprio código antes de implantá-lo. A consequência disso resulta em inúmeros erros que são introduzidos no ambiente de produção, erros que fazem serviços e sistemas falharem, erros que poderiam ter sido detectados por meio de uma melhor revisão de código. Há várias formas de mitigar isso, mas não é possível resolver esse problema completamente em um ambiente com alta velocidade de desenvolvimento. É preciso tomar cuidado para escrever extensivos testes para cada sistema e serviço, para testar cada nova mudança extensivamente antes de elas chegarem ao ambiente de produção e para garantir que, se os erros não foram detectados antes da implantação, eles serão detectados em algum outro ponto do processo de desenvolvimento ou do pipeline de deployment, o que nos leva às nossas duas próximas causas de falha.

Uma das importantes causas de interrupções em ecossistema de microsserviços são as implantações ruins. Implantações “ruins” são aquelas que contêm erros no código, versões quebradas etc. Processos de desenvolvimento ruins ou procedimentos instáveis de implantação permitem que falhas sejam introduzidas na produção, derrubando qualquer sistema ou serviço no qual o problema causador da falha tenha sido implantado junto a qualquer de suas dependências (e, às vezes, a todas). Implantar bons procedimentos de revisão e criar uma cultura de engenharia, na qual a revisão de código seja levada a sério e os desenvolvedores tenham tempo adequado para se concentrar na revisão do código de seus colegas, é o primeiro passo para evitar esses tipos de falhas, mas muitas delas continuarão não sendo detectadas: mesmo o melhor revisor de código não consegue prever exatamente como uma mudança de código ou um novo recurso se comportará em ambiente de produção sem mais testes. A única maneira de detectar essas falhas antes que elas derrubem o sistema ou o serviço é construir processos de desenvolvimento e pipelines de implantação estáveis e confiáveis. Os detalhes da construção

de processos de desenvolvimento e pipelines de implantação estáveis e confiáveis são tratados no Capítulo 3 – *Estabilidade e confiabilidade*.

Resumo: falhas comuns em um ecossistema

As falhas mais comuns que ocorrem em todos os níveis de ecossistemas de microsserviços são:

- insuficiente revisão do projeto de arquitetura de sistema e serviço;
- revisões incompletas de código;
- processos precários de desenvolvimento;
- procedimentos instáveis de implantação.

Falhas de hardware

A camada inferior da pilha é onde se encontra o hardware. A camada de hardware consiste nos computadores físicos nos quais é executado todo o código da infraestrutura e aplicação, além dos racks em que os servidores estão instalados, os datacenters em que os servidores funcionam e, no caso de provedores de serviços de nuvem, as regiões e as zonas de disponibilidade. A camada de hardware também contém o sistema operacional, o isolamento e a abstração de recursos, o gerenciamento de configuração, o monitoramento e logging no âmbito do servidor. (Para ter acesso a mais detalhes sobre a camada de hardware do ecossistema de microsserviços, veja o Capítulo 1 – *Microsserviços*.)

Muita coisa pode dar errado nesta camada do ecossistema, e ela é a camada mais afetada por genuínas catástrofes (e não apenas por falhas). Ela é também a camada mais delicada do ecossistema: se o hardware falhar e não houver alternativas, toda a organização de engenharia de software cairá junto com ele. As catástrofes que ocorrem aqui são puras falhas de hardware: uma máquina para de funcionar, um rack cai ou todo um datacenter falha. Essas catástrofes acontecem com mais frequência do que gostaríamos de admitir, e para que um ecossistema de microsserviços seja tolerante a falhas, para que um microsserviço individual seja tolerante a falhas e preparado para essas catástrofes, é preciso se preparar para enfrentar essas falhas, mitigá-las e se proteger contra elas.

Tudo mais dentro dessa camada e tudo o que se apoia sobre as máquinas

físicas também podem falhar. As máquinas precisam ser provisionadas antes de poder executar qualquer coisa, e se o provisionamento falhar, então, utilizar máquinas novas (ou, em alguns casos, mesmo máquinas antigas) não será possível. Muitos ecossistemas de microsserviços que utilizam tecnologias que suportam isolamento de recursos (como Docker) ou abstração e alocação de recursos (como Mesos e Aurora) também podem quebrar ou falhar, e suas falhas podem parar todo o ecossistema. Falhas causadas por gerenciamento de configuração separado ou por alterações de configuração também são extraordinariamente comuns e geralmente difíceis de detectar. O monitoramento e logging também podem falhar redundantemente, e se isso ocorrer no âmbito do servidor, fazer a triagem de qualquer interrupção se tornará impossível, pois os dados necessários para mitigar quaisquer problemas não estarão disponíveis. Falhas de rede (tanto internas quanto externas) também podem ocorrer. Finalmente, downtimes operacionais de componentes críticos de hardware – mesmo se comunicados adequadamente ao longo da organização – podem resultar em interrupções no ecossistema.

Resumo: cenários comuns de falhas de hardware

Alguns dos cenários mais comuns de falhas de hardware são:

- falha no servidor;
- falha no rack;
- falha no datacenter;
- falha no provedor de serviços de nuvem;
- falha no provisionamento do servidor;
- falha na tecnologia de isolamento e/ou na abstração de recursos;
- gerenciamento separado de configuração;
- falhas causadas por mudanças de configuração;
- falhas e lacunas no monitoramento no âmbito do servidor;
- falhas e lacunas no logging no âmbito do servidor;
- falha de rede;
- downtimes operacionais;
- falta de redundância de infraestrutura.

Falhas no âmbito da comunicação e plataforma de aplicação

A segunda e a terceira camada da pilha do ecossistema de microsserviços são compostas das camadas de comunicação e da plataforma de aplicação. Essas camadas localizam-se entre a camada de hardware e a camada de microsserviços, estabelecendo uma ligação entre as duas, como se fossem a cola que mantém o ecossistema unido. A camada de comunicação contém a rede, o DNS, o framework RPC, os endpoints, a troca de mensagens, a descoberta de serviços, o registro de serviços e o balanceamento de carga. A camada de plataforma de aplicação consiste nas ferramentas de desenvolvimento do tipo autosserviço, no ambiente de desenvolvimento, nas ferramentas de teste, empacotamento, liberação e versão, no pipeline de deployment, no logging no âmbito do microsserviço e no monitoramento no âmbito do microsserviço – todos esses itens são críticos para a operação e a construção diária do ecossistema de microsserviços. Como as falhas de hardware, as falhas que ocorrem nesses níveis comprometem toda a empresa, pois todos os aspectos do desenvolvimento e da manutenção dentro do ecossistema de microsserviços dependem de forma crítica do funcionamento tranquilo e sem falhas desses sistemas. Vamos dar uma olhada nas falhas mais comuns que podem ocorrer dentro dessas camadas.

Na camada de comunicação, as falhas de rede são especialmente comuns. Elas podem ser falhas da(s) rede(s) interna(s) por meio das quais todas as chamadas de procedimento remoto (RPCs) são feitas, ou falhas de redes externas. Outro tipo de falha de rede resulta de problemas com firewalls e entradas incorretas nas tabelas de IP. Erros de DNS também são comuns: quando ocorrem erros de DNS, a comunicação pode parar e erros de DNS podem ser difíceis de descobrir e diagnosticar. A camada RPC da comunicação – a cola que mantém unido todo o delicado ecossistema de microsserviços – é outra (infame) fonte de falhas, especialmente quando só há um canal conectando todos os microsserviços e sistemas internos; configurar canais separados para RPC e verificações de saúde pode mitigar um pouco esse problema se as verificações de saúde e outros monitoramentos relacionados forem mantidos separados dos canais que tratam os dados que trafegam entre os serviços. É possível que os sistemas de troca de mensagens falhem (como mencionei brevemente no exemplo Redis-Celery anteriormente neste capítulo), e filas de mensagens, brokers

de mensagens e processadores de tarefas geralmente existem em ecossistemas de microsserviços sem qualquer backup ou alternativa, operando como alarmantes pontos de falha para cada serviço que depende deles. Falhas da descoberta de serviços, do registro de serviços e do balanceamento de carga também podem (e irão) ocorrer: se qualquer parte desses sistemas falhar ou sofrer downtime sem qualquer alternativa, então o tráfego não será roteado, alocado e distribuído adequadamente.

Falhas dentro da plataforma de aplicação são mais específicas do modo como as organizações de engenharia de software configuraram seu processo de desenvolvimento e pipeline de deployment, mas, como regra, esses sistemas podem falhar de modo tão frequente e catastrófico quanto qualquer outro serviço dentro da pilha do ecossistema. Se as ferramentas e/ou os ambientes de desenvolvimento estiverem funcionando incorretamente quando os desenvolvedores tentarem construir novos recursos ou reparar erros existentes, novos modos de falha e erros poderão ser introduzidos na produção. O mesmo vale para quaisquer falhas ou defeitos dos pipelines de teste, empacotamento, liberação e versão: se os pacotes e as versões contêm erros ou não são bem construídos, então as implantações falham. Se um pipeline de deployment não estiver disponível, contiver erros ou falhar totalmente, então a implantação irá parar, impedindo não apenas a implantação de novos recursos, mas também de correções de erros críticos que estão sendo preparadas. Finalmente, o monitoramento e o logging de microsserviços individuais podem conter lacunas ou falhas, tornando a depuração ou o registro de erros impossível.

Resumo: falhas comuns de comunicação e da plataforma de aplicação

Algumas das falhas mais comuns de comunicação e da plataforma de aplicação são:

- falhas de rede;
- erros de DNS;
- falhas de RPC;
- tratamento inadequado de solicitações e/ou respostas;
- falhas no sistema de troca de mensagens;

- falhas na descoberta de serviços e no registro de serviços;
- balanceamento de carga incorreto;
- falha nas ferramentas e no ambiente de desenvolvimento;
- falhas nos pipelines de teste, empacotamento, versão e liberação;
- falhas no pipeline de deployment;
- falhas e lacunas no logging no âmbito de microsserviço.
- falhas e lacunas no monitoramento no âmbito de microsserviço.

Falhas de dependência

Falhas dentro do nível superior do ecossistema de microsserviços (a camada de microsserviços) podem ser divididas em duas categorias separadas: (1) as que são internas, de um microsserviço específico e causadas por problemas dentro dele, e (2) as que são externas em relação a um microsserviço e causadas pelas dependências desse microsserviço. Primeiro, trataremos dos cenários comuns de falha dentro da segunda categoria.

Falhas e interrupções de um microsserviço downstream (isto é, uma das dependências de um microsserviço) são extraordinariamente comuns e podem afetar dramaticamente a disponibilidade de um microsserviço. Mesmo se apenas um microsserviço na cadeia de dependências cair, este pode derrubar todos os seus clientes upstream se não houver proteções em funcionamento. Entretanto um microsserviço nem sempre precisa sofrer uma interrupção total para afetar negativamente a disponibilidade de seus clientes upstream – se ele não cumprir seu SLA por um ou dois noves, a disponibilidade de todos os microsserviços-cliente upstream diminuirá.


O verdadeiro custo de não cumprir os SLAs

Os microsserviços podem fazer com que seus clientes upstream deixem de cumprir seus SLAs. Se a disponibilidade de um serviço cair por um ou dois noves, todos os clientes upstream sofrerão graças à matemática: a disponibilidade de um microsserviço é calculada como sua própria disponibilidade multiplicada pela disponibilidade de suas dependências downstream. Deixar de cumprir um SLA é uma falha importante (e frequentemente ignorada) do microsserviço e é uma falha que derruba a disponibilidade de todos os outros serviços que dependem dele (junto aos serviços que dependem desses serviços).

Outras falhas comuns de dependência são as causadas por timeouts para outro serviço, a descontinuação ou desativação dos endpoints de API de uma dependência (sem a devida comunicação sobre a descontinuação ou a desativação para todos os clientes upstream) e a descontinuação ou a desativação de todo um microsserviço. Além disso, não é recomendável atribuir versões para bibliotecas internas e/ou microsserviços, e vincular-se a versões específicas de bibliotecas internas e/ou serviços na arquitetura de microsserviços, pois isso tende a levar a erros e (em casos extremos) a falhas sérias, graças à natureza acelerada do desenvolvimento de microsserviços: essas bibliotecas e esses serviços estão constantemente mudando, e vincular-se a versões específicas (junto à atribuição de versão desses serviços e dessas bibliotecas em geral) pode levar os desenvolvedores a usar versões instáveis, não confiáveis e às vezes não seguras deles.

Falhas de dependências externas (serviços de e/ou bibliotecas de terceiros) também podem e irão ocorrer. Elas podem ser mais difíceis de detectar e corrigir do que falhas de dependências internas, pois os desenvolvedores terão pouco ou nenhum controle sobre elas. A complexidade associada à dependência de serviços e/ou bibliotecas de terceiros pode ser tratada adequadamente se esses cenários forem antecipados desde o início do ciclo de vida do microsserviço: escolha dependências externas estáveis e estabelecidas e tente evitar usá-las a menos que seja absolutamente necessário; caso contrário, elas se tornarão um ponto único de falha para seu serviço.

Resumo: cenários comuns de falha de dependência

Alguns dos cenários mais comuns de falhas de dependência são:

- falhas ou interrupções de um microsserviço downstream (dependência);
- interrupções de serviços internos;
- interrupções de serviços externos (de terceiros);
- falhas de bibliotecas internas;
- falhas de bibliotecas externas (de terceiros);
- uma dependência não consegue cumprir seu SLA;
- descontinuação do endpoint de API;

- desativação do endpoint de API;
- descontinuação do microsserviço;
- desativação do microsserviço;
- descontinuação da interface ou do endpoint;
- timeouts para um serviço downstream;
- timeouts para uma dependência externa.

Falhas internas (microsserviço)

Na parte superior da pilha do ecossistema de microsserviços estão os microsserviços individuais. Para as equipes de desenvolvimento, essas são as falhas mais importantes, pois elas dependem completamente de boas práticas de desenvolvimento, de boas práticas de implantação e do modo como as equipes de desenvolvimento projetam, executam e mantêm seus microsserviços individuais.

Assumindo que a infraestrutura abaixo da camada de microsserviço seja relativamente estável, a maioria dos incidentes e das interrupções sofridas por um microsserviço será quase exclusivamente autoinfligida. Os desenvolvedores de plantão para seus serviços serão acionados quase exclusivamente por questões e falhas cujas causas principais são encontradas dentro de seu microsserviço – isto é, os alertas que receberão terão sido disparados pelas mudanças nas principais métricas de seu microsserviço (veja o Capítulo 6, *Monitoramento*, para obter mais informações sobre monitoramento, logging, alertas e principais métricas do microsserviço).

Revisões incompletas de código, a falta de uma adequada cobertura de testes e processos ruins de desenvolvimento em geral (especificamente a falta de um ciclo padronizado de desenvolvimento) levam à implantação de código com erro na produção – falhas que podem ser evitadas pela padronização do processo de desenvolvimento nas equipes de microsserviço (veja “O ciclo de desenvolvimento”). Sem um pipeline de deployment estável e confiável contendo as fases de staging, pré-release e produção para detectar erros antes de eles serem integralmente implantados nos servidores de produção, quaisquer problemas não detectados por testes nas fases de desenvolvimento podem causar sérios incidentes e interrupções no próprio microsserviço, em suas dependências

e em quaisquer outras partes do ecossistema de microsserviços que dependem dele.

Qualquer elemento específico da arquitetura de microsserviços também pode falhar, incluindo databases, brokers de mensagem, sistemas de processamento de tarefas e similares. É aí que erros de código genéricos e específicos dentro do microsserviço causarão falhas, assim como um tratamento inadequado de erros e exceções: exceções não tratadas e a prática de detectar exceções são frequentemente culpadas e ignoradas quando microsserviços falham. Finalmente, aumentos de tráfego podem fazer com que um serviço falhe se ele não estiver preparado para um crescimento inesperado (para mais informações sobre limitações de escalabilidade, veja o Capítulo 4, *Escalabilidade e desempenho*, e então leia a seção “Teste de carga” do atual capítulo).

Resumo: cenários comuns de falha de microsserviço

Alguns dos cenários mais comuns de falha de microsserviço são:

- revisões incompletas de código;
- arquitetura e projetos precários;
- falta de adequados testes de unidade e integração;
- implantações ruins;
- falta de um monitoramento adequado;
- tratamento inadequado de erros e exceções;
- falha no database;
- limitações de escalabilidade.

Teste de resiliência

Eliminar pontos únicos de falha e identificar possíveis cenários de falha e catástrofes não são suficientes para garantir que os microsserviços sejam tolerantes a falhas e preparados para qualquer catástrofe. Para ser realmente tolerante a falhas, um microsserviço precisa ser capaz de sofrer falhas e se recuperar com elegância sem afetar sua própria disponibilidade, a disponibilidade de seus clientes e a disponibilidade do ecossistema de microsserviços como um todo. A melhor maneira de garantir que um microsserviço seja tolerante a falhas é pegar todos os possíveis cenários de

falha que poderiam afetá-lo e então submetê-lo a eles de forma ativa, repetida e aleatória em produção – uma prática conhecida como *teste de resiliência*.

Um microsserviço resiliente é aquele que sofre e se recupera de falhas em todos os níveis do ecossistema de microsserviços: a camada de hardware (por exemplo, uma falha no servidor ou datacenter), a camada de comunicação (por exemplo, falhas de RPC), a camada de aplicação (por exemplo, uma falha no pipeline de deployment) e a camada de microsserviço (por exemplo, falha de uma dependência, uma implantação ruim ou um aumento repentino de tráfego). Há vários tipos de teste de resiliência que, quando usados para avaliar a tolerância a falhas de um microsserviço, podem garantir que o serviço esteja preparado para qualquer falha conhecida dentro de qualquer camada da pilha.

O primeiro tipo de teste de resiliência que abordaremos é o *teste de código*, que consiste em quatro tipos de testes que verificam a sintaxe, o estilo, os componentes individuais do microsserviço, a forma como os componentes trabalham em conjunto e como o microsserviço se comporta dentro de suas complexas cadeias de dependência. (O teste de código geralmente não é considerado parte do pacote de testes de resiliência, mas eu quis incluí-lo aqui por duas razões: primeiro, uma vez que ele é crucial para a tolerância a falhas e preparação para catástrofes, faz sentido mantê-lo nesta seção; segundo, eu notei que equipes de desenvolvimento preferem manter todas as informações de teste em um único lugar.) O segundo é o *teste de carga*, no qual microsserviços são expostos a cargas maiores de tráfego para ver como se comportam nessas condições. O terceiro tipo de teste de resiliência é o *teste de caos*, que é o tipo de teste de resiliência mais importante e no qual microsserviços são ativamente forçados a falhar em ambiente de produção.

Teste de código

O primeiro tipo de teste de resiliência é o *teste de código*, uma prática com a qual quase todos os desenvolvedores e engenheiros operacionais estão familiarizados. Na arquitetura de microsserviços, o teste de código precisa ser executado em todas as camadas do ecossistema, tanto dentro dos microsserviços quanto em quaisquer sistemas ou serviços que estão localizados nas camadas abaixo dele: além dos microsserviços, também é

preciso testar a descoberta de serviços, o gerenciamento de configuração e sistemas relacionados. Há vários tipos de boas práticas de teste de código, incluindo *teste lint*, *teste de unidade*, *teste de integração* e *teste fim a fim*.

Teste lint

Erros de sintaxe e estilo são detectados usando-se o *teste lint*. Testes lint são executados sobre o código, detectando quaisquer problemas específicos da linguagem, e também podem ser escritos para garantir que o código esteja de acordo com os guias de estilo específicos da linguagem (e às vezes específicos da equipe ou específicos da organização).

Testes de unidade

A maioria dos testes de código é feita por meio de *testes de unidade*, que são pequenos testes independentes executados sobre vários pequenos trechos (ou unidades) do código do microsserviço. O objetivo dos testes de unidade é garantir que os componentes de software do próprio serviço (por exemplo, funções, classes e métodos) sejam resilientes e não contenham erros. Infelizmente, muitos desenvolvedores só consideram testes de unidade ao escrever testes para suas aplicações ou seus microsserviços. Embora o teste de unidade seja bom, ele não é suficiente para de fato avaliar as várias maneiras como o microsserviço irá se comportar em produção.

Testes de integração

Enquanto testes de unidade avaliam pequenas partes do microsserviço para garantir que os componentes são resilientes, o próximo tipo de teste de código é o *teste de integração*, que testa como o serviço como um todo funciona. No teste de integração, todos os componentes menores do microsserviço (que foram testados individualmente nos testes de unidade) são combinados e testados juntos para garantir que funcionam como esperado quando precisam trabalhar em conjunto.

Testes fim a fim

Para uma aplicação monolítica ou independente, em geral, os testes de unidade e de integração são suficientes para juntos englobarem o aspecto de teste de código do teste de resiliência, mas a arquitetura de microsserviços introduz um novo nível de complexidade dentro do teste de

código devido às complexas cadeias de dependência que existem entre um microsserviço, seus clientes e suas dependências. Outro conjunto adicional de testes precisa ser acrescentado ao pacote de testes de código que avalia o comportamento do microsserviço em relação a seus clientes e suas dependências. Isso significa que desenvolvedores de microsserviço precisam construir *testes fim a fim* que funcionem como um tráfego real de produção, testes que acessem os endpoints dos clientes de seu microsserviço, acessem os endpoints de seu próprio microsserviço, acessem os endpoints das dependências do microsserviço, enviem solicitações de leitura para qualquer database e detectem quaisquer problemas no fluxo de solicitação que possam ter sido introduzidos com uma alteração de código.

Automatizar os testes de código

Todos os quatro tipos de testes de código (lint, unidade, integração e fim a fim) devem ser escritos pela equipe de desenvolvimento, mas sua execução deve ser automatizada como parte do ciclo de desenvolvimento e do pipeline de deployment. Testes de unidade e integração devem ser executados durante o ciclo de desenvolvimento em um sistema externo, logo depois que as alterações passaram pelo processo de revisão de código. Se as alterações do novo código forem rejeitadas por qualquer um dos testes de unidade ou integração, então elas não deverão ser introduzidas no pipeline de deployment como candidatas à produção, mas, sim, ser rejeitadas e levadas à atenção da equipe de desenvolvimento para correções. Se as alterações do novo código passarem por todos os testes de unidade e integração, então a nova versão deverá ser enviada para o pipeline de deployment como candidata à produção.

Resumo do teste de código

Os quatro tipos de teste de código para saber se o serviço está pronto para produção são:

- testes lint;
- testes de unidade;
- testes de integração;
- testes fim a fim.

Teste de carga

Como vimos no Capítulo 4, *Escalabilidade e desempenho*, um microsserviço pronto para produção precisa ser escalável e de alto desempenho. Ele precisa tratar um grande número de tarefas ou solicitações ao mesmo tempo e tratá-las com eficiência, e também precisa estar preparado para o crescimento do número de tarefas ou solicitações no futuro. Os microsserviços que não estiverem preparados para aumento de tráfego, de tarefas ou solicitações poderão sofrer graves interrupções quando qualquer uma dessas condições ocorrer.

Do ponto de vista de uma equipe de desenvolvimento de microsserviço, sabemos que o tráfego em direção ao nosso microsserviço provavelmente crescerá em um momento futuro, e podemos até saber exatamente de quanto será este aumento de tráfego. Queremos estar totalmente preparados para esse aumento de tráfego para evitarmos potenciais problemas e/ou falhas. Além disso, queremos identificar quaisquer possíveis desafios e gargalos de escalabilidade que talvez só percebamos quando nosso microsserviço for levado aos limites de sua escalabilidade. Para se proteger contra incidentes e interrupções relacionados à escalabilidade, e para estarmos totalmente preparados para futuros aumentos de tráfego, podemos testar a escalabilidade de nossos serviços usando o *teste de carga*.

Fundamentos do teste de carga

O teste de carga é exatamente o que o nome diz: é uma forma de testar como um microsserviço se comporta sob uma carga de tráfego específica. Durante o teste de carga, uma carga de tráfego é escolhida, esta carga de tráfego de teste é enviada ao microsserviço e o microsserviço é monitorado de perto para ver como se comporta. Se o microsserviço falhar ou experimentar qualquer problema durante o teste de carga, seus desenvolvedores poderão resolver quaisquer questões de escalabilidade que surgirem nos testes de carga e que poderiam prejudicar a disponibilidade de seu microsserviço no futuro.

O teste de carga é onde observamos a utilidade das escalas de crescimento e os gargalos e requisitos de recursos que foram tratados no Capítulo 4, *Escalabilidade e desempenho*. Com base na escala de crescimento qualitativo de um microsserviço e nas métricas de negócio de alto nível

associadas, as equipes de desenvolvimento podem saber quanto tráfego seu microsserviço deve estar preparado para tratar no futuro. A partir da escala de crescimento quantitativo, os desenvolvedores saberão exatamente quantas solicitações ou consultas por segundo seu serviço deve ser capaz de tratar. Se a maioria dos gargalos e requisitos de recursos do serviço tiver sido identificada, e os gargalos forem eliminados, os desenvolvedores saberão como traduzir a escala de crescimento quantitativo (e, consequentemente, os aspectos quantitativos de futuros aumentos de tráfego) em termos dos recursos de hardware que seu microsserviço exigirá para tratar cargas maiores de tráfego. Os testes de carga depois da aplicação dos requisitos de escalabilidade e da resolução de pendências podem validar e garantir que o microsserviço esteja preparado para o esperado aumento de tráfego.

O teste de carga pode ser usado ao contrário, para descobrir as escalas de crescimento quantitativo e qualitativo, para identificar os gargalos e requisitos de recursos, para garantir o escalamento das dependências, para determinar e planejar futuras necessidades de capacidade, e assim por diante. Quando bem executado, o teste de carga pode fornecer aos desenvolvedores um *insight* profundo sobre a escalabilidade (e as limitações de escalabilidade) de seu microsserviço: ele mede como o serviço, suas dependências e o ecossistema se comportam em um ambiente controlado sob uma carga específica de tráfego.

Executar testes de carga nas fases de staging e produção

O teste de carga é mais eficaz quando executado em cada estágio do pipeline de deployment. Para testar o próprio framework de teste de carga, para garantir que o tráfego de teste produzirá os resultados desejados e para detectar potenciais problemas que o teste de carga poderia causar em produção, o teste de carga pode ser executado na fase de staging do pipeline de deployment. Se o pipeline de deployment usar staging parcial, onde o ambiente de staging se comunica com os serviços de produção, é preciso tomar cuidado para garantir que quaisquer testes de carga executados no ambiente de staging não prejudiquem ou comprometam a disponibilidade de quaisquer serviços de produção com os quais se comunica. Se o pipeline de deployment usar staging total, que é uma cópia-espelho completa da produção e onde nenhum serviço de staging se comunica com serviços na produção, será preciso tomar cuidado para

garantir que o teste de carga no ambiente de staging total produza resultados precisos, especialmente se não houver uma paridade de servidores entre os ambientes de staging e produção.

Não basta executar o teste de carga apenas na fase de staging. Mesmo os melhores ambientes de staging – aqueles que são cópias completas da produção e possuem paridade total de servidor – não são ambientes de produção. Eles não são o mundo real, e muito raramente os ambientes de staging são um indicativo perfeito das consequências do teste de carga em produção. Quando você souber qual é a carga de tráfego que precisa atingir, tiver alertado todas as equipes de dependência de plantão e tiver executado seus testes de carga no ambiente de staging, será absolutamente necessário executar os testes de carga na produção.

 Alerte as dependências quando estiver executando o teste de carga

Se os seus testes de carga enviam solicitações para outros serviços de produção, certifique-se de alertar todas as dependências para evitar comprometer sua disponibilidade enquanto executar testes de carga. Nunca assuma que as dependências downstream podem tratar a carga de tráfego que você está prestes a enviar para elas!

O teste de carga em produção pode ser perigoso e facilmente causar a falha de um microsserviço e de suas dependências. O motivo pelo qual o teste de carga é perigoso é o mesmo pelo qual ele é essencial: na maioria do tempo você não sabe exatamente como o serviço sendo testado se comporta sob a carga de teste e você não saberá como suas dependências tratam o número maior de solicitações. O teste de carga é *a* maneira de explorar os aspectos desconhecidos sobre um serviço e garantir que ele esteja preparado para o esperado crescimento de tráfego. Quando um serviço é levado aos seus limites no ambiente de produção e as coisas começam a entrar em colapso, é preciso haver itens automatizados em funcionamento para garantir que os testes de carga possam ser rapidamente desativados. Depois de descobertas e mitigadas as limitações do serviço e que as correções forem testadas e implantadas, o teste de carga poderá continuar.

Automatizando o teste de carga

Se o teste de carga será exigido para todos os microsserviços dentro da empresa (ou mesmo para um pequeno número de microsserviços críticos para o negócio), deixar a implementação e a metodologia do teste de carga

nas mãos das equipes de desenvolvimento para que estas o projetem e executem introduz outro ponto de falha no sistema. O ideal é que uma ferramenta e/ou um sistema de teste de carga do tipo autosserviço seja parte da camada de plataforma de aplicação do ecossistema de microsserviços, permitindo que os desenvolvedores usem um serviço confiável, compartilhado, automatizado e centralizado.

O teste de carga deve ser programado regularmente e considerado um componente integral das funções diárias da organização de engenharia de software. A programação deve estar vinculada aos padrões de tráfego: teste as cargas de tráfego desejadas em produção quando o tráfego estiver baixo e nunca durante as horas de pico, para evitar comprometer a disponibilidade dos serviços. Se um sistema de teste de carga centralizado do tipo autosserviço for usado, é bastante útil ter um processo automatizado para validar novos testes, junto a um pacote de testes confiáveis (e necessários) que todo serviço pode executar. Em casos extremos e quando uma ferramenta de teste de carga do tipo autosserviço for confiável, as implantações podem ser bloqueadas (ou confinadas) se um microsserviço não se comportar bem com testes de carga. E, mais importante, todo teste de carga executado precisa ser suficientemente registrado em logs e divulgado internamente para que os problemas causados pelo teste de carga possam ser rapidamente detectados, mitigados e resolvidos.

Resumo do teste de carga

O teste de carga pronto para produção tem os seguintes componentes:

- usa uma carga de tráfego que foi calculada com base nas escalas de crescimento qualitativo e quantitativo e expressa em termos de RPS, QPS ou TPS;
- é executado em cada estágio do pipeline de deployment;
- suas execuções são comunicadas para todas as dependências;
- é totalmente automatizado, gravado em log e programado.

Teste de caos

Neste capítulo, vimos vários potenciais cenários de falha e catástrofes que

podem ocorrer em cada camada da pilha. Vimos como o teste de código detecta pequenas potenciais falhas em nível de microsserviço individual e como o teste de carga detecta falhas resultantes das limitações de escalabilidade em nível de microsserviço. Entretanto a maioria dos cenários de falha e de potenciais catástrofes encontra-se em outra parte do ecossistema e não pode ser detectada por nenhum desses testes. Para testar *todos* os cenários de falhas, para garantir que os microsserviços possam se recuperar com elegância de potenciais catástrofes, há um tipo de adicional de teste de resiliência que precisa ser usado; ele é conhecido (muito adequadamente) como *teste de caos*.

No teste de caos, os microsserviços são ativamente *forçados a falhar* em produção, pois a única maneira de garantir que um microsserviço possa sobreviver a uma falha é fazê-lo falhar o tempo todo e de todas as formas possíveis. Isso significa que todo cenário de falha e de potencial catástrofe precisa ser identificado e então forçado a ocorrer em ambiente de produção. Executar testes programados e aleatórios de cada cenário de falha e potencial catástrofe pode ajudar a simular o mundo real de complexas falhas de sistema: desenvolvedores saberão qual parte do sistema será forçada a falhar de forma programada e se prepararão para esses testes de caos programados e também serão pegos desprevenidos por testes programados aleatórios.


Teste de caos responsável

O teste de caos precisa ser bem controlado para evitar que ele derrube o ecossistema. Certifique-se de que seu software de teste de caos tenha as permissões adequadas e que cada evento seja registrado em log para que, se os microsserviços não conseguirem se recuperar com elegância (ou se o teste de caos sair de controle), a identificação e resolução dos problemas não exija uma investigação muito profunda.

Como o teste de carga (e muitos outros sistemas tratados neste livro), é melhor oferecer o teste de caos como um serviço e não como uma implementação específica das várias equipes de desenvolvimento. Automatize o teste, exija que cada microsserviço execute um pacote de testes gerais e específicos, encoraje as equipes de desenvolvimento a descobrir outras maneiras que possam levar seu serviço a falhar e então lhes forneça os recursos para que projetem novos testes de caos que forcem seus microsserviços a falhar dessas outras maneiras. Certifique-se

de que cada parte do ecossistema (incluindo o serviço de teste de caos) possa sobreviver a um conjunto-padrão de testes de caos e quebre cada microsserviço e cada item de infraestrutura múltiplas vezes, repetidamente, até que todas as equipes de desenvolvimento e infraestrutura estejam confiantes de que seus serviços e sistemas possam aguentar as falhas inevitáveis.

Finalmente, o teste de caos não serve apenas para empresas hospedadas na nuvem, embora estas sejam os usuários mais expressivos (e comuns). Há poucas diferenças entre os modos de falha de computadores físicos e hardware na nuvem, e tudo que for construído para ser executado na nuvem pode funcionar igualmente bem em computadores físicos (e vice-versa). Uma solução de código aberto como Simian Army (que vem com um pacote-padrão de testes de caos que podem ser padronizados) funciona para a maioria das empresas, mas organizações com necessidades específicas podem facilmente construir seus próprios testes.

Exemplos de testes de caos

Alguns tipos comuns de testes de caos:

- Desative o endpoint de API de uma das dependências de um microsserviço.
- Pare todas as solicitações de tráfego para uma dependência.
- Introduza uma latência entre várias partes do ecossistema para simular problemas de rede: entre clientes e dependências, entre microsserviços e databases compartilhados, entre microsserviços e sistemas distribuídos de processamento de tarefas etc.
- Pare todo o tráfego para um datacenter ou uma região.
- Desligue um servidor aleatoriamente.

Detecção e reparo de falhas

Além do pacote de testes de resiliência, no qual microsserviços são testados à procura de todas as falhas e catástrofes conhecidas, um microsserviço pronto para produção precisa ter *estratégias de detecção e reparo de falhas* para quando as falhas ocorrerem. Vamos analisar os processos organizacionais que podem ser usados no ecossistema para

detectar, mitigar e resolver incidentes e interrupções, mas antes nos concentraremos em várias estratégias de mitigação técnica nesta seção.

Quando uma falha ocorre, o objetivo da detecção e do reparo de falhas sempre precisa ser o seguinte: *reduzir o impacto para os usuários*. Em um ecossistema de microsserviços, os "usuários" são todos aqueles que usam o serviço – isso pode ser outro microsserviço (que é cliente do serviço) ou um cliente real do produto (se o serviço em questão tiver relação com o cliente). Se a falha em questão foi (ou pode ter sido) introduzida na produção por uma nova implantação, a maneira mais eficaz de reduzir o impacto para os usuários quando acontecer algo errado é retornar imediatamente para a última versão estável do serviço. Retornar para a última versão estável garante que o microsserviço voltou para um estado conhecido, um estado que não era suscetível às falhas ou catástrofes que foram introduzidas com a versão mais recente. O mesmo vale para mudanças em configurações de baixo nível: trate as configurações como código, implante-as em várias liberações sucessivas e certifique-se de que, se uma mudança de configuração causar uma interrupção, o sistema possa rapidamente e sem esforço voltar para o último conjunto estável de configurações.

Uma segunda estratégia em caso de falha é comutar para uma alternativa estável. Se uma das dependências do microsserviço tiver caído, isso significa enviar as solicitações para um endpoint diferente (se o endpoint estiver quebrado) ou um serviço diferente (se todo o serviço estiver indisponível). Se não for possível rotear para outro serviço ou endpoint, então é preciso haver um modo de enfileirar ou salvar as solicitações e aguardar até que os problemas com a dependência tenham sido mitigados. Se o problema for relegado a outro datacenter ou se um datacenter apresentar falhas, a maneira de comutar para uma alternativa estável seria redirecionar o tráfego para outro datacenter. Sempre que houver várias maneiras de tratar falhas e uma das opções for redirecionar o tráfego para outro serviço ou datacenter, o redirecionamento do tráfego quase sempre será a opção mais inteligente: rotear o tráfego é fácil e imediatamente reduz o impacto para os usuários.

O importante é que o aspecto de detecção da "detecção e reparo de falhas" só pode realmente ser concretizado por meio do monitoramento pronto para produção (veja o Capítulo 6, *Monitoramento*, para analisar todos os

detalhes essenciais do monitoramento). Seres humanos são terríveis para detectar e diagnosticar falhas de sistema, e introduzir engenheiros no processo de detecção de falhas torna-se um ponto único de falha para o sistema como um todo. Isso também vale para o reparo de falhas: a maior parte do reparo dentro de grandes ecossistemas de microsserviços é feita por engenheiros, de modo quase dolorosamente manual, introduzindo um novo ponto de falha no sistema – mas não precisa ser assim. Para eliminar o potencial e a possibilidade de erro humano no reparo de falhas, todas as estratégias de mitigação devem ser automatizadas. Por exemplo, se um serviço não passar em certas verificações de saúde ou se as suas principais métricas atingirem os limites de advertência e/ou criticidade depois de uma implantação, então o sistema poderá ser projetado para automaticamente retornar à última versão estável. O mesmo vale para roteamento de tráfego para outro endpoint, microsserviço ou datacenter: se certas métricas principais atingirem limites específicos, configure um sistema que automaticamente roteie o tráfego para você. A tolerância a falhas requer absolutamente que o potencial e a possibilidade de erro humano sejam automatizados e eliminados sempre que possível.

Incidentes e interrupções

Ao longo deste livro eu enfatizei a disponibilidade dos microsserviços e do ecossistema como um todo como o objetivo da padronização. Planejar, construir e executar uma arquitetura de microsserviços que seja voltada para alta disponibilidade pode ser possível por meio da adoção de padrões de disponibilidade de produção e seus requisitos relacionados, e é a razão pela qual eu introduzi e escolhi cada padrão de disponibilidade de produção. Não é suficiente, porém, que microsserviços individuais e cada camada da pilha do ecossistema de microsserviços sejam tolerantes a falhas e preparados para catástrofes. As equipes de desenvolvimento e as empresas de engenharia responsáveis pelos microsserviços e o ecossistema no qual eles se encontram precisam adotar os adequados procedimentos de resposta organizacional para tratar incidentes e interrupções quando estes ocorrerem.

Todo minuto em que um microsserviço estiver fora de operação reduz sua disponibilidade. Quando parte de um microsserviço ou de seu ecossistema falha, causando um incidente ou uma interrupção, todo minuto de

inatividade pesa contra sua disponibilidade e faz com que ele não cumpra seu SLA. Não cumprir um SLA e não atingir os objetivos de disponibilidade proporcionam um sério custo: na maioria das empresas, interrupções incidem num enorme custo financeiro para o negócio, um custo que geralmente é fácil de quantificar e compartilhar com as equipes de desenvolvimento dentro da organização. Com isso em mente, é fácil ver como o tempo de detecção, o tempo de mitigação e o tempo de resolução de interrupções podem rapidamente se acumular e custar dinheiro à empresa, pois eles impactam negativamente o uptime de um microsserviço (e, consequentemente, sua disponibilidade).

Categorização adequada

Nem todos os microsserviços são criados da mesma maneira, e categorizar a importância e o impacto que suas falhas terão sobre o negócio facilita a adequada detecção, a mitigação e a resolução de incidentes e interrupções. Quando um ecossistema contém centenas ou mesmo milhares de microsserviços, há dezenas ou mesmo centenas de falhas por semana; mesmo se apenas 10% dos microsserviços sofrerem falha, isso ainda representará mais de cem falhas em um ecossistema de mil serviços. Embora toda falha precise ser adequadamente tratada por sua própria equipe de plantão, nem toda falha precisa ser tratada como uma emergência envolvendo todos da equipe.

Para ter um processo de resposta a incidentes e interrupções consistente, adequado, eficaz e eficiente em toda a organização, é importante fazer duas coisas. Primeiro, é incrivelmente útil categorizar os próprios microsserviços em relação a como suas falhas afetarão o ecossistema de modo que seja fácil priorizar vários incidentes e falhas (isso também ajuda com problemas relacionados à competição por recursos – tanto recursos de engenharia como recursos de hardware – dentro da organização). Segundo, incidentes e interrupções precisam ser categorizados para que o escopo e a gravidade de cada falha sejam entendidos em toda a organização.

Categorizando microsserviços

Para mitigar os desafios da competição por recursos e garantir que sejam tomadas medidas adequadas em resposta a incidentes, cada microsserviço dentro de ecossistema pode (e deve) ser categorizado e classificado de

acordo com sua criticidade para o negócio. A categorização não precisa ser perfeita inicialmente, já que uma categorização básica funciona bem. O importante aqui é marcar os microsserviços que são críticos para o negócio como tendo a mais alta prioridade e impacto, e então todos os demais microsserviços terão classificação e prioridade menores conforme sua proximidade ou a distância em relação aos serviços mais críticos. As camadas de infraestrutura são sempre as de mais alta prioridade: qualquer coisa dentro das camadas de hardware, comunicação e plataforma de aplicação que é usada por qualquer um dos microsserviços críticos para o negócio deve ter a mais alta prioridade dentro do ecossistema.

Categorizando incidentes e interrupções

Existem dois eixos sobre os quais os incidentes, as interrupções e as falhas podem ser mapeados: o primeiro é a *gravidade* do incidente, da interrupção ou da falha, e o segundo é seu *escopo*. A gravidade está ligada à categorização da aplicação, do microsserviço ou do sistema em questão. Se o microsserviço for crítico para o negócio (isto é, se o negócio ou uma parte essencial da interface de usuário do produto não puder funcionar sem ele), então a gravidade de qualquer falha que ele sofrer deve corresponder à categorização do serviço. O escopo, por outro lado, está relacionado a quanto do ecossistema é afetado pela falha, e é geralmente dividido em três categorias: alto, médio e baixo. Um incidente cujo escopo é alto é um incidente que afeta todo o negócio e/ou um recurso externo (por exemplo, interface de usuário); um incidente de escopo médio seria aquele que afeta apenas o próprio serviço, ou o serviço e alguns clientes; um incidente de baixo escopo seria aquele cujos efeitos negativos não são percebidos pelos clientes, pelo negócio ou por clientes externos usando o produto. Em outras palavras, a gravidade deve ser categorizada com base no impacto sobre o negócio e o escopo deve ser categorizado com base no fato de o incidente ser *local* ou *global*.

Vamos analisar alguns exemplos para entender como isso funciona na prática. Atribuiremos quatro níveis de gravidade a cada falha (0 a 4 – 0 é o incidente mais grave e 4, o menos grave) e usaremos a classificação alto/médio/baixo ao determinar o escopo. Primeiro, vejamos um exemplo cuja gravidade e cujo escopo sejam muito fáceis de categorizar: uma falha completa de um datacenter. Se um datacenter cai completamente (por qualquer motivo), a gravidade é claramente 0 (todo o negócio é afetado) e

o escopo é alto (novamente, todo o negócio é afetado). Vejamos agora outro cenário: imagine que temos um microsserviço que seja responsável por uma função crítica para o negócio e que ele esteja indisponível durante trinta minutos; como resultado desta falha, vamos imaginar que um de seus clientes seja afetado, mas o restante do ecossistema permaneça não afetado. Nós categorizaríamos esta gravidade como 0 (pois um recurso crítico para o negócio é afetado) e escopo médio (ela não afeta todo o negócio, apenas a si mesmo e um serviço-cliente). Finalmente, vamos considerar uma ferramenta interna responsável por gerar modelos para novos microsserviços, e imagine que ela fique indisponível por várias horas – como isso seria categorizado? Gerar modelos para novos microsserviços (e acelerar novos microsserviços) não é crítico para o negócio e não afeta algum recurso que tenha contato com o usuário, portanto este não seria um problema com gravidade 0 (também não seria de gravidade 1 ou 2); entretanto, já que o próprio serviço estaria indisponível, provavelmente nós classificaríamos sua gravidade como 3 e seu escopo como baixo (já que ele é o único serviço afetado por sua falha).

Os cinco estágios da resposta a incidentes

Quando falhas acontecem, é essencial para a disponibilidade de todo o sistema a existência de procedimentos padronizados de resposta a incidentes. Ter um conjunto claro de medidas a ser tomadas quando ocorre um incidente ou uma interrupção reduz o tempo de mitigação e resolução, o que por sua vez diminui o downtime sofrido por cada microsserviço. Na indústria atual, há três passos-padrão no processo de reagir a um incidente e solucioná-lo: triagem, mitigação e resolução. Adotar a arquitetura de microsserviços, porém, e alcançar alta disponibilidade e tolerância a falhas requer a adoção de dois passos adicionais no processo de resposta a incidentes: coordenação e acompanhamento (“follow-up”). Juntos, estes passos nos fornecem os cinco estágios da resposta a incidentes (Figura 5.1): *avaliação*, *coordenação*, *mitigação*, *resolução* e *acompanhamento*.

Figura 5.1 – Os cinco estágios da resposta a incidentes.

Avaliação

Sempre que um alerta é disparado por uma mudança em uma métrica-chave do serviço (veja o Capítulo 6, *Monitoramento*, para mais detalhes sobre alertas, métricas-chave e turnos de plantão) e o desenvolvedor de plantão para o serviço precisa responder ao alerta, o primeiro passo que precisa ser tomado é *avaliar* o incidente. O engenheiro de plantão é o primeiro a responder, fazendo uma triagem de todo o problema assim que ele dispara um alerta, e seu trabalho é definir a gravidade e o escopo desse problema.

Coordenação

Uma vez avaliado e triado o incidente, o próximo passo é primeiro *coordenar* com outros desenvolvedores e equipes e então começar a *comunicar o incidente*. Poucos desenvolvedores de plantão para dado serviço serão capazes de resolver todo problema do serviço, portanto a coordenação com outras equipes que *possam* resolver o problema garante que ele seja mitigado e resolvido rapidamente. Isso significa que precisa haver canais claros de comunicação de incidentes e interrupções de modo que qualquer problema de alta gravidade e alto escopo possa receber a atenção imediata que ele requer.

Durante o incidente ou a interrupção, é importante ter um registro claro da comunicação referente ao incidente por várias razões. Primeiro, registrar a comunicação durante o incidente (em logs de bate-papos, por e-mail etc.) ajuda a diagnosticar, determinar a causa fundamental e mitigar o incidente: todos sabem quem está trabalhando em qual correção, todos sabem quais

possíveis falhas foram eliminadas como possíveis causas e, uma vez identificada a causa fundamental, todos sabem exatamente o que causou o problema. Segundo, outros serviços que dependem do serviço que está sofrendo o incidente ou a interrupção precisam ser informados de quaisquer problemas de modo que possam mitigar seus efeitos negativos e garantir que seu próprio serviço esteja protegido contra a falha. Isso mantém a disponibilidade alta e evita que um serviço derrube toda a cadeia de dependências. Terceiro, isso ajuda quando forem escritas as "autópsias" dos incidentes globais graves ao se fornecer um registro claro e detalhado do que aconteceu exatamente e como o problema foi triado, mitigado e resolvido.

Mitigação

O terceiro passo é a *mitigação*. Depois que o problema foi avaliado e a comunicação organizacional teve início (garantindo que as pessoas certas estejam trabalhando para corrigir o problema), os desenvolvedores precisam trabalhar para reduzir o impacto do incidente sobre os clientes, o negócio e qualquer outra coisa que possa ser afetada pelo incidente. Mitigação não é o mesmo que resolução: não é *corrigir* a causa fundamental do problema completamente, apenas *reduzir seu impacto*. Um problema não é mitigado até que tanto sua disponibilidade como a disponibilidade de seus clientes não estejam mais comprometidas ou sofrendo.

Resolução

Depois que os efeitos do incidente ou da interrupção tiverem sido mitigados, os engenheiros poderão trabalhar para *resolver* a causa fundamental do problema. Este é o quarto passo do processo de resposta a incidentes. Isso implica em realmente corrigir a causa fundamental do problema, o que pode não ter sido feito quando o problema foi mitigado. O mais importante é que neste momento o relógio para de funcionar. As duas quantidades mais importantes que pesam contra o SLA de um microsserviço são o tempo de detecção (TTD) e o tempo de mitigação (TTM). Uma vez mitigado um problema, ele não deve mais afetar os usuários finais ou comprometer o SLA do serviço, e então o tempo de resolução (TTR) raramente (se alguma vez) pesa contra a disponibilidade de um serviço.

Acompanhamento

Três coisas precisam acontecer no quinto e último estágio, *acompanhamento*, da resposta a incidentes: é preciso escrever autópsias para analisar e entender o incidente ou a interrupção, os incidentes e as interrupções graves precisam ser compartilhados e analisados, e uma lista de ações precisa ser elaborada para que as equipes de desenvolvimento possam concluí-las e o microsserviço afetado retorne a um estado pronto para produção (ações geralmente podem se encaixar nas autópsias).

O aspecto mais importante do acompanhamento de incidentes é a *autópsia*. Em geral, uma autópsia é um documento detalhado que segue cada incidente e/ou interrupção e contém informações críticas sobre o que aconteceu, por que aconteceu e o que poderia ter sido feito para evitá-lo. Toda autópsia deve, no mínimo, conter um resumo do que aconteceu, dados sobre o que aconteceu (tempo de detecção, tempo de mitigação, tempo de resolução, downtime total, número de usuários afetados, quaisquer gráficos relevantes etc.), uma linha do tempo detalhada, uma análise abrangente sobre a causa fundamental, um resumo de como o incidente poderia ter sido evitado, maneiras de evitar interrupções semelhantes no futuro e uma lista de ações que precisam ser concluídas para que o serviço volte a um estado pronto para produção. Autópsias são mais eficazes quando não culpam ninguém, não apontam nomes, mas apenas fatos objetivos sobre o serviço. Apontar culpados, dar nome aos bois e culpar desenvolvedores e engenheiros pelas interrupções sufocam o aprendizado e o compartilhamento organizacional, que são essenciais para manter um ecossistema confiável e sustentável.

Em grandes e complexos ecossistemas de microsserviços, qualquer falha ou problema que derrube um microsserviço – seja pequeno ou grande – quase certamente pode (e irá) afetar pelo menos outro microsserviço no ecossistema. Comunicar incidentes e interrupções para as várias equipes (e por toda a organização) pode ajudar a detectar essas falhas em outros serviços antes que elas ocorram. Eu já vi como as análises de incidentes e interrupções podem ser eficazes quando feitas adequadamente e observei desenvolvedores participando dessas reuniões e então correndo para seu microsserviço a fim de corrigir quaisquer erros em seu próprio serviço que levaram aos incidentes e/ou às interrupções que estavam sendo analisados.

Avalie seu microsserviço

Agora que você compreende mais a tolerância a falhas e preparação para catástrofes, use a seguinte lista de perguntas para avaliar a disponibilidade de produção de seu(s) microsserviço(s) e ecossistema de microsserviços. As perguntas são organizadas por tópico e correspondem às seções dentro deste capítulo.

Evitando pontos únicos de falha

- O microsserviço tem um ponto único de falha?
- Ele tem mais de um ponto de falha?
- Os pontos de falha podem ser eliminados ou eles precisam ser mitigados?

Cenários de catástrofes e falhas

- Todos os cenários de falha e possíveis catástrofes do microsserviço foram identificados?
- Quais as falhas comuns no ecossistema de microsserviços?
- Quais os cenários de falha da camada de hardware que podem afetar este microsserviço?
- Quais falhas nas camadas de comunicação e aplicação podem afetar este microsserviço?
- Quais falhas de dependência podem afetar este microsserviço?
- Quais falhas internas podem derrubar este microsserviço?

Teste de resiliência

- Este microsserviço tem testes adequados do tipo lint, de unidade, de integração e fim a fim?
- Este microsserviço é submetido a testes de carga regulares e programados?
- Todos os possíveis cenários de falha são implementados e testados por meio do teste de caos?

Detecção e reparo de falhas

- Existem processos padronizados na organização de engenharia de software para tratar incidentes e interrupções?
- Como as falhas e as interrupções deste microsserviço impactam o negócio?
- Existem níveis claramente definidos de falha?
- Existem estratégias de mitigação claramente definidas?
- A equipe segue os cinco estágios de resposta a incidentes quando ocorrem incidentes e interrupções?

CAPÍTULO 6

Monitoramento

Um microsserviço pronto para produção é adequadamente monitorado. Um monitoramento adequado é uma das partes mais importantes de construir um microsserviço pronto para produção e garantir maior disponibilidade do microsserviço. Neste capítulo trataremos dos componentes essenciais do monitoramento do microsserviço, incluindo quais importantes métricas monitorar, como gravar log das métricas principais, construir dashboards que exibem as métricas principais, como abordar os alertas e as melhores práticas de plantões.

Princípios do monitoramento de microsserviços

A maioria das interrupções em um ecossistema de microsserviços é causada por implantações ruins. A segunda causa mais comum de interrupções é a falta de um adequado *monitoramento*. É fácil ver a razão disso. Se o estado de um microsserviço for desconhecido, se as métricas principais não forem rastreadas, então quaisquer falhas resultantes permanecerão desconhecidas até ocorrer uma real interrupção. Quando um microsserviço sofre uma interrupção em consequência da falta de monitoramento, sua disponibilidade já foi comprometida. Durantes essas interrupções, o tempo de mitigação e o tempo de reparo são prolongados, derrubando ainda mais a disponibilidade do microsserviço: sem informações facilmente acessíveis sobre as métricas principais do microsserviço, desenvolvedores ficam no escuro, despreparados para resolver rapidamente o problema. É por isso que um monitoramento adequado é essencial: ele fornece à equipe de desenvolvimento todas as informações relevantes sobre o microsserviço. Quando um microsserviço é adequadamente monitorado, seu estado nunca é desconhecido.

Monitorar um microsserviço pronto para produção requer quatro componentes. O primeiro é uma adequado *logging* de todas as informações

relevantes e importantes, o que permite que os desenvolvedores entendam o estado do microsserviço em qualquer momento do presente ou do passado. O segundo é o uso de bem projetados dashboards que refletem com precisão a saúde do microsserviço e são organizados de forma que qualquer pessoa na empresa seja capaz de ver o dashboard e entender a saúde e status do microsserviço sem dificuldade. O terceiro componente são *alertas* eficazes e acionáveis sobre todas as métricas principais, uma prática que facilita para os desenvolvedores mitigarem e resolverem problemas com o microsserviço antes que eles causem interrupções. O componente final é a implementação e a prática de executar um *turno de plantão* sustentável e responsável por monitorar o microsserviço. Com logging, dashboards, alertas e turnos de plantão eficazes, a disponibilidade do microsserviço pode ser protegida: falhas e erros serão detectados, mitigados e resolvidos antes de derrubarem qualquer parte do ecossistema de microsserviços.

Um serviço pronto para produção é adequadamente monitorado quando:

- suas métricas principais são identificadas e monitoradas nos âmbitos de servidor, infraestrutura e microsserviço;
- ele tem uma adequado logging que reflete com precisão os estados passados do microsserviço;
- seus dashboards são fáceis de interpretar e contêm todas as métricas principais;
- seus alertas são acionáveis e definidos por limites sinalizadores;
- há um turno de plantão dedicado responsável por monitorar e responder a quaisquer incidentes e interrupções;
- há um procedimento de plantão claro, bem definido e padronizado para tratar incidentes e interrupções.

Métricas principais

Antes de passarmos para os componentes de um adequado monitoramento, é importante identificar precisamente *o que* queremos e precisamos monitorar: queremos monitorar um microsserviço, mas o que isso

realmente quer dizer? Um microsserviço não é um objeto individual que podemos seguir ou rastrear, ele não pode ser isolado ou mantido em quarentena – é muito mais complicado que isso. Implantado em dezenas, quando não em centenas de servidores, o comportamento de um microsserviço é a soma de seu comportamento em todas estas instâncias, o que não é algo fácil de quantificar. O essencial é identificar quais propriedades de um microsserviço são necessárias e suficientes para descrever seu comportamento e então determinar o que as mudanças nessas propriedades nos dizem sobre o status geral e a saúde do microsserviço. Chamaremos estas propriedades de *métricas principais*.

Há dois tipos de métricas principais: métricas de servidor e infraestrutura, e métricas de microsserviço. Métricas de servidor e infraestrutura são as que dizem respeito ao status da infraestrutura e dos servidores nos quais o microsserviço está sendo executado, enquanto as métricas de microsserviço são métricas exclusivas do microsserviço individual. Em termos do modelo de quatro camadas do ecossistema de microsserviços descrito no Capítulo 1, *Microsserviços*, métricas de servidor e infraestrutura pertencem às camadas 1 a 3, enquanto as métricas de microsserviço pertencem à camada 4.

Separar as métricas principais nestas duas categorias diferentes é importante tanto do ponto de vista organizacional como técnico. Métricas de servidor e infraestrutura em geral afetam mais de um microsserviço: por exemplo, se houver um problema com um servidor particular e o ecossistema de microsserviços compartilhar os recursos de hardware entre vários microsserviços, as métricas de servidor serão relevantes para todas as equipes de microsserviços que tenham um microsserviço implantado naquele servidor. Da mesma forma, métricas específicas do microsserviço raramente serão aplicáveis ou úteis para alguém que não seja a equipe de desenvolvedores trabalhando nesse microsserviço particular. As equipes devem monitorar ambos os tipos de métricas principais (isto é, todas as métricas relevantes para seu microsserviço), e quaisquer métricas relevantes para os vários microsserviços devem ser monitoradas e compartilhadas entre as equipes adequadas.

As métricas de servidor e infraestrutura que devem ser monitoradas para cada microsserviço são a CPU utilizada pelo microsserviço em cada servidor, a RAM utilizada pelo microsserviço em cada servidor, as threads

disponíveis, os descritores de arquivo (FD) abertos do microsserviço e o número de conexões com os databases usados pelo microsserviço. Monitorar essas métricas principais deve ser feito de modo que o status de cada métrica seja acompanhado por informações sobre a infraestrutura e o microsserviço. Isso significa que o monitoramento deve ser granular o suficiente para que os desenvolvedores possam ver as métricas principais de seu microsserviço em um servidor particular e em todos os servidores nos quais ele é executado. Por exemplo, os desenvolvedores devem ser capazes de saber quanta CPU seu microsserviço está usando um servidor particular *e* quanta CPU seu microsserviço está usando em todos os servidores nos quais ele é executado.

Monitorar métricas de servidor quando os recursos são abstraídos

Alguns ecossistemas de microsserviços podem usar aplicações de gerenciamento de cluster (como o Mesos), em que os recursos (CPU, RAM etc.) são abstraídos do âmbito do servidor. As métricas de servidor não estarão disponíveis da mesma forma para os desenvolvedores nessas situações, mas todas as métricas principais do microsserviço devem continuar sendo monitoradas pela equipe do microsserviço.

Determinar as métricas necessárias e suficientes no âmbito do microsserviço é um pouco mais complicado, pois elas dependem da linguagem particular na qual o microsserviço foi escrito. Cada linguagem traz sua forma própria e especial de processar tarefas, por exemplo, e esses recursos específicos da linguagem precisam ser monitorados de perto na maioria dos casos. Considere um serviço Python que utiliza workers uwsgi: o número de workers uwsgi é uma métrica necessária para um adequado monitoramento.

Além de métricas principais específicas da linguagem, também precisamos monitorar a disponibilidade do serviço, o acordo de nível de serviço (SLA) desse serviço, a latência (do serviço como um todo e de seus endpoints de API), o sucesso dos endpoints de API, as respostas e o tempo médio de resposta dos endpoints de API, os serviços (clientes) dos quais as solicitações de API são originadas (junto aos endpoints para os quais eles enviam solicitações), os erros e as exceções (tanto tratados como não tratados) e a saúde e o status das dependências.

E, o mais importante, todas as métricas principais devem ser monitoradas em todo local em que a aplicação estiver implantada. Isso significa que

todo estágio do pipeline de deployment deve ser monitorado. A fase de staging deve ser monitorada de perto para detectar quaisquer problemas antes que uma nova candidata à produção (uma nova versão) seja implantada em servidores que recebam tráfego de produção. É quase desnecessário dizer que todas as implantações em servidores de produção devem ser monitoradas atentamente tanto na fase de pré-release como em produção. (Para mais informações sobre os pipelines de implantação, veja o Capítulo 3, *Estabilidade e confiabilidade*.)

Uma vez identificadas as métricas principais para um microsserviço, o próximo passo é capturar as métricas emitidas por seu serviço. Capture-as, grave seus logs, crie gráficos com elas e gere alertas com base nelas. Nós cobriremos cada um desses passos nas próximas seções.

Resumo das métricas principais

Métricas principais de servidor e infraestrutura:

- CPU
- RAM
- threads
- descritores de arquivo
- conexões com o database

Métricas principais de microsserviço:

- métricas específicas da linguagem
- disponibilidade
- SLA
- latência
- sucesso do endpoint
- respostas do endpoint
- tempos de resposta do endpoint
- clientes
- erros e exceções
- dependências

Logging

Logging é o primeiro componente de um monitoramento pronto para produção. Ela começa no código-base de cada microsserviço e faz parte desse código-base, está contida no código de cada serviço, capturando todas as informações necessárias para descrever o estado do microsserviço. Na verdade, descrever o estado do microsserviço em qualquer momento do passado recente é o objetivo básico do logging.

Um dos benefícios da arquitetura de microsserviço é a liberdade que os desenvolvedores têm de implantar novos recursos e mudanças de código frequentemente, e uma das consequências desta recém-descoberta liberdade do desenvolvedor e da maior velocidade de desenvolvimento é que o microsserviço está sempre mudando. Na maioria dos casos o serviço não será o mesmo de doze horas atrás, muito menos de vários dias atrás, e reproduzir qualquer problema será impossível. Quando nos confrontamos com um problema, muitas vezes, a única maneira de determinar a causa fundamental de um incidente ou interrupção é vasculhar os logs, descobrir o estado do microsserviço no momento da interrupção e descobrir por que o serviço falhou naquele estado. O logging deve ser tal que os desenvolvedores possam determinar a partir dos logs exatamente o que deu errado e onde as coisas começaram a desmoronar.

 Logging sem atribuir versão ao microsserviço

Em geral, atribuir uma versão ao microsserviço não é recomendado, pois isso pode fazer com que outros (cliente) serviços se vinculem a versões específicas de um microsserviço que pode não ser a melhor ou a mais atualizada. Sem atribuir uma versão, pode ser difícil determinar o estado de um microsserviço quando ocorrer uma falha ou interrupção, mas por meio da gravação de logs é possível evitar que isso se torne um problema: se o logging for bom o bastante, o estado de um microsserviço no momento de uma interrupção pode ser suficientemente conhecido e compreendido; a falta de uma versão deixa de ser um obstáculo para uma rápida e eficaz mitigação e a resolução de problemas.

Determinar precisamente *o que* deve ser gravado no log é específico de cada microsserviço. A melhor orientação para determinar o que precisa ser gravado no log é, infelizmente, necessariamente vaga: grave qualquer informação essencial para descrever o estado do serviço em determinado momento. Felizmente podemos limitar quais informações são necessárias restringindo nosso logging a tudo aquilo que pode ser contido no código do serviço. Informações em nível de servidor e de infraestrutura não serão

(e não devem ser) gravadas no log pela própria aplicação, mas pelos serviços e ferramentas que executam a plataforma de aplicação. Algumas métricas principais e informações de microsserviço, como IDs de usuário e detalhes da solicitação e resposta, podem e devem ser localizadas nos logs do microsserviço.

Claro que existem coisas que *nunca devem ser gravadas em log*. Os logs nunca devem conter informações de identificação, como nomes de clientes, números da previdência social e outros dados privados. Eles nunca devem conter informações que possam apresentar um risco à segurança, como senhas, chaves de acesso ou segredos. Na maioria dos casos, mesmo informações aparentemente inócuas como IDs de usuário e nomes de usuário nunca devem ser gravadas em log, exceto quando criptografadas.

Às vezes, gravar logs no âmbito do microsserviço individual não é suficiente. Como temos visto ao longo deste livro, os microsserviços não existem sozinhos, mas dentro de complexas cadeias de clientes e dependências dentro do ecossistema de microsserviços. Embora os desenvolvedores possam fazer seu melhor para gravar logs e monitorar tudo que for importante e relevante para seu serviço, rastrear e gravar logs de solicitações e respostas por toda a cadeia de cliente e dependências fim a fim pode revelar informações importantes sobre o sistema que de outro modo permaneceriam desconhecidas (como a latência total e a disponibilidade da pilha). Para tornar estas informações acessíveis e visíveis, construir um ecossistema de microsserviços pronto para produção requer rastrear cada solicitação por toda a pilha.

O leitor pode ter notado neste ponto que aparentemente muitas informações precisam ser gravadas em log. Logs são dados, e gravar logs é caro: armazená-los é caro, acessá-los é caro, e tanto armazenar como acessar logs têm um custo adicional associado a fazer chamadas de rede caras. O custo de armazenar logs pode não parecer muito para um microsserviço individual, mas se somarmos todas as necessidades de logging de todos os microsserviços dentro de um ecossistema de microsserviços, veremos que seu custo é bastante alto.


Logs e depuração

Evite adicionar logs de depuração no código que será implantado em produção – tais logs são muito caros. Se qualquer log for adicionado especificamente com a finalidade de depuração, os desenvolvedores deverão prestar atenção para garantir que qualquer ramo ou versão contendo esses logs adicionais nunca cheguem ao ambiente de produção.

O logging precisa ser escalável, disponível, facilmente acessível *e* pesquisável. Para manter o custo dos logs baixo e garantir escalabilidade e alta disponibilidade, geralmente é necessário impor cotas de logging por serviço junto a limites e padrões sobre quais informações podem ser gravadas em log, quantos logs cada microsserviço pode armazenar e por quanto tempo os logs serão armazenados antes de ser apagados.

Dashboards

Cada microsserviço deve ter pelo menos um dashboard no qual todas as métricas principais (como utilização de hardware, conexões a databases, disponibilidade, latência, respostas e o status de endpoints de API) são coletadas e exibidas. Um dashboard é uma visualização gráfica que é atualizada em tempo real para refletir todas as mais importantes informações sobre um microsserviço. Dashboards devem ser facilmente acessíveis, centralizados e padronizados em todo o ecossistema de microsserviços.

Dashboards devem ser fáceis de interpretar de modo que alguém de fora possa rapidamente determinar a saúde do microsserviço: qualquer um deve ser capaz de olhar o dashboard e saber imediatamente se o microsserviço está funcionando corretamente ou não. Isso requer encontrar um equilíbrio entre sobrecarregar o usuário com informações (o que tornaria o dashboard praticamente inútil) e não exibir informações suficientes (o que também tornaria o dashboard inútil): apenas o mínimo necessário de informações sobre as métricas principais deve ser exibido.

Um dashboard também serve como um reflexo preciso da qualidade geral do monitoramento de todo o microsserviço. Qualquer métrica principal que gere alertas deve ser incluída no dashboard (trataremos disso na próxima seção): a exclusão de qualquer métrica do dashboard principal refletirá um monitoramento precário do serviço, enquanto a inclusão de métricas que não são necessárias refletirá uma não observância das melhores práticas de alerta (e, consequentemente, do monitoramento).

Há várias exceções à regra contra a inclusão de métricas não essenciais.

Além das métricas principais, informações sobre cada fase do pipeline de deployment devem ser exibidas, embora não necessariamente dentro do mesmo dashboard. Os desenvolvedores trabalhando em microsserviços que requerem monitorar um grande número de métricas principais precisam optar por configurar um dashboard separado para cada fase de implantação (um para staging, um para pré-release e um para produção) para refletir com precisão a saúde do microsserviço em cada fase da implantação: uma vez que diferentes versões estarão sendo executadas ao mesmo tempo nas fases de implantação, refletir com precisão a saúde do microsserviço em um dashboard pode exigir encarar o projeto do dashboard com o objetivo de refletir a saúde do microsserviço em uma fase particular da implantação (tratando-os quase como microsserviços diferentes, ou pelo menos como diferentes instâncias de um microsserviço).

Dashboards e detecção de interrupção

Embora dashboards possam exibir anomalias e tendências negativas das métricas principais de um microsserviço, desenvolvedores nunca devem precisar observar o dashboard de um microsserviço para detectar incidentes e interrupções. Fazê-lo é um antipadrão que leva a deficiências na geração de alertas e no monitoramento geral.

Para ajudar a determinar problemas introduzidos por novas implantações, é útil incluir informações sobre quando ocorreu uma implantação no dashboard. O modo mais eficaz e útil de fazê-lo é garantir que os tempos de implantação sejam exibidos nos gráficos de cada métrica essencial. Isso permite que os desenvolvedores rapidamente verifiquem os gráficos depois de cada implantação para ver se qualquer padrão estranho surgiu em alguma das métricas principais.

Dashboards bem projetados também oferecem aos desenvolvedores uma maneira fácil e visual de detectar anomalias e determinar limites de alerta. Mudanças ou perturbações muito pequenas ou graduais correm o risco de não ser detectadas pela geração de alertas, mas uma olhada atenta em um dashboard preciso pode revelar anomalias que, caso contrário, não seriam detectadas. Limites de alerta, que trataremos na próxima seção, são notoriamente difíceis de determinar, mas podem ser configurados adequadamente quando examinamos dados de histórico no dashboard: os desenvolvedores podem ver padrões normais nas métricas principais, ver

picos nos valores das métricas que ocorreram com as interrupções (ou levaram às interrupções) no passado e então definir limites de acordo.

Alertas

O terceiro componente do monitoramento de um microsserviço pronto para produção são os *alertas* em tempo real. A detecção de falhas, assim como a detecção de mudanças nas métricas principais que possam levar a uma falha, é efetuada por meio de alertas. Para isso, todas as métricas principais – métricas de servidor, métricas de infraestrutura e métricas específicas de microsserviço – devem gerar alertas, e os alertas devem ser configurados em vários limites. Alertas eficazes e acionáveis são essenciais para preservar a disponibilidade de um microsserviço e evitar downtime.

Configurando alertas eficazes

É preciso configurar alertas para todas as métricas principais. Qualquer mudança em uma métrica no âmbito do servidor, de infraestrutura ou microsserviço que possa levar a uma interrupção, causar um pico de latência ou de alguma forma prejudicar a disponibilidade do microsserviço deve disparar um alerta. E, o mais importante, alertas também devem ser disparados sempre que uma métrica essencial *não* for vista.

Todos os alertas devem ser úteis: eles devem ser definidos por adequados limites sinalizadores. Três tipos de limites devem ser configurados para cada métrica com limites superiores e inferiores: *normal*, *advertência* e *crítico*. Limites normais refletem os limites inferiores e superiores adequados e usuais de cada métrica e não devem disparar um alerta. Limites de advertência de cada métrica irão disparar alertas quando houver um desvio da norma que possa levar a um problema com o microsserviço; limites de advertência devem ser configurados de modo que disparem alertas *antes* que qualquer desvio da norma cause uma interrupção ou afete negativamente o microsserviço. Limites críticos devem ser configurados com base em quais limites inferiores e superiores das métricas principais realmente causam uma interrupção, um pico de latência ou prejudicam a disponibilidade de um microsserviço. O ideal é que os limites de advertência disparem alertas que levem a uma detecção, mitigação e resolução rápidas antes que limites críticos sejam atingidos. Em cada

categoria, os limites devem ser altos o suficiente para evitar ruído, mas baixos o suficiente para detectar qualquer problema real com as métricas principais.

 Determinando alertas no início do ciclo de vida de um microsserviço

Sem dados de histórico, pode ser muito difícil configurar limites para as métricas principais. Qualquer limite configurado no início do ciclo de vida de um microsserviço corre o risco de ser inútil ou disparar muitos alertas. Para determinar os limites adequados para um novo microsserviço (ou mesmo um antigo), os desenvolvedores podem executar testes de carga no microsserviço para medir onde devem ficar os limites. Submeter o microsserviço a cargas "normais" de tráfego pode determinar os limites normais, enquanto cargas de tráfego acima do esperado podem ajudar a determinar os limites de advertência e críticos.

Todos os alertas precisam ser acionáveis. Alertas não acionáveis são aqueles que são disparados e então resolvidos (ou ignorados) pelo(s) desenvolvedor(es) de plantão para o microsserviço, pois não são importantes, não são relevantes, não significam que exista algo de errado com o microsserviço, ou alertam sobre um problema que não pode ser resolvido pelo(s) desenvolvedor(es). Qualquer alerta que não possa ser imediatamente tratado pelo(s) desenvolvedor(es) de plantão deve ser removido do pool de alertas, atribuído novamente ao turno de plantão relevante ou (se possível) alterado para se tornar um alerta acionável.

Algumas das métricas principais do microsserviço correm o risco de ser não acionáveis. Por exemplo, alertar sobre a disponibilidade de dependências pode facilmente levar a alertas não acionáveis se interrupções da dependência, aumentos da latência da dependência ou downtime da dependência não requerem que uma ação seja tomada por seu(s) cliente(s). Se nenhuma ação precisa ser tomada, então os limites devem ser configurados adequadamente, ou, em casos mais extremos, nenhum alerta deve ser configurado para as dependências. Entretanto, se alguma ação precisa ser tomada, mesmo algo pequeno como contatar o turno de plantão ou a equipe de desenvolvimento da dependência para alertá-los sobre o problema e/ou coordenar a mitigação e resolução, então um alerta deve ser disparado.

Tratando alertas

Uma vez disparado o alerta, ele precisa ser tratado de forma rápida e

eficaz. A causa fundamental do alerta disparado deve ser mitigada e resolvida. Para tratar alertas de forma rápida e eficaz, há várias medidas que podem ser tomadas.

O primeiro passo é criar instruções para cada alerta conhecido, detalhando como selecionar, mitigar e resolver cada alerta. Estas instruções passo a passo devem fazer parte do roteiro do plantão dentro da documentação centralizada de cada microsserviço, tornando-as facilmente acessíveis para todos que estiverem de plantão para o microsserviço (mais detalhes sobre roteiros podem ser encontrados no Capítulo 7, *Documentação e compreensão*). Roteiros são cruciais para o monitoramento de um microsserviço: eles fornecem a qualquer desenvolvedor de plantão instruções sobre como mitigar e resolver as causas fundamentais de cada alerta. Uma vez que cada alerta está vinculado a um desvio de uma métrica principal, roteiros podem ser escritos para tratar de cada métrica principal, das causas conhecidas dos desvios da norma e de como depurar o problema.

Devem ser criados dois tipos de roteiros de plantão. O primeiro faz parte dos roteiros para alertas de servidor e infraestrutura que devem ser compartilhados entre toda a organização de engenharia de software – eles devem ser escritos abordando todas as principais métricas de servidor e infraestrutura. O segundo tipo faz parte dos roteiros de plantão para microsserviços específicos que contêm instruções referentes aos alertas específicos de microsserviço e que são disparados por mudanças nas métricas principais; por exemplo, um pico de latência deve disparar um alerta, e o roteiro deve conter instruções passo a passo que documentem claramente como depurar, mitigar e resolver picos de latência do microsserviço.

O segundo passo é identificar antipadrões de alerta. Se o turno de plantão do microsserviço estiver sobrecarregado por alertas e mesmo assim o microsserviço aparentemente funcionar como esperado, então quaisquer alertas que forem vistos mais de uma vez, mas puderem ser facilmente mitigados e/ou resolvidos, deverão ser automatizados. Isto é, os passos para mitigação e/ou resolução devem ser incluídos no próprio microsserviço. Isso vale para qualquer alerta, e escrever instruções passo a passo para alertas dentro dos roteiros de plantão permite uma execução eficaz dessa estratégia. Na verdade, qualquer alerta que, uma vez

disparado, requer um simples conjunto de passos para ser mitigado e resolvido pode ser facilmente automatizado. Se for estabelecido este nível de monitoramento pronto para produção, um microsserviço nunca mais sofrerá o mesmo problema duas vezes.

Turnos de plantão

Em um ecossistema de microsserviços, as próprias equipes de desenvolvimento são responsáveis pela disponibilidade de seus microsserviços. No que diz respeito ao monitoramento, isso significa que os desenvolvedores precisam estar de plantão para seus próprios microsserviços. O objetivo de cada desenvolvedor de plantão para um microsserviço deve ser claro: eles devem detectar, mitigar e resolver qualquer problema que ocorra com o microsserviço durante seu turno de plantão antes que o problema cause uma interrupção de seu microsserviço ou impacte o próprio negócio.

Em algumas organizações maiores de engenharia, engenheiros de confiabilidade, DevOps ou outros engenheiros operacionais podem assumir a responsabilidade pelo monitoramento e pelo plantão, mas isso requer que cada microsserviço seja relativamente estável e confiável antes que as responsabilidades do plantão possam ser transferidas para outra equipe. Na maioria dos ecossistemas de microsserviços, os microsserviços raramente atingem este alto nível de estabilidade, pois, como vimos nos capítulos anteriores, microsserviços mudam constantemente. Em um ecossistema de microsserviços, os desenvolvedores devem assumir a responsabilidade de monitorar o código que estiverem implantando.

Elaborar bons turnos de plantão é fundamental e requer o envolvimento de toda a equipe. Para evitar esgotamento, os turnos de plantão devem ser breves e compartilhados: pelo menos dois desenvolvedores devem estar de plantão por vez e os turnos não devem durar mais de uma semana, com intervalos de pelo menos um mês.

Os turnos de plantão de cada microsserviço devem ser divulgados internamente e devem ser facilmente acessíveis. Se uma equipe de microsserviço tiver problemas com uma de suas dependências, eles devem ser capazes de localizar os engenheiros de plantão do microsserviço e contatá-los muito rapidamente. Armazenar as informações em um local

centralizado ajuda a tornar os desenvolvedores mais eficazes em selecionar os problemas e evitar interrupções.

Desenvolver procedimentos padronizados de plantão para toda a organização de engenharia de software ajudará a construir um ecossistema de microsserviço sustentável. Os desenvolvedores devem ser orientados sobre como abordar seus turnos de plantão, sobre as melhores práticas de plantão e devem ser estimulados a juntar-se ao turno de plantão muito rapidamente. Padronizar este processo e deixar as expectativas sobre o plantão completamente claras para todos os desenvolvedores evitará o esgotamento, a confusão e a frustação que geralmente acompanham qualquer menção de entrar em um turno de plantão.

Avalie seu microsserviço

Agora que você tem um melhor entendimento sobre monitoramento, use a seguinte lista de perguntas para avaliar a disponibilidade de produção de seu(s) microsserviço(s) e do ecossistema de microsserviços. As perguntas são organizadas por tópico e correspondem às seções dentro deste capítulo.

Métricas principais

- Quais são as métricas principais deste microsserviço?
- Quais são as métricas de servidor e infraestrutura?
- Quais são as métricas no âmbito de microsserviço?
- Todas as métricas principais do microsserviço são monitoradas?

Logging

- Quais informações este microsserviço precisa gravar em log?
- Este microsserviço grava em log todas as solicitações importantes?
- O logging reflete o estado do microsserviço em qualquer momento?
- Esta solução de logging é escalável e eficaz em termos de custo?

Dashboards

- Este microsserviço tem um dashboard?
- O dashboard é fácil de interpretar? Todas as métricas principais são

exibidas no dashboard?

- Posso determinar se este microsserviço está funcionando corretamente ou não olhando o dashboard?

Alertas

- Existe um alerta para todas as métricas principais?
- Todos os alertas são definidos por adequados limites sinalizadores?
- Os limites de alerta são configurados adequadamente para disparar um alerta antes que ocorra uma interrupção?
- Todos os alertas são acionáveis?
- Há instruções passo a passo para triagem, mitigação e resolução para cada alerta no roteiro de plantão?

Turnos de plantão

- Há um turno de plantão dedicado e responsável por monitorar este microsserviço?
- Cada turno de plantão contém pelo menos dois desenvolvedores?
- Existem procedimentos padronizados de plantão em toda a organização de engenharia de software?

CAPÍTULO 7

Documentação e compreensão

Um microsserviço pronto para produção é documentado e compreendido. A documentação e a compreensão organizacional aumentam a velocidade do desenvolvedor e ao mesmo tempo mitigam duas das mais significativas contrapartidas que acompanham a adoção da arquitetura de microsserviços: dispersão organizacional e defasagem técnica. Este capítulo explora os elementos essenciais da documentação e compreensão de um microsserviço, incluindo como construir uma documentação abrangente e útil, como aumentar a compreensão de um microsserviço em todos os níveis do ecossistema de microsserviços e como implementar disponibilidade de produção por toda organização de engenharia de software.

Princípios de documentação e compreensão de um microsserviço

Abrirei este capítulo final sobre o último princípio de padronização de microsserviços com uma famosa história da literatura russa. Embora pareça pouco ortodoxo citar Dostoiévski em um livro sobre arquitetura de software, o personagem Gruchénka em *Os irmãos Karamazov* captura perfeitamente o que acredito ser a essência da documentação e compreensão de microsserviços: “Saiba uma coisa, Rakitka, por pior que eu seja, até eu dei uma cebola.”.

Minha parte favorita do brilhante romance de Dostoiévski é uma história contada pelo personagem Gruchénka sobre uma velha senhora e uma cebola. A história é mais ou menos assim: havia uma senhora amargurada que era muito egoísta e cruel. Certo dia, ela encontrou um mendigo e por alguma razão sentiu muita pena dele. Ela queria dar algo ao mendigo, mas tudo que tinha era uma cebola, portanto ela deu a cebola ao mendigo. A velha morreu e, graças à sua amargura e crueldade, acabou indo parar no

inferno. Depois de ter sofrido um bom tempo, um anjo veio salvá-la, pois Deus havia se lembrado de seu único gesto generoso em toda a sua vida e queria retribuir com a mesma generosidade. O anjo aproximou-se dela segurando uma cebola. A velha agarrou a cebola, mas, para seu desalento, os outros pecadores ao lado dela também tentaram pegar a cebola. Sua natureza insensível e cruel aflorou e ela tentou afastá-los, pois não queria que nenhum deles ficasse com pedaço algum da cebola e, infelizmente, enquanto tentava tirar a cebola deles, a cebola partiu-se em muitas camadas e ela e os outros pecadores foram mandados de volta ao inferno.

Não é a história mais reconfortante, mas existe uma moral nessa história de Gruchénka que acho que se aplica incrivelmente à prática de documentação de microsserviços: sempre dê uma cebola.

A importância de uma documentação completa e atualizada para cada microsserviço não pode ser enfatizada o suficiente. Pergunte aos desenvolvedores que trabalham com ecossistema de microsserviços quais são suas principais preocupações, e eles recitarão uma lista de recursos que ainda serão implementados, de erros a ser corrigidos, de dependências que estão causando problemas e de coisas que eles não entendem sobre seu próprio serviço e as dependências das quais ele depende. Quando pedimos para dar mais detalhes sobre os dois últimos itens, eles tendem a dar respostas parecidas: eles não entendem como funciona, é uma caixa preta, e a documentação é completamente inútil.

Uma documentação de baixa qualidade das dependências e ferramentas internas retarda o trabalho dos desenvolvedores e afeta sua capacidade de tornar seus próprios serviços prontos para produção. Ela impede que eles usem dependências e ferramentas internas corretamente e desperdiça inúmeras horas de engenharia, pois às vezes a única maneira de descobrir o que um serviço ou uma ferramenta faz (sem uma documentação adequada) é fazer engenharia reversa até entender seu funcionamento.

Uma documentação de baixa qualidade de um serviço também prejudica a produtividade dos desenvolvedores que estão contribuindo com ele. Por exemplo, a falta de roteiros para um turno de plantão significa que quem estiver de plantão terá de solucionar cada problema desde o início toda vez. Sem um guia de bordo, cada novo desenvolvedor que estiver trabalhando no serviço precisará começar do zero para entender como funciona o serviço. Pontos únicos de falha e problemas com o serviço

passarão despercebidos até que causem uma interrupção. Novos recursos adicionados ao serviço frequentemente não terão uma ideia geral de como o serviço realmente funciona.

O objetivo de uma boa documentação pronta para produção é criar e manter um repositório central de conhecimento sobre o serviço. Compartilhar esse conhecimento tem dois componentes: os fatos objetivos sobre o serviço e a compreensão organizacional do que o serviço faz e onde ele se encaixa na organização como um todo. O problema de uma documentação de baixa qualidade pode ser dividido em dois subproblemas: falta de documentação (os fatos) e falta de compreensão. Resolver esses dois subproblemas requer padronizar a documentação para todos os microsserviços e criar estruturas organizacionais para compartilhar o conhecimento sobre o microsserviço.

A história de Gruchénka é a regra de ouro da documentação de microsserviços: sempre dê uma cebola. Dê uma cebola para o seu próprio bem, para o bem dos colegas desenvolvedores que estão trabalhando em seu serviço e para o bem dos desenvolvedores cujos serviços dependem do seu.

Um serviço pronto para produção é documentado e compreendido quando:

- ele tem uma documentação abrangente.
- sua documentação é atualizada regularmente.
- sua documentação contém uma descrição do microsserviço; um diagrama da arquitetura; informações de contato e de plantão; links para informações importantes; um guia de bordo e desenvolvimento; informações sobre o(s) fluxo(s) de solicitação, os endpoints e as dependências do serviço; um roteiro de plantão; e respostas para perguntas mais frequentes.
- ele é bem compreendido nos âmbitos do desenvolvedor, da equipe e da organização.
- ele mantém um conjunto de padrões de disponibilidade de produção e cumpre os requisitos associados.
- sua arquitetura é revista e auditada frequentemente.

Documentação de microsserviços

A documentação de todos os microsserviços em uma organização de engenharia de software deve ser armazenada em um local centralizado, compartilhado e de fácil acesso. Qualquer desenvolvedor de qualquer equipe deve ser capaz de encontrar a documentação de cada microsserviço sem dificuldade. Um site interno contendo a documentação de todos os microsserviços e das ferramentas internas tende a ser o melhor meio para isso.

Arquivos README e comentários no código não são documentação

Muitos desenvolvedores limitam a documentação de seus microsserviços a um arquivo README em seu repositório ou comentários espalhados ao longo do código. Embora um arquivo README seja essencial e todo código de microsserviço deva conter comentários adequados, isso não é documentação pronta para produção, e exige que os desenvolvedores vasculhem o código à procura de informações. Uma documentação adequada é armazenada em um local centralizado (como um site), onde se encontra a documentação de todos os microsserviços da empresa de engenharia.

A documentação deve ser atualizada regularmente. Sempre que uma mudança significativa for feita no serviço, a documentação deverá ser atualizada. Por exemplo, se um novo endpoint de API for adicionado, as informações sobre o endpoint também precisarão ser acrescentadas à documentação. Se um novo alerta for adicionado, então, instruções passo a passo sobre como selecionar, mitigar e resolver o alerta também deverão ser acrescentadas ao roteiro de plantão do serviço. Se uma nova dependência for adicionada, então, informações sobre essa dependência deverão ser acrescentadas à documentação. Sempre dê uma cebola.

A melhor maneira de fazê-lo é tornar o processo de atualizar a documentação parte do fluxo de trabalho de desenvolvimento. Se a atualização da documentação for considerada uma tarefa separada do (e secundária ao) desenvolvimento, então ela nunca será feita e se tornará parte da defasagem técnica do serviço. Para reduzir a defasagem técnica, os desenvolvedores devem ser encorajados (ou, se necessário, obrigados) a acompanhar toda mudança significativa de código com uma atualização da documentação.

 Tornar a atualização da documentação parte do ciclo de desenvolvimento

Se atualizar e melhorar a documentação forem consideradas tarefas secundárias em relação a escrever código, elas geralmente serão abandonadas e se tornarão parte da defasagem técnica do serviço. Para evitar isso, torne as atualizações e melhorias da documentação uma parte necessária do ciclo de desenvolvimento do serviço.

A documentação deve ser abrangente e útil. Ela deve conter todos os fatos relevantes e importantes sobre o serviço. Depois de ler a documentação, um desenvolvedor deve saber como desenvolver e contribuir para o serviço e a arquitetura do serviço, o contato e as informações do plantão para o serviço; como o serviço funciona (fluxos de solicitações, endpoints, dependências etc.); como selecionar, mitigar e corrigir incidentes e interrupções, assim como resolver alertas gerados pelo serviço; e ter respostas para perguntas mais frequentes sobre o serviço.

E, o mais importante, a documentação deve ser escrita de forma clara e fácil de entender. Uma documentação cheia de jargões é inútil, uma documentação excessivamente técnica e que não explica os aspectos que podem ser exclusivos do serviço também é inútil, assim como a documentação que não entra em nenhum detalhe significativo. O objetivo de escrever uma documentação boa e clara é que ela possa ser compreendida por qualquer desenvolvedor, gerente, gerente de produtos ou executivo dentro da empresa.

Vamos nos aprofundar um pouco mais em cada um dos elementos de uma documentação de microsserviço pronto para produção.

Descrição

A documentação de cada microsserviço deve começar com uma *descrição* do serviço. Ela deve ser curta, cativante e objetiva. Por exemplo, se houver um microsserviço chamado *emissor-de-recibo* cuja finalidade seja enviar um recibo depois que o cliente concluir um pedido, a descrição deve ser:

> Descrição
>
> Depois que um cliente faz um pedido, o emissor-de-recibo envia um recibo ao cliente via e-mail.

Isso é essencial, pois garante que qualquer pessoa que encontrar a documentação saberá qual o papel desempenhado pelo microsserviço no

ecossistema de microsserviços.

Diagrama da arquitetura

A descrição do serviço deve ser seguida de um *diagrama da arquitetura*. Este diagrama deve detalhar a arquitetura do serviço, incluindo seus componentes, seus endpoints, o fluxo de solicitações, suas dependências (tanto upstream como downstream) e informações sobre quaisquer databases ou caches. Veja um exemplo de diagrama de arquitetura na Figura 7.1.

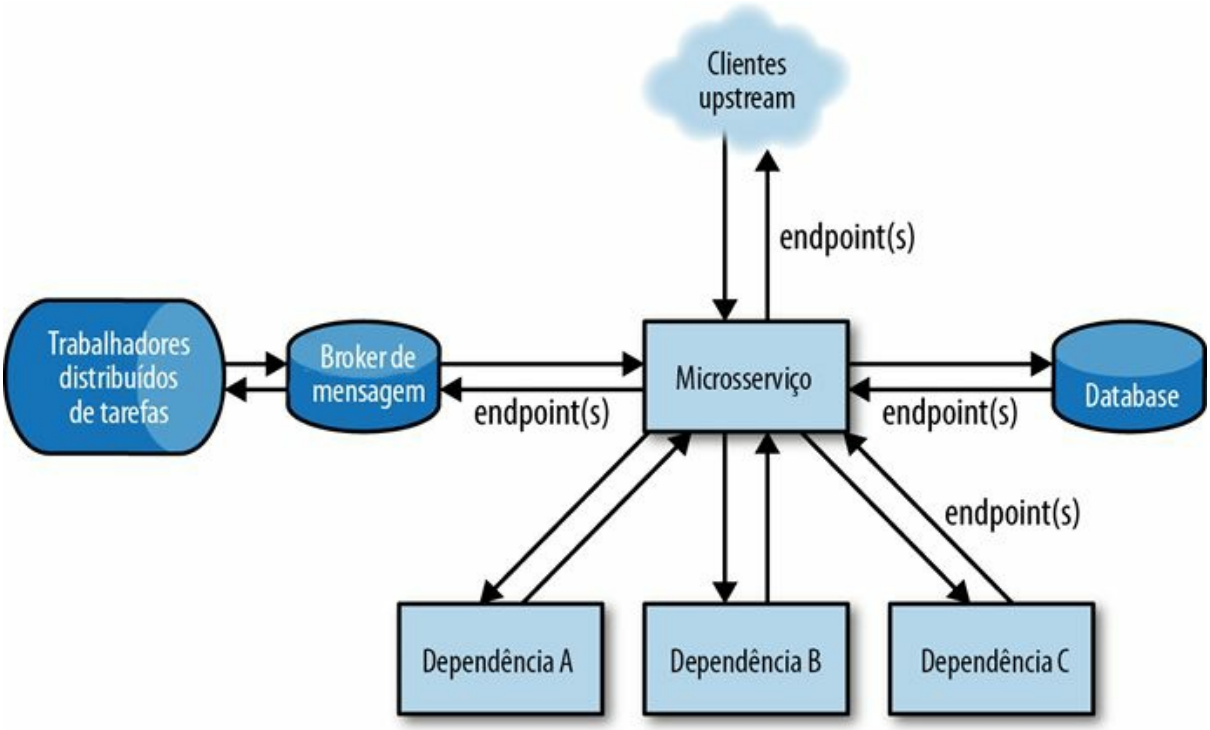


Figura 7.1 – Exemplo de um diagrama de arquitetura de microsserviço.

Diagramas de arquitetura são essenciais por vários motivos. É quase impossível entender como e por que um microsserviço funciona apenas lendo o código, portanto um diagrama de arquitetura bem desenhado é uma descrição visual e um resumo de fácil entendimento do microsserviço. Esses diagramas também ajudam desenvolvedores a acrescentar novos recursos, isolando o funcionamento interno do serviço de modo que eles possam ver onde e como os novos recursos irão (ou não) se encaixar. E, o mais importante, eles revelam questões e problemas com o serviço que passariam despercebidos sem uma representação visual completa de sua arquitetura: é difícil descobrir os pontos de falha de um serviço apenas percorrendo as linhas de código, mas eles tendem a se destacar claramente

em um diagrama de arquitetura preciso.

Informações de contato e plantão

A probabilidade é que alguém olhando a documentação de um serviço seja um membro da equipe do serviço ou alguém de uma equipe diferente que esteja com problemas relacionados ao serviço ou queira saber como ele funciona. Para os desenvolvedores do segundo grupo, ter acesso às informações sobre a equipe é útil e necessário, portanto muitos fatos importantes devem ser incluídos na seção *informações de contato e plantão* dentro da documentação.

Essa seção deve conter os nomes, os cargos e as informações de contato de todos na equipe (incluindo colaboradores individuais, gerentes e gerentes de programa/produto). Assim fica fácil para os desenvolvedores em outras equipes determinar rapidamente quem eles devem contatar se tiverem um problema com o serviço ou uma dúvida relativa a ele. Essas informações são úteis, por exemplo, quando um desenvolvedor tem problemas relativos a uma de suas dependências: saber quem contatar e qual o seu papel na equipe torna a comunicação entre equipes fácil e eficiente.

Adicionar informações sobre o turno de plantão (e mantê-las atualizadas para refletir quem está de plantão para o serviço em qualquer momento) garante que as pessoas saibam exatamente quem contatar em caso de problemas gerais ou emergências: o engenheiro que está de plantão para o serviço.

Links

A documentação precisa ser um recurso centralizado para todas as informações sobre um microsserviço. Para isso ser efetivo, a documentação precisa conter links para o repositório (para que os desenvolvedores possam facilmente conferir o código), um link para o dashboard, um link para a RFC original do microsserviço e um link para os mais recentes slides de revisão da arquitetura. Quaisquer informações extras sobre outros microsserviços, tecnologias usadas pelo microsserviço etc., que possam ser úteis para o desenvolvedor devem ser incluídas na seção *links* da documentação.

Guia de bordo e desenvolvimento

A finalidade da seção *guia de bordo e desenvolvimento* é ajudar um novo desenvolvedor a entrar na equipe, começar a contribuir com código, adicionar recursos ao microsserviço e introduzir novas alterações no pipeline de deployment.

A primeira parte dessa seção deve ser um guia passo a passo para configurar o serviço. Ele deve mostrar a um desenvolvedor como conferir o código, configurar o ambiente, iniciar o serviço e verificar que o serviço está funcionando corretamente (incluindo todos os comandos ou scripts que precisam ser executados para realizar a tarefa).

A segunda parte deve guiar o desenvolvedor pelo ciclo de desenvolvimento e pipeline de deployment do serviço (detalhes sobre um ciclo de desenvolvimento e pipeline de deployment prontos para produção podem ser encontrados nas seções “O ciclo de desenvolvimento” e “O pipeline de deployment”). Devem ser incluídos os detalhes técnicos (por exemplo, comandos que precisam ser executados, junto a vários exemplos) de cada um dos passos: como conferir o código, como fazer uma alteração no código, como escrever um teste de unidade para a alteração (se necessário), como executar os testes necessários, como entregar suas alterações, como enviar as alterações para a revisão de código, como garantir que o serviço seja construído e liberado corretamente e como implantá-lo (como o pipeline de deployment é configurado para o serviço).

Fluxos de solicitações, endpoints e dependências

A documentação também deve conter informações críticas sobre *fluxos de solicitações, endpoints e dependências* do microsserviço.

A documentação do fluxo de solicitações pode conter um diagrama dos fluxos de solicitações da aplicação. Isso pode ser o diagrama de arquitetura, se o fluxo de solicitações for detalhado adequadamente dentro do diagrama de arquitetura. Qualquer diagrama deve ser acompanhado por uma descrição qualitativa dos tipos de solicitações que são feitas para o microsserviço e de como elas são tratadas.

Este é também o lugar para documentar todos os endpoints de API do serviço. Geralmente é suficiente uma lista dos endpoints com seus nomes e uma descrição qualitativa de cada um, junto a suas respostas. Esta deve ser clara e compreensível o suficiente para que outro desenvolvedor que esteja trabalhando em uma equipe diferente possa ler as descrições dos endpoints

de API de seu serviço e tratar seu microsserviço como uma caixa preta, acessando os endpoints com sucesso e recebendo as respostas esperadas.

O terceiro elemento desta seção são as informações sobre as dependências do serviço. Relacione as dependências, os endpoints relevantes dessas dependências e quaisquer solicitações que o serviço faça a elas, junto das informações sobre seus SLAs, de quaisquer alternativas/caches/backups em caso de falha e dos links para sua documentação e seus dashboards.

Roteiros de plantão

Como abordamos no Capítulo 6, *Monitoramento*, todo alerta deve ser incluído em um roteiro de plantão e acompanhado de instruções passo a passo descrevendo como ele deve ser selecionado, mitigado e resolvido. O roteiro de plantão deve ser mantido na documentação centralizada do serviço em uma seção chamada *roteiro de plantão*, junto a orientações gerais e detalhadas sobre soluções de problemas e depuração de novos erros.

Um bom roteiro começa com requisitos e procedimentos gerais do plantão, e então apresenta uma lista completa dos alertas do serviço. Para cada alerta, o roteiro de plantão deve incluir o nome do alerta, uma descrição do alerta, uma descrição do problema e um guia passo a passo sobre como selecionar o alerta, mitigá-lo e resolvê-lo. Ele também descreve quaisquer implicações organizacionais do alerta: a gravidade do problema, se o alerta significa uma interrupção ou não, e informações sobre como comunicar quaisquer incidentes e interrupções para a equipe e, se necessário, para o restante da organização de engenharia de software.

Escreva roteiros de plantão que desenvolvedores sonolentos possam entender às duas da madrugada

Os desenvolvedores de plantão para um serviço podem (ou, mais realisticamente, serão) acionados a qualquer hora do dia, incluindo tarde da noite ou muito cedo pela manhã. Escreva seus roteiros de plantão de forma que um desenvolvedor sonolento seja capaz de acompanhar sem dificuldades.

Escrever roteiros bons, claros e facilmente compreensíveis é extremamente importante. Eles devem ser escritos de modo que qualquer desenvolvedor de plantão para o serviço ou que tenha problemas com o serviço seja capaz

de agir rapidamente, diagnosticar o problema, mitigar o incidente e resolvê-lo, tudo isso em pouquíssimo tempo para manter o downtime do serviço muito baixo.

Nem todo alerta será facilmente mitigado ou resolvido, e a maioria das interrupções (exceto aquelas causadas por erros no código introduzidos por uma implantação recente) não terá sido observada antes. Para permitir que os desenvolvedores tratem esses problemas com sabedoria, adicione uma seção geral *solução de problemas e depuração* ao roteiro de plantão na documentação que contém dicas sobre como abordar novos problemas de forma estratégica e metódica.

FAQ

Um elemento frequentemente esquecido da documentação é uma seção dedicada a responder perguntas comuns sobre o serviço. Com uma seção de "perguntas mais frequentes", quem estiver de plantão e, consequentemente, o restante da equipe não terão o trabalho de responder a perguntas mais comuns.

Existem duas categorias de perguntas que devem ser respondidas aqui. A primeira envolve as perguntas que os desenvolvedores em outras equipes fazem sobre o serviço. A maneira de saber se essas perguntas devem ser incluídas numa FAQ é simples: se alguém fizer uma pergunta e você achar que ela pode ser feita novamente, acrescente-a à lista FAQ. A segunda categoria engloba as perguntas que vêm dos membros da equipe, e a mesma abordagem pode ser adotada: se houver uma pergunta sobre como ou por que ou quando fazer algo relacionado ao serviço, acrescente-a à lista FAQ.

Resumo: elementos da documentação de microsserviços prontos para produção

A documentação de microsserviços prontos para produção inclui:

- uma descrição do microsserviço e de seu lugar no ecossistema de microsserviços e no negócio;
- um diagrama de arquitetura detalhando a arquitetura do serviço, seus clientes e dependências em um alto nível de abstração;
- informações de contato e do plantão sobre a equipe de

desenvolvimento do microsserviço;

- links para o repositório, os dashboards, a RFC para o serviço, revisões de arquitetura e outras informações relevantes ou úteis;
- um guia de bordo e desenvolvimento contendo detalhes sobre o processo de desenvolvimento, o pipeline de deployment e quaisquer outras informações que serão úteis para os desenvolvedores que contribuem com código para o serviço;
- informações detalhadas sobre os fluxos de solicitações, o SLA, o status de disponibilidade de produção, os endpoints de API, importantes clientes e dependências do microsserviço;
- um roteiro de plantão contendo procedimentos gerais de resposta a incidentes e interrupções, instruções passo a passo sobre como selecionar, mitigar e resolver cada alerta e uma seção geral de solução de problemas e depuração;
- uma seção FAQ (“perguntas mais frequentes”).

Compreensão do microsserviço

Uma documentação centralizada, atualizada e completa é apenas uma parte da documentação e da compreensão de microsserviços prontos para produção. Além de escrever e atualizar a documentação, é preciso criar processos organizacionais para garantir que os microsserviços sejam bem compreendidos não apenas pelas equipes individuais de desenvolvimento, mas também pela organização como um todo. De várias maneiras, um microsserviço bem compreendido é aquele que atende a todos os requisitos de disponibilidade de produção.

Compreender o microsserviço é realmente indispensável para o desenvolvedor, para a equipe e a organização. Embora a noção de “compreender” um microsserviço possa parecer muito vaga para ser útil à primeira vista, o conceito de um microsserviço pronto para produção pode ser usado para orientar e definir a compreensão de um microsserviço em todos os níveis. Equipados com padrões e requisitos de disponibilidade de produção, junto a uma compreensão realista da complexidade organizacional e dos desafios que a arquitetura de microsserviço traz, os desenvolvedores podem quantificar sua compreensão de cada microsserviço e (como eu insisti com o leitor no início deste capítulo)

oferecer uma cebola para o restante da empresa.

Para o desenvolvedor individual isso se traduz em ser capaz de responder a perguntas sobre seu microsserviço. Por exemplo, quando perguntado se seu microsserviço é escalável, ele poderá consultar uma lista de requisitos de escalabilidade e responder com confiança “sim”, “não” ou algo entre essas duas respostas (por exemplo, “ele atende aos requisitos *x* e *z*, mas *y* ainda não foi implementado”). Da mesma forma, quando perguntado se seu microsserviço é tolerante a falhas, ele poderá falar sem parar sobre todos os cenários de falha e as possíveis catástrofes, então explicar em detalhes como ele está preparado para elas usando vários tipos de teste de resiliência.

No âmbito da equipe, *compreender* significa que a equipe tem consciência do estado de seu microsserviço em termos de disponibilidade de produção e o que precisa ser feito para levar seu serviço a um estado de pronto para produção. Isso precisa ser um elemento cultural de cada equipe para ser bem-sucedida: padrões e requisitos de disponibilidade de produção precisam orientar as decisões tomadas pela equipe e ser encarados não apenas como itens a ser ticados em uma lista, mas como princípios que orientam a equipe no sentido de construir o melhor microsserviço possível.

A compreensão precisa ser incorporada ao tecido da própria organização. Isso requer que padrões e requisitos de disponibilidade de produção façam parte do processo organizacional. Antes de um serviço ser construído e uma *solicitação de comentários* (RFC) ser enviada para revisão, o serviço pode ser avaliado com base em padrões e requisitos de disponibilidade de produção. Os desenvolvedores, projetistas e engenheiros de operação podem garantir que o serviço seja construído para ser estável, confiável, escalável, de alto desempenho, tolerante a falhas, preparado para catástrofes, corretamente monitorado e adequadamente documentado e compreendido antes mesmo de começar a ser executado – garantindo que assim que o novo serviço começar a receber tráfego de produção ele tenha sido projetado e otimizado para disponibilidade e seja confiável em termos de tráfego de produção.

Não é suficiente apenas revisar e projetar para disponibilidade de produção no início do ciclo de vida de um microsserviço. Os serviços existentes precisam ser revisados e auditados constantemente para que a qualidade de cada microsserviço seja mantida em um nível suficientemente elevado,

garantindo alta disponibilidade e confiança entre as várias equipes de microsserviço e todo o ecossistema de microsserviços. Automatizar essas auditorias de disponibilidade de produção dos serviços existentes e divulgar internamente os resultados pode ajudar a estabelecer por toda a organização uma percepção sobre a qualidade do ecossistema de microsserviços como um todo.

Revisões de arquitetura

Algo que aprendi depois de implantar esses padrões e seus requisitos de disponibilidade de produção em mais de mil diferentes microsserviços e com suas equipes de desenvolvimento é que o modo mais imediato e eficaz de compreender o microsserviço é organizar *revisões de arquitetura* programadas para cada microsserviço. Uma boa revisão de arquitetura requer uma reunião com os desenvolvedores e engenheiros de confiabilidade (ou outros engenheiros operacionais) que estejam trabalhando no serviço, em que eles se encontrem em uma sala, desenhem a arquitetura do serviço em um quadro branco e avaliem cuidadosamente essa arquitetura.

Passados vários minutos desse exercício, tende a ficar muito claro qual é exatamente o escopo da compreensão nos âmbitos de desenvolvedor e equipe. Ao discutirem a arquitetura, os desenvolvedores imediatamente devem descobrir gargalos de escalabilidade e desempenho, pontos de falha anteriormente ocultos, possíveis interrupções e futuros incidentes, cenários de falha e catástrofes, e novos recursos que devem ser adicionados. Decisões ruins de arquitetura que foram tomadas no passado ficarão óbvias, e antigas tecnologias que devam ser substituídas por novas e melhores também se destacarão. Para garantir que a avaliação e a discussão sejam produtivas e objetivas, é útil chamar desenvolvedores de outras equipes (especialmente das áreas de infraestrutura, DevOps ou engenharia de confiabilidade) com experiência em arquitetura de sistemas distribuídos de grande porte (e o ecossistema de microsserviços específico da organização) para que possam apontar problemas que os desenvolvedores talvez não notem.

Cada reunião deve produzir um novo e atualizado diagrama de arquitetura para o serviço, junto a uma lista de projetos nos quais trabalhar nas semanas e nos meses seguintes. O novo diagrama deve ser definitivamente

acrescentado à documentação, e os projetos podem ser incluídos em cada *plano estratégico* (veja “Planos estratégicos de disponibilidade de produção”) e *objetivos e resultados-chave* (OKRs) do serviço.

Considerando que o desenvolvimento de microsserviços avança rapidamente, os microsserviços evoluem em um ritmo acelerado e as camadas inferiores do ecossistema de microsserviços estão mudando constantemente. Para manter a arquitetura e sua compreensão relevantes e produtivas, estas reuniões devem ser realizadas regularmente. Descobri que uma boa regra é programá-las de modo que elas se alinhem com o OKR e o planejamento de projetos. Se projetos e OKRs forem planejados e programados trimestralmente, então as revisões trimestrais de arquitetura deverão ser realizadas a cada trimestre antes do início do ciclo de planejamento.

Auditorias de disponibilidade de produção

Para garantir que um microsserviço cumpra os padrões e requisitos de disponibilidade de produção e esteja realmente pronto para produção, a equipe pode realizar uma *auditoria de disponibilidade de produção* no serviço. Fazer uma auditoria é simples: a equipe se reúne com uma checklist de requisitos de disponibilidade de produção e verifica se o seu serviço cumpre cada requisito ou não. Isso permite compreender um serviço: cada desenvolvedor e cada equipe saberão, ao final da auditoria, qual o estado exato de seu serviço e o que pode ser melhorado.

A estrutura de uma auditoria deve espelhar os padrões e os requisitos de disponibilidade de produção que a organização de engenharia de software adotou. A equipe deve usar as auditorias para quantificar a estabilidade, a confiabilidade, a escalabilidade, a tolerância a falhas, a preparação para catástrofes, o desempenho, o monitoramento e a documentação do serviço. Como descrevi nos capítulos anteriores, cada um desses padrões é acompanhado por um conjunto de requisitos que podem ser usados para fazer com que cada serviço atenda a esses padrões – os desenvolvedores podem ajustar esses requisitos de cada padrão de disponibilidade de produção para que eles atendam às necessidades e aos objetivos da organização. Os requisitos exatos dependerão dos detalhes do ecossistema de microsserviços da empresa, mas os padrões e seus componentes básicos são relevantes para todo ecossistema (veja o Apêndice A para um resumo

de checklist contendo os padrões de disponibilidade de produção e seus requisitos gerais).

Planos estratégicos de disponibilidade de produção

Assim que a equipe de desenvolvimento de um microsserviço tiver concluído uma auditoria completa de disponibilidade de produção de seu microsserviço e a equipe souber se seu serviço está pronto para produção, o próximo passo é planejar como levar o serviço para um estado de pronto para produção. As auditorias facilitam esta tarefa: neste ponto, a equipe tem uma checklist de quais requisitos de disponibilidade de produção não são cumpridos por seu serviço, então basta atender a cada um desses requisitos não satisfeitos.

É neste momento que *planos estratégicos de disponibilidade de produção* podem ser desenvolvidos. Percebi que eles são um item extremamente útil do processo de disponibilidade de produção e compreensão do microsserviço. Cada microsserviço é diferente, e os detalhes de implementação de cada requisito não satisfeito irão variar entre os serviços, portanto produzir um plano estratégico detalhado que documente todos os detalhes de implementação orientará a equipe no sentido de tornar seu microsserviço pronto para produção. Os requisitos que precisam ser atendidos podem ser acompanhados dos detalhes técnicos, dos problemas que surgirem (interrupções e incidentes) e que estiverem relacionados ao requisito, de um link para algum ticket em um sistema de gerenciamento de tarefas e do(s) nome(s) do(s) desenvolvedor(es) que trabalharão no projeto.

O plano estratégico e a lista de requisitos não satisfeitos de disponibilidade de produção que ele contém podem se tornar parte de qualquer futuro planejamento e (se usado na empresa) dos OKRs para o serviço. Atender aos requisitos de disponibilidade de produção funciona melhor quando o processo é acompanhado do desenvolvimento de recursos e da adoção de novas tecnologias. Tornar cada serviço no ecossistema de microsserviços estável, confiável, escalável, de alto desempenho, tolerante a falhas, preparado para catástrofes, monitorado, documentado e compreendido é uma maneira direta e quantificável de garantir que cada serviço esteja realmente pronto para produção e garantir a disponibilidade de todo o ecossistema de microsserviços.

Automatização da disponibilidade de produção

Revisões de arquitetura, auditorias e planos estratégicos resolvem os desafios de compreender um microsserviço nos âmbitos de desenvolvedor e de equipe, mas a compreensão no âmbito organizacional requer um componente adicional. Da forma como apresentei tudo até agora, todo o trabalho envolvido na construção de um microsserviço pronto para produção é em grande parte manual, exigindo que os desenvolvedores sigam individualmente cada passo da auditoria, preparem tarefas, listas e planos estratégicos e verifiquem o cumprimento de cada requisito individual. Trabalho manual desse tipo geralmente é postergado junto ao restante da defasagem técnica, mesmo nas equipes mais produtivas e motivadas para disponibilidade de produção.

Um dos princípios-chave da engenharia de software na prática é: se você tem de fazer algo manualmente mais de uma vez, automatize essa tarefa para que você nunca precise realizá-la novamente. Isso se aplica ao trabalho operacional, se aplica a qualquer situação pontual e não recorrente, e se aplica a qualquer coisa que você precise digitar em um terminal e, o que não é nenhuma surpresa, se aplica a fazer cumprir padrões de disponibilidade de produção em uma organização de engenharia de software. A automatização é a melhor cebola que você pode oferecer às suas equipes de desenvolvimento.

É fácil elaborar uma lista de requisitos de disponibilidade de produção para cada microsserviço. Eu mesma o fiz no Uber, eu vi outros desenvolvedores implementarem os mesmos padrões de disponibilidade de produção deste livro em suas empresas e criei um modelo de checklist (Apêndice A, *Checklist de disponibilidade de produção*) que você, leitor, pode usar. Uma lista como esta torna a automatização da checklist bastante fácil. Por exemplo, para verificar a tolerância a falhas e a preparação para catástrofes, você pode executar verificações automatizadas para garantir que os adequados testes de resiliência estejam prontos e sendo executados, e que cada microsserviço passe nos testes com distinção.

A dificuldade para automatizar cada uma dessas verificações de disponibilidade de produção dependerá totalmente da complexidade de seus serviços internos dentro de cada camada do ecossistema de microsserviços. Se todos os microsserviços e as ferramentas do tipo autosserviço tiverem APIs decentes, a automatização será muito fácil. Se

os seus serviços tiverem problemas de comunicação ou se alguma ferramenta interna do tipo autosserviço for muito exigente ou mal escrita, você terá dificuldades (e não apenas com a disponibilidade de produção, mas com a integridade de seu serviço e de todo o ecossistema de microsserviços).

Automatizar a disponibilidade de produção aumenta a compreensão organizacional de muitas maneiras extremamente importantes e eficazes. Se você automatizar estas verificações e executá-las constantemente, as equipes na organização sempre saberão o estado de cada microsserviço. Divulgue estes resultados internamente, dê a cada microsserviço uma nota de disponibilidade de produção, que mede o quão pronto para produção seu serviço está, exija que serviços críticos para o negócio tenham uma nota mínima alta de disponibilidade de produção e confine as implantações. A disponibilidade de produção pode fazer parte da cultura de engenharia, e este é um modo infalível de conseguir isso.

Avalie seu microsserviço

Agora que você tem mais entendimento a respeito de documentação, use a lista de perguntas apresentada a seguir para avaliar a disponibilidade de produção de seu(s) microsserviço(s) e do ecossistema de microsserviços. As perguntas são organizadas por tópicos e correspondem às seções dentro deste capítulo.

Documentação do microsserviço

- A documentação de todos os microsserviços está armazenada em um local centralizado, compartilhado e de fácil acesso?
- É fácil pesquisar a documentação?
- As mudanças significativas no microsserviço são acompanhadas por atualizações da documentação do microsserviço?
- A documentação do microsserviço contém uma descrição do microsserviço?
- A documentação do microsserviço contém um diagrama da arquitetura?
- A documentação do microsserviço contém informações de contato e plantão?

- A documentação do microsserviço contém links para informações importantes?
- A documentação do microsserviço contém um guia de bordo e desenvolvimento?
- A documentação do microsserviço contém informações sobre o fluxo de solicitações, os endpoints e as dependências do microsserviço?
- A documentação do microsserviço contém um roteiro de plantão?
- A documentação do microsserviço contém uma seção FAQ?

Compreensão do microsserviço

- Todo desenvolvedor da equipe é capaz de responder a perguntas sobre a disponibilidade de produção do microsserviço?
- Existe um conjunto de princípios e padrões que todos os microsserviços devem seguir?
- Existe um processo RFC para cada novo microsserviço?
- Os microsserviços existentes são revisados e auditados com frequência?
- São realizadas revisões de arquitetura para cada equipe de microsserviço?
- Existe um processo de auditoria de disponibilidade de produção?
- São usados planos estratégicos de disponibilidade de produção para levar o microsserviço a um estado de pronto para produção?
- Padrões de disponibilidade de produção orientam os OKRs da organização?
- O processo de disponibilidade de produção é automatizado?

APÊNDICE A

Checklist de disponibilidade de produção

Esta será a checklist a ser executada em todos os microsserviços – manualmente ou de forma automatizada.

Um serviço pronto para produção é estável e confiável quando:

- ele tem um ciclo de desenvolvimento padronizado.
- seu código é cuidadosamente testado por meio de teste lint, de unidade, de integração e testes fim a fim.
- seu processo de teste, empacotamento, versão e liberação é completamente automatizado.
- ele tem um pipeline de deployment padronizado, que contém as fases de staging, pré-release e produção.
- seus clientes são conhecidos.
- suas dependências são conhecidas e existem backups, alternativas, fallbacks e cache em caso de falhas.
- ele tem roteamento e descoberta estáveis e confiáveis.

Um serviço pronto para produção é escalável e de alto desempenho quando:

- suas escalas de crescimento qualitativo e quantitativo são conhecidas.
- ele usa recursos de hardware de forma eficiente.
- seus gargalos e requisitos de recursos foram identificados.
- o planejamento de capacidade é automatizado e feito de forma

programada.

- suas dependências escalam com ele.
- ele escala com seus clientes.
- seus padrões de tráfego são compreendidos.
- o tráfego pode ser redirecionado em caso de falhas.
- ele é escrito em uma linguagem de programação que permite escalabilidade e alto desempenho.
- ele trata e processa tarefas com alto desempenho.
- ele trata e armazena dados de forma escalável e com alto desempenho.

Um serviço pronto para produção é tolerante a falhas e preparado para qualquer catástrofe quando:

- ele não tem um ponto único de falha.
- todos os cenários de falha de possíveis catástrofes foram identificados.
- ele é testado para resiliência por meio de testes de código, testes de carga e testes de caos.
- a detecção e o reparo de falhas foram automatizados.
- há procedimentos padronizados de incidente e interrupção dentro da equipe de desenvolvimento do microsserviço e da organização.

Um serviço pronto para produção é adequadamente monitorado quando:

- suas métricas principais são identificadas e monitoradas nos níveis de servidor, infraestrutura e microsserviço.
- ele tem um logging adequado, que reflete com precisão os estados passados do microsserviço.
- seus dashboards são fáceis de interpretar e contêm todas as métricas principais.
- seus alertas são acionáveis e definidos por limites sinalizadores.
- existe um turno dedicado de plantão responsável por monitorar e

responder a quaisquer incidentes e interrupções.

- há um procedimento claro, bem definido e padronizado de plantão para tratar incidentes e interrupções.

Um serviço pronto para produção é documentado e compreendido quando:

- ele tem uma documentação abrangente.
- sua documentação é atualizada regularmente.
- sua documentação contém uma descrição do microsserviço; um diagrama da arquitetura; informações de contato e plantão; links para informações importantes; um guia de bordo e desenvolvimento; informação sobre o(s) fluxo(s) de solicitação, os endpoints e as dependências do serviço; um roteiro de plantão; respostas a perguntas mais frequentes.
- ele é bem compreendido nos níveis de desenvolvedor, equipe e organizacional.
- ele segue um conjunto de padrões de disponibilidade de produção e cumpre os requisitos associados.
- sua arquitetura é revisada e auditada frequentemente.

APÊNDICE B

Avalie seu microsserviço

Para ajudar o leitor a avaliar a disponibilidade de produção de seu(s) microsserviço(s) e ecossistema de microsserviços, os capítulos 3 a 7 concluem com uma pequena lista de perguntas associadas ao padrão de disponibilidade de produção discutido. As perguntas são organizadas por tópicos e correspondem às seções dentro de cada capítulo. Todas as perguntas de cada capítulo foram agrupadas aqui para uma fácil referência.

Estabilidade e confiabilidade

Ciclo de desenvolvimento

- O microsserviço tem um repositório central, no qual todo o código está armazenado?
- Os desenvolvedores trabalham em um ambiente de desenvolvimento que reflete com precisão o estado da produção (por exemplo, que reflete com precisão o mundo real)?
- Existem testes adequados do tipo lint, de unidade, de integração e fim a fim em operação para o microsserviço?
- Existem procedimentos e políticas de revisão de código em vigor?
- O processo de teste, empacotamento, construção e liberação é automatizado?

Pipeline de deployment

- O ecossistema de microsserviços tem um pipeline de deployment padronizado?
- Existe uma fase de staging total ou parcial no pipeline de deployment?
- Qual o acesso que o ambiente de staging tem aos serviços de produção?

- Existe uma fase canário no pipeline de deployment?
- As implantações são executadas na fase de pré-release por um período de tempo longo o suficiente para detectar quaisquer falhas?
- A fase de pré-release trata com precisão uma amostra aleatória do tráfego de produção?
- As portas do microsserviço são as mesmas para o ambiente de pré-release e de produção?
- As implantações no ambiente de produção são feitas todas ao mesmo tempo ou gradualmente?
- Existe um procedimento em vigor para pular as fases de staging e pré-release em caso de emergência?

Dependências

- Quais são as dependências deste microsserviço?
- Quais são seus clientes?
- Como este microsserviço mitiga as falhas de dependência?
- Existem backups, alternativas, fallbacks ou cache defensivo para cada dependência?

Roteamento e descoberta

- As verificações de saúde do microsserviço são confiáveis?
- As verificações de saúde refletem com precisão a saúde do microsserviço?
- As verificações de saúde são realizadas em um canal separado dentro da camada de comunicação?
- Existem circuit breakers ativados para evitar que microsserviços não saudáveis façam solicitações?
- Existem circuit breakers ativados para evitar que o tráfego de produção seja enviado para servidores e microsserviços não saudáveis?

Descontinuação e desativação

- Existem procedimentos em funcionamento para desativar um microsserviço?

- Existem procedimentos em funcionamento para descontinuar endpoints de API de um microsserviço?

Escalabilidade e desempenho

Conhecer a escala de crescimento

- Qual a escala de crescimento qualitativo do microsserviço?
- Qual a escala de crescimento quantitativo do microsserviço?

Uso eficiente de recursos

- O microsserviço está sendo executado em hardware dedicado ou compartilhado?
- Estão sendo usadas tecnologias de abstração e alocação de recursos?

Percepção de recursos

- Quais os requisitos de recursos do microsserviço (CPU, RAM etc.)?
- Quanto tráfego uma instância do microsserviço consegue tratar?
- Quanta CPU uma instância do microsserviço exige?
- Quanta memória uma instância do microsserviço exige?
- Existem outros requisitos de recurso que são específicos deste microsserviço?
- Quais os gargalos de recursos deste microsserviço?
- Este microsserviço precisa ser escalado verticalmente, horizontalmente ou ambos?

Planejamento de capacidade

- O planejamento de capacidade é executado de forma programada?
- Qual o tempo de espera pelo novo hardware?
- Qual a frequência das solicitações de hardware?
- Alguns microsserviços têm prioridade quando as solicitações de hardware são feitas?
- O planejamento de capacidade é automatizado ou manual?

Dimensionamento das dependências

- Quais as dependências deste microsserviço?
- As dependências são escaláveis e de alto desempenho?
- As dependências escalarão com o crescimento esperado deste microsserviço?
- Os donos das dependências estão preparados para o crescimento esperado deste microsserviço?

Gerenciamento de tráfego

- Os padrões de tráfego do microsserviço são bem entendidos?
- As mudanças no serviço são programadas de acordo com os padrões de tráfego?
- As mudanças drásticas nos padrões de tráfego (especialmente picos de tráfego) são tratadas de forma cuidadosa e adequada?
- O tráfego pode ser automaticamente roteado para outros datacenters em caso de falha?

Tratamento e processamento de tarefas

- O microsserviço está escrito em uma linguagem de programação que permite que o serviço seja escalável e de alto desempenho?
- Existem limitações de escalabilidade ou desempenho na forma como o microsserviço trata as solicitações?
- Existem limitações de escalabilidade ou desempenho na forma como o microsserviço processa tarefas?
- Os desenvolvedores na equipe do microsserviço entendem como seu serviço processa tarefas, o quão eficientemente ele processa essas tarefas e como o serviço irá se comportar à medida que o número de tarefas e solicitações aumentar?

Armazenamento escalável de dados

- Este microsserviço trata dados de forma escalável e com alto desempenho?
- Que tipos de dados este microsserviço precisa armazenar?

- Qual o schema necessário para seus dados?
- Quantas transações são necessárias e/ou feitas por segundo?
- Este microsserviço precisa de um desempenho maior de escrita ou leitura?
- Ele faz uso intensivo de operações de leitura, escrita ou ambos?
- O database deste serviço é escalado horizontalmente ou verticalmente? Ele é replicado ou particionado?
- Este microsserviço usa um database dedicado ou compartilhado?
- Como o serviço trata e/ou armazena dados de teste?

Tolerância a falhas e preparação para catástrofes

Evitando pontos únicos de falha

- O microsserviço tem um ponto único de falha?
- Ele tem mais de um ponto de falha?
- Os pontos de falha podem ser eliminados ou eles precisam ser mitigados?

Cenários de catástrofes e falhas

- Todos os cenários de falha e possíveis catástrofes do microsserviço foram identificados?
- Quais as falhas comuns no ecossistema de microsserviços?
- Quais os cenários de falha da camada de hardware que podem afetar este microsserviço?
- Quais falhas nas camadas de comunicação e aplicação podem afetar este microsserviço?
- Quais falhas de dependência podem afetar este microsserviço?
- Quais falhas internas podem derrubar este microsserviço?

Teste de resiliência

- Este microsserviço tem testes adequados do tipo lint, de unidade, de integração e fim a fim?

- Este microsserviço é submetido a testes de carga regulares e programados?
- Todos os possíveis cenários de falha são implementados e testados usando o teste de caos?

Detecção e reparo de falhas

- Existem processos padronizados na organização de engenharia de software para tratar incidentes e interrupções?
- Como as falhas e interrupções deste microsserviço impactam o negócio?
- Existem níveis claramente definidos de falha?
- Existem estratégias de mitigação claramente definidas?
- A equipe segue os cinco estágios de resposta a incidentes quando ocorrem incidentes e interrupções?

Monitoramento

Métricas principais

- Quais são as métricas principais deste microsserviço?
- Quais são as métricas de servidor e infraestrutura?
- Quais são as métricas no âmbito de microsserviço?
- Todas as métricas principais do microsserviço são monitoradas?

Logging

- Quais informações este microsserviço precisa gravar em log?
- Este microsserviço grava em log todas as solicitações importantes?
- O logging reflete o estado do microsserviço em qualquer momento?
- Este logging é escalável e eficaz em termos de custo?

Dashboards

- Este microsserviço tem um dashboard?
- O dashboard é fácil de interpretar? Todas as métricas principais são

exibidas no dashboard?

- Posso determinar se este microsserviço está funcionando corretamente ou não apenas olhando o dashboard?

Alertas

- Existe um alerta para todas as métricas principais?
- Todos os alertas são definidos por adequados limites sinalizadores?
- Os limites de alerta são configurados adequadamente de modo a disparar um alerta antes que ocorra uma interrupção?
- Todos os alertas são acionáveis?
- Existem instruções passo a passo para triagem, mitigação e resolução para cada alerta no roteiro de plantão?

Turnos de plantão

- Existe um turno de plantão dedicado e responsável por monitorar este microsserviço?
- Cada turno de plantão contém pelo menos dois desenvolvedores?
- Existem procedimentos padronizados de plantão em toda a organização de engenharia de software?

Documentação e compreensão

Documentação do microsserviço

- A documentação de todos os microsserviços está armazenada em um local centralizado, compartilhado e de fácil acesso?
- É fácil pesquisar a documentação?
- As mudanças significativas no microsserviço são acompanhadas por atualizações da documentação do microsserviço?
- A documentação do microsserviço contém uma descrição do microsserviço?
- A documentação do microsserviço contém um diagrama da arquitetura?
- A documentação do microsserviço contém informações de contato e plantão?

- A documentação do microsserviço contém links para informações importantes?
- A documentação do microsserviço contém um guia de bordo e desenvolvimento?
- A documentação do microsserviço contém informações sobre o fluxo de solicitações, os endpoints e as dependências do microsserviço?
- A documentação do microsserviço contém um roteiro de plantão?
- A documentação do microsserviço contém uma seção FAQ?

Compreensão do microsserviço

- Todo desenvolvedor da equipe é capaz de responder perguntas sobre a disponibilidade de produção do microsserviço?
- Existe um conjunto de princípios e padrões que todos os microsserviços devem seguir?
- Existe um processo RFC para cada novo microsserviço?
- Os microsserviços existentes são revisados e auditados com frequência?
- São feitas revisões de arquitetura para cada equipe de microsserviço?
- Há um processo de auditoria de disponibilidade de produção?
- São usados planos estratégicos de disponibilidade de produção para levar o microsserviço a um estado de pronto para produção?
- Padrões de disponibilidade de produção orientam os OKRs da organização?
- O processo de disponibilidade de produção é automatizado?

Glossário

alerta acionável

Um alerta que, quando disparado, contém um processo passo a passo que o turno de plantão pode seguir para triar, mitigar e resolver o alerta.

alocação de recursos

Dividir os recursos de hardware disponíveis pelos ecossistemas de microsserviços.

ambiente de desenvolvimento

Um sistema que contém ferramentas, variáveis de ambiente e processos usados pelos desenvolvedores para escrever código para microsserviços.

arquitetura de três camadas

Uma arquitetura básica para aplicações de software, que consiste em um elemento frontend (lado do cliente), um elemento de backend e algum tipo de armazenamento de dados.

auditoria de disponibilidade de produção

O processo de avaliar a disponibilidade de produção de um microsserviço usando uma checklist de disponibilidade de produção.

automatização da disponibilidade de produção

Um método para garantir que microsserviços atendam aos padrões de disponibilidade de produção ao verificar de forma automática e programada se cada microsserviço cumpre os requisitos associados a cada padrão de disponibilidade de produção.

balanceamento de carga

Um dispositivo ou serviço que distribui o tráfego por vários servidores ou microsserviços.

cache defensivo

A prática de salvar em cache os dados das dependências downstream de um microsserviço para proteger esse microsserviço contra problemas de

estabilidade e confiabilidade se a dependência downstream estiver indisponível.

camada de comunicação

A segunda camada do ecossistema de microsserviços; contém a rede, o DNS, o frameworks RPC, os endpoints, troca de mensagens, descoberta de serviços, registro de serviços e balanceamento de carga.

camada de hardware

A primeira camada do ecossistema de microsserviços; contém servidores físicos, sistemas operacionais, tecnologias de isolamento e abstração de recursos, gerenciamento de configuração, monitoramento em nível de servidor e logging em nível de servidor.

camada de microsserviço

A quarta camada do ecossistema de microsserviços; contém os microsserviços e todas as configurações específicas do microsserviço.

camada de plataforma de aplicação

A terceira camada de um ecossistema de microsserviços, que contém ferramentas internas do tipo autosserviço, o ambiente de desenvolvimento, ferramentas de teste, empacotamento, versão e liberação, o pipeline de deployment, logging em nível de microsserviço e monitoramento em nível de microsserviço.

candidata à produção

Uma versão que passou com sucesso por todos os testes lint, de unidade, de integração e testes fim a fim no ciclo de desenvolvimento e está pronta para ser introduzida no pipeline de deployment.

chamada de procedimento remoto (RPC)

Uma chamada feita pela rede para um servidor remoto que foi concebida para se parecer e se comportar exatamente como uma chamada para um procedimento local; usada amplamente na arquitetura de microsserviços e em todos os sistemas distribuídos de larga escala.

checklist de disponibilidade de produção

Uma lista de padrões de disponibilidade de produção, junto a requisitos específicos que podem ser implementados para se alcançar o padrão de

disponibilidade de produção.

ciclo de desenvolvimento

Um nome para o processo geral associado ao desenvolvimento de uma aplicação, um microsserviço ou sistema.

dashboard

Uma representação visual e gráfica em um site interno que contém gráficos e tabelas da saúde e o comportamento de métricas principais de uma aplicação, um microsserviço ou sistema.

dependência

Um nome de qualquer outro microsserviço para o qual um microsserviço envia solicitações; refere-se a bibliotecas das quais um microsserviço depende; também usado para se referir a serviços externos (de terceiros) dos quais um microsserviço depende.

desativação

O processo de retirar um microsserviço e/ou seus endpoints de API de modo que eles não estejam mais disponíveis para uso por serviços upstream (cliente).

descoberta de serviços (service discovery)

Um sistema que descobre onde todas as instâncias de um microsserviço estão hospedadas, garantindo que o tráfego seja roteado para os servidores adequados que estão hospedando a aplicação.

descontinuação

Quando um microsserviço e/ou seus endpoints não são mais mantidos por uma equipe de desenvolvimento e não mais recomendados para uso por serviços upstream (cliente).

diagrama de arquitetura

Uma representação visual de alto nível da arquitetura de um microsserviço.

dividir o monólito

O nome dado ao processo de quebrar uma grande aplicação monolítica em um conjunto de microsserviços.

ecossistema de microsserviços

Um termo para o sistema geral que contém os microsserviços e a infraestrutura, que pode ser dividido em quatro camadas contendo as camadas de microsserviços, plataforma de aplicação, comunicação e hardware.

endpoint

Neste livro, este termo se refere a endpoints estáticos de API (HTTP, Thrift etc.) de microsserviços para os quais as solicitações são roteadas.

engenharia de confiabilidade (SRE)

Engenheiros operacionais responsáveis em grandes empresas pela confiabilidade das aplicações, dos microsserviços ou sistemas dentro das organizações de engenharia de software.

engenheiros operacionais

Engenheiros cujas principais responsabilidades são as tarefas operacionais associadas à execução de uma aplicação de software, incluindo administradores de sistema, TechOps, DevOps e engenheiros de confiabilidade.

envio de alertas

A prática de notificar um desenvolvedor (ou alguns desenvolvedores) de plantão quando uma das métricas principais do serviço atingiu um limite de alerta crítico ou de advertência.

escala de crescimento

Um nome dado à medida de crescimento de uma aplicação, um microsserviço ou sistema; toda aplicação, todo microsserviço e sistema têm dois tipos de escalas de crescimento, uma escala de crescimento quantitativo e uma escala de crescimento qualitativo.

escala de crescimento qualitativo

Uma medida qualitativa de alto nível de como uma aplicação, um microsserviço ou sistema escala que está vinculada a métricas de negócio de alto nível; um tipo de escala de crescimento.

escala de crescimento quantitativo

Uma medida quantitativa de alto nível de como uma aplicação, um

microsserviço ou sistema escala; obtida por meio da versão da escala de crescimento qualitativo em uma quantidade mensurável; um tipo de escala de crescimento; geralmente expressa em termos de solicitações por segundo, consultas por segundo ou transações por segundo que a aplicação, o microsserviço ou sistema consegue processar.

escalamento horizontal

Quando uma aplicação ou um sistema é escalado adicionando-se mais servidores (ou outros recursos de hardware).

escalamento vertical

Quando uma aplicação ou um sistema é escalado aumentando-se os recursos (CPU, RAM) de cada servidor no qual a aplicação ou o sistema está sendo executado.

falhas externas

Falhas dentro das três camadas inferiores da pilha do ecossistema de microsserviços.

falhas internas

Falhas dentro de um microsserviço.

ferramentas internas do tipo autosserviço

Ferramentas padronizadas na camada de plataforma de aplicação de um ecossistema de microsserviços que são construídas para ajudar os desenvolvedores a trabalhar com as camadas inferiores do ecossistema de microsserviços para desenvolver, implantar e executar seus microsserviços.

fluxo de solicitação

Um nome para o padrão de passos que são tomados quando uma solicitação é feita de um microsserviço para outro.

hardware compartilhado

Servidores ou databases que são usados para abrigar ou armazenar dados para mais de uma aplicação, um microsserviço ou sistema ao mesmo tempo.

hardware dedicado

Servidores ou databases que abrigam ou armazenam dados apenas de uma aplicação, um microsserviço ou sistema.

gargalos de recursos

Limitações de escalabilidade causadas pelo modo como uma aplicação, um microsserviço ou sistema usa seus recursos.

implantação

O processo pelo qual uma nova versão é enviada para os servidores e o serviço é iniciado.

infraestrutura

Um termo usado neste livro para se referir à combinação da camada da plataforma de aplicação e da camada de comunicação ou à combinação das três camadas inferiores do ecossistema de microsserviços (camada de hardware, camada de comunicação e camada de plataforma de aplicação).

integração contínua

Um processo que automaticamente integra, testa, empacota e constrói novas alterações no código de forma programada e contínua.

interface de programação da aplicação (API)

Uma bem definida interface do lado cliente em cada microsserviço que permite que outros serviços interajam com ele programaticamente enviando solicitações para endpoints estáticos.

interrupção

Um período de tempo durante o qual uma aplicação, um microsserviço ou sistema está inacessível (downtime).

Lei de Conway

Uma “lei” informal de arquitetura de software cujo nome é uma homenagem a Melvin Conway e segundo a qual a estrutura da arquitetura dos produtos de uma empresa é determinada pelos padrões de comunicação da organização; veja também Lei Reversa de Conway.

Lei Reversa de Conway

O reverso da Lei de Conway, que diz que a estrutura organizacional de

uma empresa é determinada pela arquitetura de seu(s) produto(s).

limite de alerta

Quantidades estáticas ou dinâmicas que são configuradas para cada métrica principal, indicando que essa métrica está no nível normal, de advertência ou crítico; atingir o limite deve disparar um alerta acionável.

logging

A prática de gravar os eventos de uma aplicação, um microsserviço ou sistema.

métrica do microsserviço

As métricas principais exclusivas de cada microsserviço na camada de microsserviço do ecossistema de microsserviços.

métricas de servidor e infraestrutura

Métricas principais das três camadas inferiores (camada de hardware, camada de comunicação e camada de plataforma de aplicação) do ecossistema de microsserviços.

métricas principais

Propriedades de uma aplicação, um microsserviço ou sistema que são necessárias e suficientes para descrever a saúde, o status e o comportamento da aplicação, do microsserviço ou do sistema.

microsserviço

Uma aplicação de software pequena, substituível, modular, desenvolvida independentemente e implantada independentemente que é responsável por executar uma função dentro de um sistema maior.

monitoramento

A prática de observar e rastrear o status, a saúde e o comportamento de uma aplicação ou das métricas principais de um microsserviço num determinado período de tempo.

monólito

Sistemas de software grandes e complexos que são mantidos, executados e implantados como uma única aplicação contendo todo o código e os recursos relativos à aplicação.

nota de disponibilidade de produção

Uma nota atribuída aos microsserviços que é calculada com base em quão bem o microsserviço em questão cumpre os requisitos associados a cada padrão de disponibilidade de produção.

paridade de servidor

Quando dois ambientes, dois sistemas ou duas fases isoladas do pipeline de deployment (por exemplo, staging e produção) têm o mesmo número de servidores em cada ambiente, sistema, datacenter ou fase de implantação.

particionamento

O processo e a prática arquitetônica de dividir cada tarefa em partes menores que podem ser processadas em paralelo; propriedade essencial da escalabilidade.

pipeline de deployment

O processo de implantar novas versões em três estágios (staging, pré-release e então produção).

planejamento de capacidade

A prática organizacional de uma alocação de recursos planejada e programada.

plano estratégico de disponibilidade de produção

Um documento usado como parte do processo de disponibilidade de produção que detalha os passos que precisam ser dados para levar um microsserviço a um estado de disponibilidade de produção.

ponto único de falha (SPOF)

Um trecho de uma aplicação, um microsserviço ou sistema que, se falhar, derruba a aplicação, o microsserviço ou o sistema.

pré-release (fase canary)

O segundo estágio do pipeline de deployment, que contém um pequeno percentual dos servidores que recebem tráfego de produção (2% a 5% do tráfego de produção); usado para testar novas versões que passaram pela fase de staging antes de ser implantadas em todos os servidores de produção.

produção

O estágio final do pipeline de deployment, onde todo tráfego do mundo real é hospedado; também usado para se referir ao tráfego do mundo real e o ambiente que hospeda esse tráfego.

provedores de serviços em nuvem

Empresas como a Amazon Web Serviços (AWS), a Google Cloud Platform (GCP) e a Microsoft Azure, que permitem que recursos de hardware sejam alugados e facilmente acessíveis por meio de redes seguras.

recursos

Uma abstração de várias propriedades de desempenho de hardware (servidores), como CPU, memória, rede etc.

recursos de hardware

Veja recursos.

registro de serviços (service registry)

Um database que registra todas as portas e o IPs de todos os microsserviços e sistemas dentro de um ecossistema de microsserviços.

repositório

Um arquivo centralizado, em que todo o código-fonte de uma aplicação ou serviço está armazenado.

requisitos de recursos

Os recursos exigidos por uma aplicação, um microsserviço ou sistema.

revisão de arquitetura

Uma prática e um processo organizacional para avaliar, compreender e melhorar a arquitetura de um microsserviço.

roteiro de plantão

Uma seção da documentação do microsserviço que contém procedimentos gerais de resposta a incidentes e interrupções, instruções passo a passo sobre como triar, mitigar e resolver cada alerta, e dicas gerais sobre como depurar e solucionar problemas do microsserviço; usado pelos desenvolvedores ou engenheiros de operação que estão de

plantão para o serviço.

servidor físico

O termo usado para se referir a servidores que são de propriedade, operados e mantidos pela própria organização, ao contrário de hardware alugado dos chamados provedores de serviço em nuvem.

simultaneidade

Aplicações e microsserviços que têm simultaneidade dividem cada tarefa em partes pequenas, em vez de um único processo que faz todo o serviço; propriedade essencial exigida para escalabilidade.

staging

A primeira fase de um pipeline de deployment que não atende tráfego de produção e é usada para testar novas versões; geralmente é uma cópia-espelho da produção; pode ser implementada como staging total ou staging parcial.

staging parcial

Quando a fase de staging do pipeline de deployment não é uma cópia-espelho completa da produção, mas onde microsserviços no ambiente de staging conversam com versões de produção dos clientes, dependências e databases.

staging total

Quando a fase de staging do pipeline de deployment executa uma cópia-espelho completa da produção.

teste de código

Testes que verificam sintaxe, estilo, componentes individuais de um microsserviço, como os componentes trabalham juntos e como o microsserviço se comporta dentro de suas complexas cadeias de dependências; consiste em testes lint, testes de unidade, testes de integração e testes fim a fim.

testes de integração

Testam como os componentes do microsserviço (que são testados individualmente usando testes de unidade) funcionam juntos.

testes de unidade

Pequenos testes independentes que são executados sobre pequenos trechos (ou unidades) do código de um microsserviço; teste de parte do código.

testes fim a fim

Testes que verificam se as alterações feitas em uma aplicação, um serviço ou sistema funcionam como esperado, avaliando endpoints, clientes, dependências e quaisquer databases.

testes lint

Testes que verificam os erros de sintaxe e estilo; parte do pacote de teste de código.

troca de mensagens do tipo publicar – assinar

Um paradigma de troca de mensagens assíncrono no qual clientes assinam um tópico e recebem uma mensagem sempre que um editor publica uma mensagem nesse tópico.

troca de mensagens do tipo solicitação – resposta

Um paradigma de troca de mensagens no qual um cliente envia uma solicitação para um microsserviço (ou broker de mensagem) que responderá com as informações solicitadas.

turno de plantão

Um grupo de desenvolvedores ou engenheiros de operação que são responsáveis por responder, mitigar e resolver os alertas, incidentes e falhas de uma aplicação, microsserviço ou sistema.

velocidade do desenvolvedor

A velocidade na qual as equipes de desenvolvimento são capazes de iterar, lançar novos recursos e implantar.

Sobre a autora

Susan Fowler é uma engenheira de confiabilidade na Uber Technologies, onde ela divide seu tempo entre implantar uma iniciativa de disponibilidade de produção em todos os microsserviços do Uber e integrar-se a equipes de aplicações críticas para o negócio, buscando tornar seus serviços prontos para produção. Ela trabalhou em plataformas e infraestrutura de aplicação em várias pequenas startups antes de entrar no Uber, e antes disso ela estudou física de partículas na University of Pennsylvania, onde pesquisou supersimetria e projetou hardware para os detectores ATLAS e CMS.

Colofão

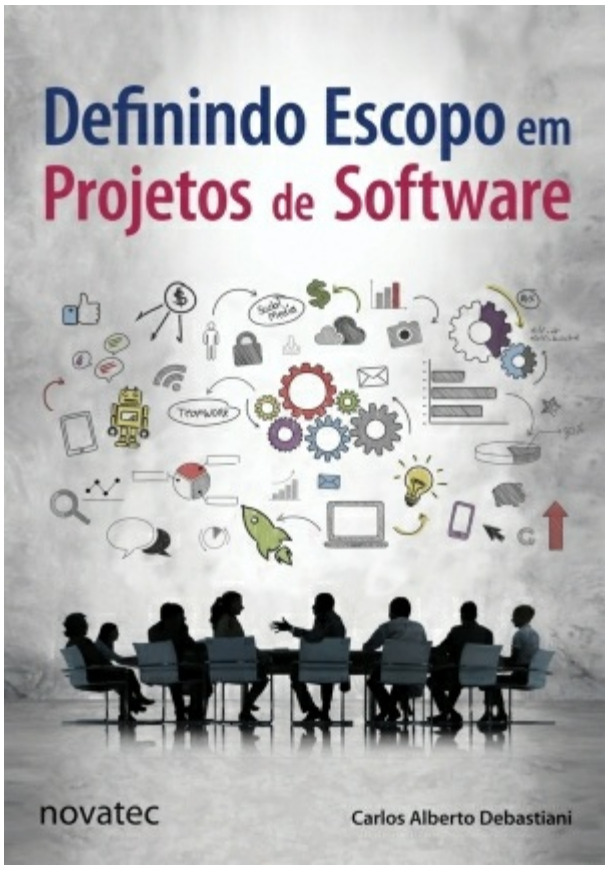
Os animais na capa do livro *Microsserviços Prontos para Produção* são abelhas cortadeiras (do gênero Megachile). Existem mais de 1.500 espécies desse inseto espalhadas pelo mundo. Uma espécie da Indonésia, chamada Megachile pluto, é considerada a maior abelha do mundo, podendo atingir de 2,3 a 3,8 centímetros de comprimento.

Abelhas cortadeiras têm este nome em razão da atividade comum da fêmea de cortar perfeitos semicírculos das extremidades das folhas. Ela então carrega esses pedaços de folha em forma de disco para seu ninho, que pode ser construído em vários lugares, como cavidades prontas, buracos no solo ou madeira apodrecida que a abelha pode perfurar. Os ninhos têm entre 10 e 20 centímetros de comprimento, são cilíndricos e revestidos com pedaços de folhas em um padrão de sobreposição. Estes insetos não vivem em colônias, embora seja possível que abelhas individuais construam ninhos próximos entre si.

As fêmeas organizam seus ninhos em células separadas (construindo de dentro para fora) e põem um ovo em cada célula, juntamente com uma bola regurgitada de pólen e néctar para alimentar a larva. A teoria é que as folhas evitam que o alimento da larva seque até que possa ser consumido. Abelhas adultas também se alimentam de néctar e pólen e são polinizadoras muito eficientes graças ao seu vigoroso movimento, semelhante ao ato de nadar, enquanto estão dentro das flores (o que espalha bastante pólen e recobre os longos pelos no abdômen do inseto). As fêmeas geralmente precisam de dez a quinze viagens para formar uma célula individual no ninho, o que aumenta ainda mais sua eficácia de polinização cruzada. Assim, essas abelhas são bem-vindas em jardins e fazendas; é possível colocar caixas de nidificação artificiais ou tubos para atraí-las.

Muitos dos animais nas capas de livros da O’Reilly são espécies em extinção; todos são importantes para o mundo. Para saber mais sobre como você pode ajudar, acesse *animals.oreilly.com*.

A imagem da capa foi tirada do livro Royal Natural History (Real História Natural), de Richard Lydekker.


Definindo Escopo em
Projetos de Software
novatec
Carlos Alberto Debastiani

Definindo Escopo em Projetos de Software

Debastiani, Carlos Alberto

9788575224960

144 páginas

Definindo Escopo em Projetos de Software é uma obra que pretende tratar, de forma clara e direta, a definição de escopo como o fator mais influente no sucesso dos projetos de desenvolvimento de sistemas, uma vez que exerce forte impacto sobre seus custos. Abrange diversas áreas do conhecimento ligadas ao tema, abordando desde questões teóricas como a normatização e a definição das características de engenharia de software, até questões práticas como métodos para coleta de requisitos e ferramentas para desenho e projeto de soluções sistêmicas. Utilizando uma linguagem acessível, diversas ilustrações e citações de casos vividos em sua própria experiência profissional, o autor explora, de forma abrangente, os detalhes que envolvem a definição de escopo, desde a identificação das melhores fontes de informação e dos envolvidos na tomada de decisão, até as técnicas e ferramentas usadas no levantamento de requisitos, no projeto da solução e nos testes de aplicação.

O'REILLY

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações nativas de nuvem

Bilgin Ibryam
Roland Huß

novatec

Padrões para Kubernetes

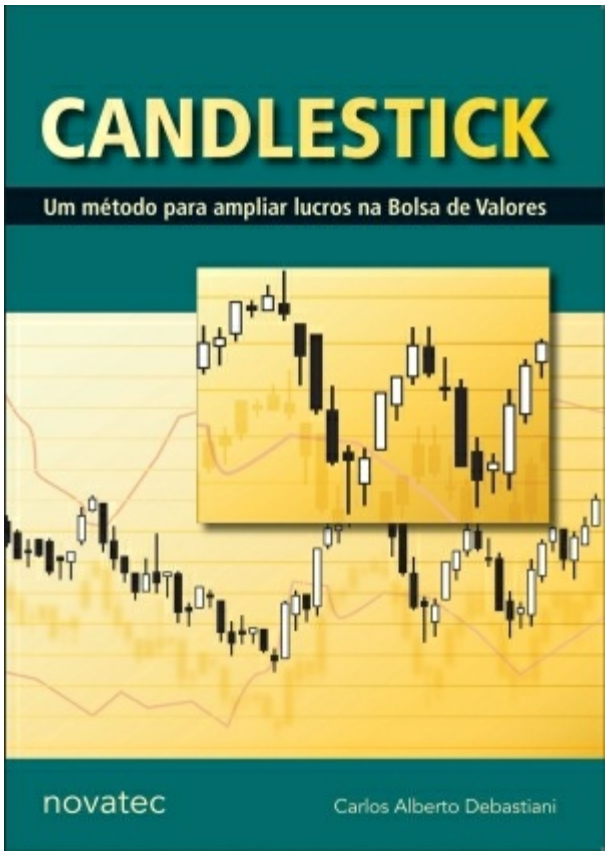
Ibryam, Bilgin

9788575228159

272 páginas

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões: • Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações

nativas de nuvem com base em contêineres. • Padrões comportamentais, que exploram conceitos mais específicos para administrar contêineres e interações com a plataforma. • Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. • Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. • Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).


CANDLESTICK
Um método para ampliar lucros na Bolsa de Valores
novatec
Carlos Alberto Debastiani

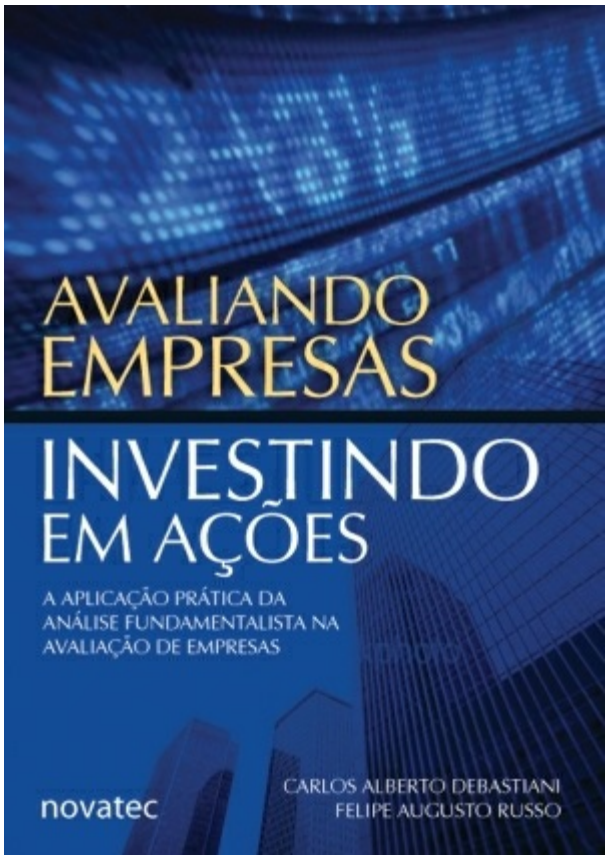
Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.


AVALIANDO
EMPRESAS
INVESTINDO
EM AÇÕES
A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS
novatec
CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.


Marcos Abe
MANUAL DE
ANÁLISE
TÉCNICA
ESSÊNCIA E ESTRATÉGIAS AVANÇADAS
TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE
novatec

Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá: - os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado; - identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas; um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos; - estruturar e proteger operações por meio do gerenciamento de capital. Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para

operar lucrativamente.