LUCAS NUNES [1]
PEDRO SIDNEI ZANCHETT [2]

Migração de Um Sistema de CRM Monolítico para uma Arquitetura de Microsserviços

Migrating a monolith CRM system to a microservices architecture

ARTIGO 6

79-96

[1] Bacharel em Sistemas de Informação - Centro Universitário Leonardo da Vinci – UNIASSELVI/Blumenau/SC. E-mail: lucasnunes.ln365@gmail.com
[2] Prof. Mestre em Gestão do Conhecimento. Centro Universitário Leonardo da Vinci – UNIASSELVI/Blumenau/SC. E-mail: pedro.zanchett@uniasselvi.com.br

Resumo: Este trabalho aborda a migração de um sistema de CRM monolítico para uma arquitetura de microsserviços, destacando os desafios e benefícios envolvidos. A arquitetura monolítica, embora simples para desenvolvimento inicial, apresenta limitações significativas em termos de escalabilidade e manutenção à medida que os sistemas crescem. Em contrapartida, os microsserviços oferecem maior flexibilidade, escalabilidade e resiliência, permitindo que os serviços sejam desenvolvidos e implantados de forma independente. A conversão, entretanto, exige planejamento cuidadoso. Neste sentido, o Strangler Pattern se mostrou uma abordagem eficaz para realizar essa transição de forma incremental, minimizando riscos. Esta pesquisa objetiva trazer exemplos práticos que demonstrem a implementação de microsserviços que podem aumentar a agilidade e a capacidade de resposta das empresas, mas é essencial uma análise detalhada das necessidades para garantir o sucesso.

Palavras-chave: Arquitetura de Microsserviços. Arquitetura Monolítica. Migração. CRM. Conversão.

Abstract: This paper addresses the migration of a monolithic CRM system to microservices architecture, highlighting the challenges and benefits involved. The monolithic architecture, although simple for initial development, presents significant limitations in terms of scalability and maintenance as systems grow. In contrast, microservices offer greater flexibility, scalability, and resilience, allowing services to be developed and deployed independently. The conversion, however, requires careful planning, and the Strangler Pattern has proven to be an effective approach to accomplish this transition incrementally, minimizing risks. This research then aims to provide practical examples that demonstrate the implementation of microservices that can increase the agility and responsiveness of companies, but a detailed analysis of the needs is essential to ensure success.

Keywords: Microservices Architecture. Monolithic Architecture. Migration. CRM. Conversion.

INTRODUÇÃO

Nos últimos anos, a transformação digital tem impulsionado a evolução dos sistemas empresariais, aumentando a necessidade de escalabilidade, flexibilidade e eficiência. Nesse contexto, os sistemas de Customer Relationship Management (CRM) desempenham um papel essencial ao gerenciar o relacionamento das empresas com seus clientes, organizando informações estratégicas para os negócios. Para Giacom (2021), a adoção de soluções tecnológicas voltadas para o relacionamento com o cliente contribui significativamente para a reputação e o sucesso das organizações. À medida que as empresas crescem, os sistemas de CRM precisam se adaptar para suportar uma maior quantidade de dados, processos mais complexos e integrações com diferentes plataformas. A arquitetura tradicional monolítica, amplamente utilizada, tem se mostrado limitada e ultrapassada para lidar com essas novas demandas, apresentando desafios relacionados à escalabilidade, manutenção e implementação de novas funcionalidades.

A migração de sistemas CRM de uma arquitetura monolítica para uma arquitetura de microsserviços surge como uma resposta a esses desafios. Microsserviços permitem que um sistema seja dividido em módulos independentes, cada um com suas responsabilidades específicas, possibilitando um desenvolvimento mais ágil, manutenção mais simples e maior escalabilidade horizontal. Segundo Mathlimma (2021), empresas como Netflix, Amazon e Uber são exemplos de sucesso na adoção dessa abordagem, permitindo-lhes crescer de forma rápida e eficiente.

De acordo com Blinowski *et al*. (2022), a arquitetura de microsserviços permite uma escalabilidade horizontal eficiente, pela qual os serviços são implementados de forma autônoma e desacoplada, facilitando a organização de equipes menores e mais ágeis para o desenvolvimento. Essa abordagem modular ajuda a estruturar melhor o domínio de negócios em contextos menores e mais focados, permitindo maior flexibilidade e agilidade tanto no desenvolvimento quanto na escalabilidade dos serviços.

A modularidade proporcionada pela arquitetura de microsserviços permite que novas funcionalidades sejam implementadas e testadas sem impacto nas demais partes do sistema. Além disso, a separação de responsabilidades facilita o isolamento de falhas e a recuperação de incidentes, tornando o sistema mais resiliente a problemas técnicos.

Para Abgaz *et al*. (2023), a arquitetura de microsserviços tem crescido significativamente nos últimos anos devido à sua capacidade de melhorar a escalabilidade e agilidade no desenvolvimento de software. A transição de sistemas monolíticos para microsserviços oferece vantagens como maior independência entre componentes e menor acoplamento, o que resulta em uma maior flexibilidade na manutenção e implantação dos sistemas. Embora desafiador, esse processo é impulsionado pela demanda por sistemas que lidam com cargas variáveis e requerem mais velocidade nas atualizações.

A principal problemática abordada nesta pesquisa é a persistência de grandes e consolidados sistemas de CRM construídos sob uma arquitetura monolítica, o que limita sua capacidade de adaptação a novas demandas do mercado e ao crescimento contínuo.

Com o aumento da base de clientes e a necessidade de respostas em tempo real, sistemas monolíticos podem se tornar lentos e difíceis de escalar, além de tornarem a implementação de novas funcionalidades mais complexa e propensa a erros. Conforme destacado por Stradolini (2022), a migração de sistemas monolíticos para uma arquitetura de microsserviços visa resolver problemas como a dificuldade de escalabilidade e manutenção, proporcionando maior flexibilidade e resiliência, e permitindo que as empresas atendam melhor seus clientes e se adaptem às mudanças de mercado de maneira mais ágil.

OBJETIVO

Definir a migração de um sistema de CRM monolítico para uma arquitetura de microsserviços, visando aumentar a escalabilidade, flexibilidade e independência dos componentes, além de facilitar a manutenção e integração de novas tecnologias. Essa migração permitirá a divisão do sistema em serviços menores e mais coesos, melhorando o desempenho e a capacidade de resposta, com foco na redução do acoplamento entre as partes do sistema e no aumento da velocidade de desenvolvimento e entrega de novas funcionalidades.

REFERENCIAL TEÓRICO

Esta seção busca destacar os principais conceitos teóricos sobre migração de um sistema de CRM monolítico para uma arquitetura de microsserviços. De acordo com a empresa Salesforce (2024), um CRM (*Customer Relationship Management*) é um sistema que auxilia as empresas a gerenciar suas interações e relacionamentos com clientes, fornecedores e outros contatos importantes.

O objetivo principal de um CRM é melhorar o atendimento ao cliente, otimizar processos e aumentar as vendas por meio de uma abordagem centralizada e organizada. Ele permite que as empresas acompanhem o histórico de interações com cada cliente, integrem dados de diferentes departamentos (vendas, marketing e suporte) e automatize tarefas administrativas. Além disso, os CRMs fornecem insights valiosos por meio da análise de dados, ajudando a melhorar a tomada de decisões.

Algumas funcionalidades podem ser destacadas como componentes essenciais de um CRM. De acordo com a ClickUp (2024), os principais elementos de um CRM incluem:

1. **Gestão de Contatos**: registro detalhado de informações sobre clientes, como histórico de compras, preferências e dados de contato.
2. **Gestão de Vendas**: acompanhamento do ciclo de vendas, processos e oportunidades de negócio.
3. **Automação de Marketing**: ferramentas para campanhas de marketing direcionadas, mensuração de resultados e segmentação de clientes.
4. **Suporte ao Cliente**: gerenciamento de tickets, rastreamento de problemas e centralização de interações de suporte.
5. **Relatórios e Análises**: dashboards e relatórios personalizados que ajudam na visualização e interpretação dos dados dos clientes.

Esses sistemas podem ser customizados de acordo com as necessidades da empresa, sendo utilizados tanto por pequenas quanto grandes organizações. Segundo Ferrer-Estévez e Chalmeta (2022), um CRM é uma mudança na estratégia empresarial que muda de uma estratégia focada no produto para uma focada no cliente. O CRM permite que as empresas tenham uma visão única e integrada dos clientes, usando ferramentas analíticas que gerenciam os relacionamentos com os clientes de uma única maneira, independentemente do canal de comunicação.

De acordo com a Amazon (2024), uma arquitetura monolítica é um modelo de design de software no qual todas as funcionalidades de uma aplicação estão agrupadas em um único código-fonte. Essa abordagem centralizada significa que todos os componentes do sistema, como a interface de usuário, lógica de negócios e banco de dados, são executados como uma única unidade em um único processo. Em uma arquitetura monolítica, qualquer modificação no sistema requer o redesenvolvimento e reimplantação de toda a aplicação.

Em uma arquitetura monolítica, o sistema é construído como um bloco único e indivisível. O ciclo de vida de desenvolvimento segue o padrão tradicional, com as fases de compilação, teste e implantação acontecendo de uma só vez. De acordo com a Amazon (2024), a estrutura monolítica geralmente inclui:

1. **Interface de usuário**: responsável pela interação com o usuário final.
2. **Lógica de negócios**: implementa as regras de negócios da aplicação.
3. **Acesso a dados**: cuida da comunicação com o banco de dados ou serviços de armazenamento.

Podemos observar alguns pontos positivos de uma arquitetura monolítica, como todos esses componentes são integrados diretamente, e o software é executado como um único processo, isso facilita alguns aspectos do desenvolvimento de uma aplicação. De acordo com a TechTarget (2024), podemos citar os principais pontos positivos de uma arquitetura monolítica:

1. **Facilidade de Desenvolvimento Inicial**: no começo, é mais simples desenvolver e testar uma aplicação monolítica, já que todas as partes do sistema estão contidas em um único local.
2. **Desempenho Interno**: como todos os componentes residem no mesmo processo, a comunicação interna é rápida, não sendo necessário lidar com problemas de latência causados por redes, como acontece em arquiteturas distribuídas.
3. **Facilidade de depuração e teste**: em uma arquitetura monolítica, todo o sistema é executado como um único aplicativo, o que facilita o processo de depuração, já que desenvolvedores podem usar ferramentas centralizadas para identificar e corrigir problemas.
4. **Menor complexidade de implantação**: com uma única unidade de implantação, o processo de implantação é direto, sem a necessidade de gerenciar a comunicação entre diferentes serviços ou componentes distribuídos.

Segundo Shrestha (2024), a arquitetura monolítica possui diversos benefícios que justificam sua popularidade no desenvolvimento de software. Um dos principais pontos positivos é sua simplicidade, pois a base de código unificada facilita os processos de implantação e desenvolvimento. Com todos os componentes centralizados, o ciclo de desenvolvimento se torna mais eficiente, além de simplificar o teste e a depuração. Esse formato de base de código tende a acelerar o desenvolvimento inicial, especialmente em projetos menores (TechTarget, 2024).

A arquitetura monolítica, infelizmente, apresenta mais desafios do que benefícios. De acordo com Shrestha (2024), essa abordagem enfrenta obstáculos significativos, como dificuldades de escalabilidade, alta dependência entre equipes e baixa resiliência. Como toda a aplicação precisa ser escalada como um único bloco, o consumo de recursos aumenta consideravelmente. Além disso, essa arquitetura limita a flexibilidade na escolha de tecnologias e restringe as opções de escalabilidade, tornando a adaptação e evolução do sistema mais complexas.

De acordo com a TechTarget (2024), podemos citar alguns pontos negativos relevantes que devem ser levados em conta ao pensar nessa arquitetura:

1. **Desvantagens com o Crescimento**: conforme a aplicação cresce, a arquitetura monolítica pode se tornar difícil de manter. Mudanças em um pequeno componente podem impactar todo o sistema, exigindo a recompilação e reimplantação de toda a aplicação.
2. **Escalabilidade**: em uma arquitetura monolítica, a escalabilidade é um desafio, pois geralmente envolve replicar toda a aplicação, mesmo que apenas uma parte dela precise de mais recursos.
3. **Dependência de Equipes**: como tudo está interligado, equipes de desenvolvimento precisam colaborar constantemente, e qualquer mudança precisa passar por uma série de testes e validações para evitar impactos em outras áreas do sistema.
4. **Resiliência**: se um componente falhar, ele pode derrubar toda a aplicação, pois todos os componentes estão interligados. Isso difere de arquiteturas mais distribuídas, nas quais um erro em um serviço não necessariamente afeta o restante do sistema.
5. **Ciclo de Vida Complexo**: o ciclo de vida da aplicação (desenvolvimento, testes e implantação) se torna mais complexo à medida que o sistema cresce. Pequenas mudanças exigem testes em toda a aplicação, e a implantação pode ser lenta e arriscada.

Aplicações monolíticas são comuns em softwares legados e em sistemas que começaram como pequenas soluções e foram crescendo. Exemplos incluem sistemas de CRM e ERP, em que toda a funcionalidade está em um único bloco de código. Com o passar do tempo, esse sistema se desenvolveu de tal forma que a arquitetura voltada para um projeto simples e pequeno não suporta mais um sistema robusto e complexo.

Segundo a IBM (2024), a arquitetura de microsserviços é uma abordagem moderna de design de software, na qual uma aplicação é construída como um conjunto de serviços pequenos e independentes. Cada serviço tem uma função específica e pode ser desenvolvido, implantado e escalado de forma independente. Essa arquitetura contrasta com a arquitetura monolítica tradicional, em que todos os componentes de um sistema estão interligados e executados como uma única unidade.

Em uma arquitetura de microsserviços, a aplicação é dividida em serviços menores, cada um representando uma funcionalidade de negócios distinta. Esses serviços geralmente se comunicam por meio de APIs ou Mensageria, permitindo que cada serviço seja desenvolvido com sua própria lógica e dados. Os microsserviços são independentes e podem ser escritos em diferentes linguagens de programação, usando diferentes tecnologias de armazenamento de dados, dependendo das necessidades do serviço (IBM, 2024).

Segundo Abgaz *et al*. (2023), a arquitetura de microsserviços favorece a decomposição de sistemas em pequenos componentes independentes, que podem ser invocados conforme necessário. Essa abordagem oferece benefícios como maior escalabilidade e frequência de implantação.

Apesar de a arquitetura de microsserviços ser amplamente considerada uma solução moderna e eficiente, com benefícios como escalabilidade e independência de componentes, ela também apresenta pontos negativos. Segundo a IBM (2024), a complexidade operacional, dificuldade em gerenciar a comunicação entre serviços e a necessidade de uma infraestrutura robusta são desafios significativos. De acordo com a Red Hat (2024), podemos citar mais alguns pontos negativos dessa arquitetura:

1. **Complexidade operacional**: a arquitetura de microsserviços traz uma complexidade adicional ao gerenciamento da infraestrutura. A implantação, gerenciamento e monitoramento de muitos serviços independentes requer ferramentas avançadas, como Kubernetes e sistemas de monitoramento distribuídos, para garantir que todos os serviços funcionem corretamente.
2. **Gerenciamento de dados**: como os microsserviços têm bancos de dados independentes, garantir a consistência de dados entre serviços pode ser desafiador. Além disso, transações distribuídas podem ser complexas, e os desenvolvedores devem lidar com a consistência eventual em vez da consistência imediata.
3. **Comunicação entre serviços**: a comunicação entre os microsserviços pode adicionar latência e aumentar a probabilidade de falhas na rede. Isso exige que os desenvolvedores implementem mecanismos de tolerância a falhas, como *retries* e *circuit breakers*, para evitar a propagação de falhas em toda a aplicação.
4. **Sobrecarga de desenvolvimento**: embora a independência dos serviços ofereça muitos benefícios, também cria uma sobrecarga. Cada serviço precisa ser projetado, testado e monitorado individualmente, o que pode aumentar a complexidade do desenvolvimento.
5. **Gerenciamento de dependências**: com muitos microsserviços em uma aplicação, as dependências entre serviços podem se tornar difíceis de gerenciar, principalmente quando diferentes serviços são mantidos por equipes distintas. O alinhamento de versões e a compatibilidade entre serviços são desafios importantes.

Segundo a Atlassian (2024), a arquitetura de microsserviços oferece diversos pontos positivos, como a escalabilidade independente de componentes, que permite ajustar recursos conforme a demanda de cada serviço. Além disso, facilita o desenvolvimento paralelo, pois equipes podem trabalhar em diferentes microsserviços sem interferir umas nas outras. A implantação contínua também se beneficia, uma vez que atualizações e correções podem ser feitas em serviços específicos sem interromper a aplicação inteira. De acordo com a Red Hat (2024), podemos citar mais alguns pontos positivos dessa arquitetura:

1. **Escalabilidade**: permite aumentar a capacidade de serviços específicos conforme a demanda, otimizando o uso de recursos.
2. **Desenvolvimento e implantação mais rápidos**: equipes podem trabalhar paralelamente em diferentes serviços, reduzindo o tempo de entrega e acelerando os ciclos de DevOps.
3. **Resiliência**: falhas são isoladas em serviços específicos, evitando que todo o sistema seja comprometido.
4. **Flexibilidade tecnológica**: cada serviço pode usar tecnologias diferentes conforme sua necessidade, aumentando a eficiência e adaptabilidade.
5. **Manutenção facilitada**: alterações podem ser feitas em um serviço sem afetar os demais, tornando a manutenção mais simples e segura.

PROCEDIMENTOS METODOLÓGICOS PARA MIGRAÇÃO DE UM SISTEMA DE CRM MONOLITO PARA UMA ARQUITETURA DE MICROSSERVIÇOS

Esta seção apresenta como deve ser iniciada a migração de um sistema de CRM Monolito para uma Arquitetura de Microsserviços.

INICIANDO A MIGRAÇÃO

O início da migração de um sistema monolítico para uma arquitetura de microsserviços requer planejamento cuidadoso e uma abordagem incremental. Antes de iniciar a migração, é essencial avaliar o sistema monolítico existente. Isso inclui identificar quais partes do sistema são mais críticas, frequentemente modificadas ou apresentam gargalos de desempenho. Compreender o comportamento atual do sistema ajuda a definir as prioridades para a migração.

Segundo o Google Cloud (2024), a arquitetura de microsserviços depende da separação lógica do sistema em diferentes domínios de negócios, conhecidos como *bounded contexts*. Isso envolve a definição de quais partes do sistema serão extraídas e transformadas em serviços independentes. Um bom ponto de partida é identificar módulos que já estão relativamente desacoplados no monolito e podem ser migrados mais facilmente.

É recomendado começar com uma funcionalidade pequena e de baixo risco, que pode ser migrada sem comprometer a integridade do sistema como um todo. Isso ajuda a testar a nova arquitetura e ajustá-la antes de migrar para partes maiores e mais críticas.

DEFININDO METODOLOGIA DE MIGRAÇÃO

A forma mais aceita para realizar a migração de um monolito para uma arquitetura de microsserviços é utilizar o Strangler Pattern. O Strangler Pattern é um padrão de design usado para facilitar a migração gradual de um sistema monolítico para uma arquitetura mais moderna, sem interromper o funcionamento do sistema durante o processo. Segundo a Microsoft (2024), a ideia do Strangler Pattern é adicionar novas funcionalidades ao sistema em paralelo ao sistema legado, substituindo gradualmente partes antigas por novas versões, até que o monolito seja completamente substituído. A aplicação desse padrão se dá em alguns passos:

1. **Divisão por funcionalidade**: o sistema monolítico é gradualmente dividido em funcionalidades ou serviços menores, que podem ser desenvolvidos e implantados de maneira independente.
2. **Envolvimento com APIs**: cada funcionalidade antiga do monolito é envolvida ou redirecionada por APIs ou rotas intermediárias, permitindo que as novas implementações possam substituir a funcionalidade sem impactar diretamente o sistema legado.
3. **Transição gradual**: o sistema continua operando enquanto as novas versões dos serviços são desenvolvidas. Aos poucos, as partes do monólito que foram substituídas podem ser "desligadas" ou eliminadas.
4. **Convergência completa**: quando todas as partes do sistema legado foram substituídas, o sistema monolítico original pode ser completamente descartado.

RECOMENDAÇÕES PARA MIGRAÇÃO

Seguindo o Strangler Pattern, algumas recomendações devem ser levadas em conta antes de iniciar a migração. Pensando de forma incremental, as novas funcionalidades devem ser criadas aos poucos; as novas substituem as antigas, seguindo uma ordem de mais desacoplado e menos crítico para mais acoplado e mais crítico.

Manter a compatibilidade do novo sistema com o legado é de extrema importância até o fim da migração, para isso existem algumas recomendações que serão apresentadas nos subtópicos: 6.2.1, 6.2.2, 6.2.3, 6.2.4, 6.2.5 e 6.2.6. Lembrando que são recomendações e não diretrizes para serem seguidas ao extremo, todas as recomendações podem ser adaptadas e ajustadas conforme o cenário da migração.

ABORDAGEM

Existem três principais abordagens para migrar o sistema: a primeira abordagem adota a ideia de uma arquitetura desacoplada com uma base de dados unificada; a segunda adota uma arquitetura desacoplada com uma base de dados individual para cada microsserviço; e a terceira adota a migração para uma arquitetura desacoplada sem alterar a base de dados. Neste trabalho, adotamos a segunda abordagem, na qual cada microsserviço terá sua base de dados individual. Posteriormente, esta pesquisa poderá ser evoluída para contemplar todas as abordagens.

DESENVOLVIMENTO DE NOVA FUNCIONALIDADE

Seguindo o Strangler Pattern, devemos criar uma nova funcionalidade no sistema modernizado, que irá substituir a do sistema legado. Segundo a Microsoft (2024), essa nova funcionalidade deve conter as mesmas funcionalidades básicas da versão antiga, mas pode ser expandida para aproveitar as vantagens das novas tecnologias. Ela só passa a existir para o cliente quando estiver totalmente finalizada.

SINCRONIZAÇÃO DE DADOS ENTRE OS SISTEMAS

Existem duas abordagens para os microsserviços, a primeira abordagem adota a ideia de uma arquitetura desacoplada com uma base de dados unificada; a segunda adota uma arquitetura desacoplada com uma base de dados individual para cada microsserviço.

Este artigo adota a segunda abordagem, a qual necessita de atenção na parte de sincronização. Durante o período de transição, você precisará manter os dados sincronizados entre o novo sistema e o sistema legado. Existem duas abordagens comuns para isso:

- **Sincronização em tempo real**: cada vez que um novo cadastro é feito no novo sistema, os dados são replicados no banco de dados legado por meio de APIs ou mecanismos de sincronização de dados (eventos).
- **Sincronização periódica**: dependendo do volume de dados e da criticidade, você pode optar por sincronizar dados em intervalos específicos (ex.: a cada noite).

CARGA INICIAL

A carga inicial de dados é essencial para assegurar que o novo sistema tenha uma base de dados consistente e alinhada com o sistema antigo antes de começar a funcionar. Existem várias formas de realizar essa transferência de dados, como o uso de scripts de migração ou criação de serviços temporários para trazer os dados do sistema legado. Outras abordagens também são possíveis, e a escolha deve ser feita com base nas características do sistema em migração.

TESTES E MONITORAMENTO

Antes de desligar completamente a tela do sistema legado, é essencial realizar testes extensivos para garantir que o novo sistema funcione corretamente. Monitore a integração entre os dois sistemas para detectar problemas com a sincronização de dados ou falhas de comunicação.

DESLIGAMENTO DA FUNCIONALIDADE LEGADA

Uma vez que a nova funcionalidade esteja totalmente funcional e os dados estejam corretamente integrados, podemos desativar a funcionalidade do sistema legado, removendo o roteamento para essa tela e mantendo apenas o novo sistema para essa funcionalidade.

REPLICAÇÃO DE DADOS ENTRE MICROSSERVIÇOS

Segundo o Google Cloud (2024), na arquitetura de microsserviços, a replicação de dados é um ponto-chave. Como cada microsserviço nasce com sua própria base de dados, a replicação é necessária para que um serviço possa conhecer os dados de outro microsserviço.

Quando um serviço necessita dos dados de outro serviço, é feito um controle para trazer somente os dados necessários para o seu funcionamento. Esses dados devem ser replicados via evento conforme a demanda, já a carga inicial de um microsserviço para outro pode ser feita através de bibliotecas que acessam as bases e replicam os dados diretamente.

Essas recomendações geralmente são utilizadas com Strangler Pattern, mas não são regras a serem seguidas, dependendo do sistema algumas recomendações podem ser alteradas, melhoradas ou até reduzidas.

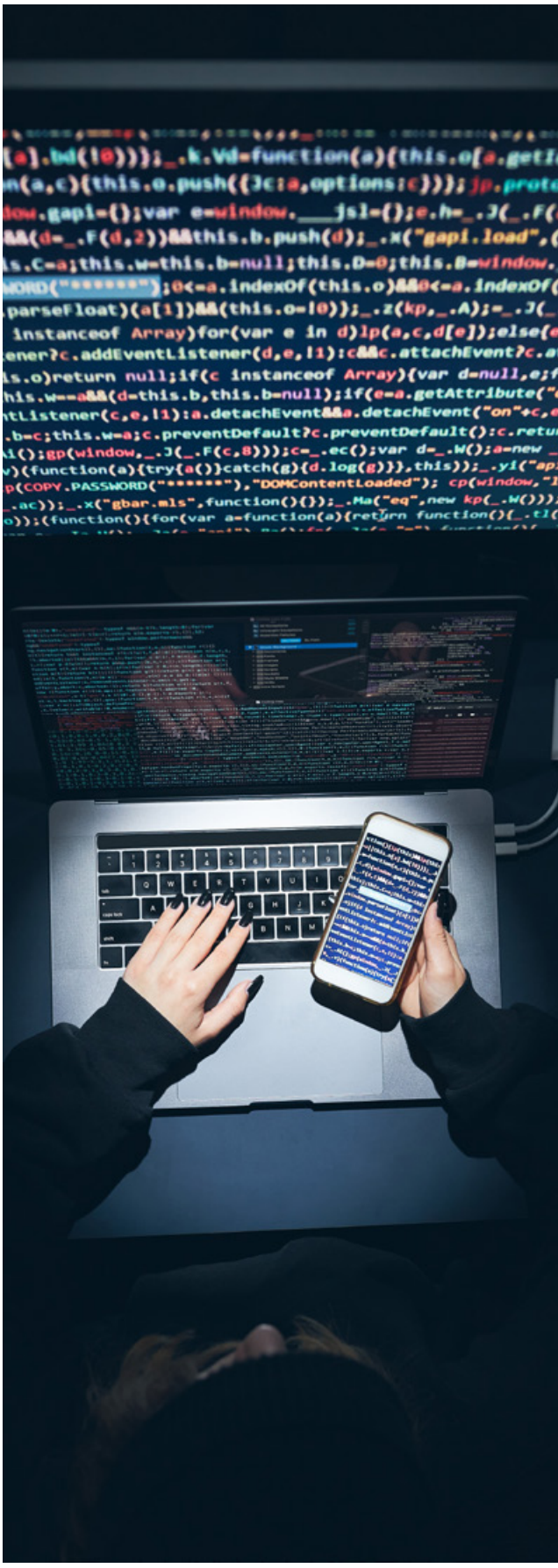
DEFININDO METODOLOGIA DE COMUNICAÇÃO

Existem muitas formas de comunicação entre microsserviços. Podemos separá-las em dois grupos, síncronas e assíncronas. É importante que ambas sejam adotadas para a criação de um microsserviço, talvez um microsserviço não use ambos os métodos, mas com certeza se comunicará com outros através deles.

No contexto de comunicação síncrona, o REST se destaca como a abordagem mais utilizada. Ele é amplamente adotado na comunicação entre microsserviços, utilizando os protocolos HTTP/HTTPS. Cada serviço expõe suas funcionalidades por meio de APIs RESTful, permitindo que outros serviços realizem chamadas HTTP para interagir com essas APIs de forma eficiente.

Como comunicação assíncrona, a mensageria é o meio mais comum; ferramentas como RabbitMQ, Apache Kafka ou ActiveMQ permitem que os microsserviços se comuniquem enviando e recebendo mensagens através de filas. Cada serviço coloca mensagens em uma fila, e outros serviços consomem essas mensagens de forma assíncrona.

Segundo o Google Cloud (2024), é necessário implementar um meio de comunicação em comum entre os serviços, APIs que podem garantir que os microsserviços possam se comunicar de maneira eficiente. Durante o processo de migração, os microsserviços precisarão interagir tanto entre si quanto com o sistema monolítico.



DEFININDO ORDEM DE CONVERSÃO DE MÓDULOS

Segundo a Microsoft (2024), o Strangler Pattern indica a migração dos módulos mais desacoplados e menos críticos para os menos desacoplados e mais críticos. Nem todos os sistemas de CRM possuem os mesmos módulos, mas existem os módulos principais que são comuns em todos os CRMs: Cadastros Básicos, Campanhas, Contas, Oportunidades e Análise e Relatórios são os mais comuns. Vamos definir uma possível ordem para a migração de módulos, seguindo o fluxo dito anteriormente.

MÓDULO DE CADASTROS BÁSICOS

O CRM é composto por alguns módulos, um deles é o módulo de Cadastros Básicos, esse módulo não é algo exclusivo como um módulo de Contas ou Oportunidades, mas sim um módulo que abrange os cadastros gerais utilizados em um CRM. Esse módulo inclui muitas funcionalidades, e isso é variável entre determinados sistemas de CRM, mas como os principais podemos citar:

1. Endereço (País, Estado, Cidade).
2. Tipo da Conta.
3. Tipo da Oportunidade.
4. Tabela de Preço.
5. Empresa Filial.
6. Transportadora.

Existem mais componentes do módulo de Cadastros Básicos, os citados são os mais comuns entre sistemas de CRM. Esse módulo é considerado o mais desacoplado do sistema, pois esses cadastros geralmente não possuem relacionamentos necessários para sua existência. O fato de não possuir muitos relacionamentos ajuda, pois é possível realizar a migração dessas funcionalidades facilmente. Seguindo o padrão Strangler Pattern, vamos começar a construir os microsserviços responsáveis por cada funcionalidade. A separação dos microsserviços geralmente é dada por domínios, os domínios geralmente são representados pelos módulos do sistema.

No caso dos cadastros básicos, ele é um módulo composto por funcionalidades de outros módulos. Exemplo: o cadastro do Tipo de Conta pertence ao domínio do módulo de contas, o cadastro do Tipo de Oportunidade pertence ao domínio do módulo de Oportunidades e assim sucessivamente. A separação dos domínios deve ser feita da melhor maneira de acordo com o sistema de CRM, devemos analisar a forma mais adequada para a situação. Após começarmos a criar os microsserviços, devemos nos atentar aos pontos vistos anteriormente. Após finalizar a criação dos microsserviços e telas, vamos começar a pensar na migração de dados.

A carga inicial é um aspecto fundamental na migração de sistemas e requer atenção especial. É essencial garantir que a estrutura básica do sistema legado também esteja presente no novo sistema. No entanto, algumas funcionalidades anteriormente existentes podem não ser mais necessárias na nova solução. Nesses casos, é importante analisar a situação e definir a melhor abordagem para transferir apenas os dados relevantes.

Para esse processo, podemos implementar um controle dedicado por meio de microsserviços específicos, responsáveis tanto pela carga inicial dos dados quanto pela manutenção da sincronização entre os sistemas. Essa estratégia garante uma transição mais eficiente e organizada.

Os dados serão mantidos em sincronia até a possibilidade de extinção da antiga funcionalidade, dessa forma, é possível reduzir custos e manutenção do sistema legado. Vale ressaltar que, antes de qualquer medida de extinção do módulo/funcionalidade antiga, todos os testes que validam as métricas de uso devem ter sido concluídos. Se, por algum motivo, existirem registros de usabilidade do sistema legado, ações devem ser tomadas para migrar o uso para a nova funcionalidade. Avisos a clientes e prazos razoáveis ajudam nesse processo, assim como redirecionamento de telas em caso de acessos diretos.

MÓDULO DE CAMPANHAS/MARKETING

Entre os módulos, o de campanhas é um de fácil migração também, ele possui vínculos com módulos mais importantes, como contas, por exemplo, mas ainda assim são poucos os módulos relacionados. Segundo a Salesforce (2024), esse módulo é responsável por conter toda a parte de envio de e-mails marketing, abordagens de mala direta digital, entre outros.

O módulo de campanhas é extremamente dependente de agendamentos e automações para seu correto funcionamento. Esses pontos devem ser tratados com atenção, uma vez que os microsserviços responsáveis poderão receber muitas requisições, e cada uma delas poderá levar muito tempo para ser processada, dependendo do público-alvo abordado.

A escalabilidade neste ponto deve estar bem definida com os padrões dos microsserviços. Caso haja má construção da arquitetura dos microsserviços, o novo sistema poderá sofrer com sobrecarga e custos mais elevados do que o esperado. Por serem campanhas, é comum possuir arquivos grandes, assim como imagens e arquivos que serão enviados. Todas essas mídias devem ser salvas em um FTP separado dos microsserviços, o que não acontece em uma arquitetura monolítica, já que tudo fica em um só local. Nesses casos, os microsserviços apenas vão armazenar as URLs para acessarem os arquivos quando necessário, uma data para a exclusão desses dados também é de extrema importância e pode ser decidida com o cliente dependendo da necessidade (Salesforce, 2024).

MÓDULO DE CONTAS E OPORTUNIDADES

Segundo a Salesforce (2024) e a RD Station (2024), os módulos de contas e oportunidades são, com certeza, os mais importantes dentro de um CRM. É no módulo de contas que todos os dados dos clientes são armazenados, além de que o módulo de contas possui uma visão 360° de todo o CRM. Todas as funcionalidades que existem em um CRM só existem para atender um cliente, no caso, uma conta. Esse módulo deve ser migrado juntamente com o módulo de oportunidades, que é responsável pelas propostas, orçamentos e revisões enviadas aos clientes.

Nesta etapa, o módulo de contas já estará em um estado bem inicial, pois precisou atender o módulo de campanhas de forma mais enxuta. Nesse momento, o microsserviço responsável pelo módulo de contas ficará consideravelmente grande e, dependendo da necessidade, mais serviços podem ser criados para separar melhor o módulo de contas. Durante o processo de migrar o módulo de contas, o módulo de oportunidades também pode ser iniciado, já que o módulo de oportunidades é dependente do módulo de contas para realizar suas ações, assim como o módulo de contas também necessita dos dados do módulo de oportunidades.

Por esses dois módulos serem os maiores, o cuidado deve ser redobrado, testes de performance, carga e estresse devem ser muito mais intensificados, é importante que ao final de cada migração modular testes de uso de memória e recursos sejam feitos, muitos gargalos podem ser identificados dessa forma, o que ajuda a tornar o microsserviço menos custoso e mais eficiente.

MÓDULO DE ANÁLISE E RELATÓRIOS

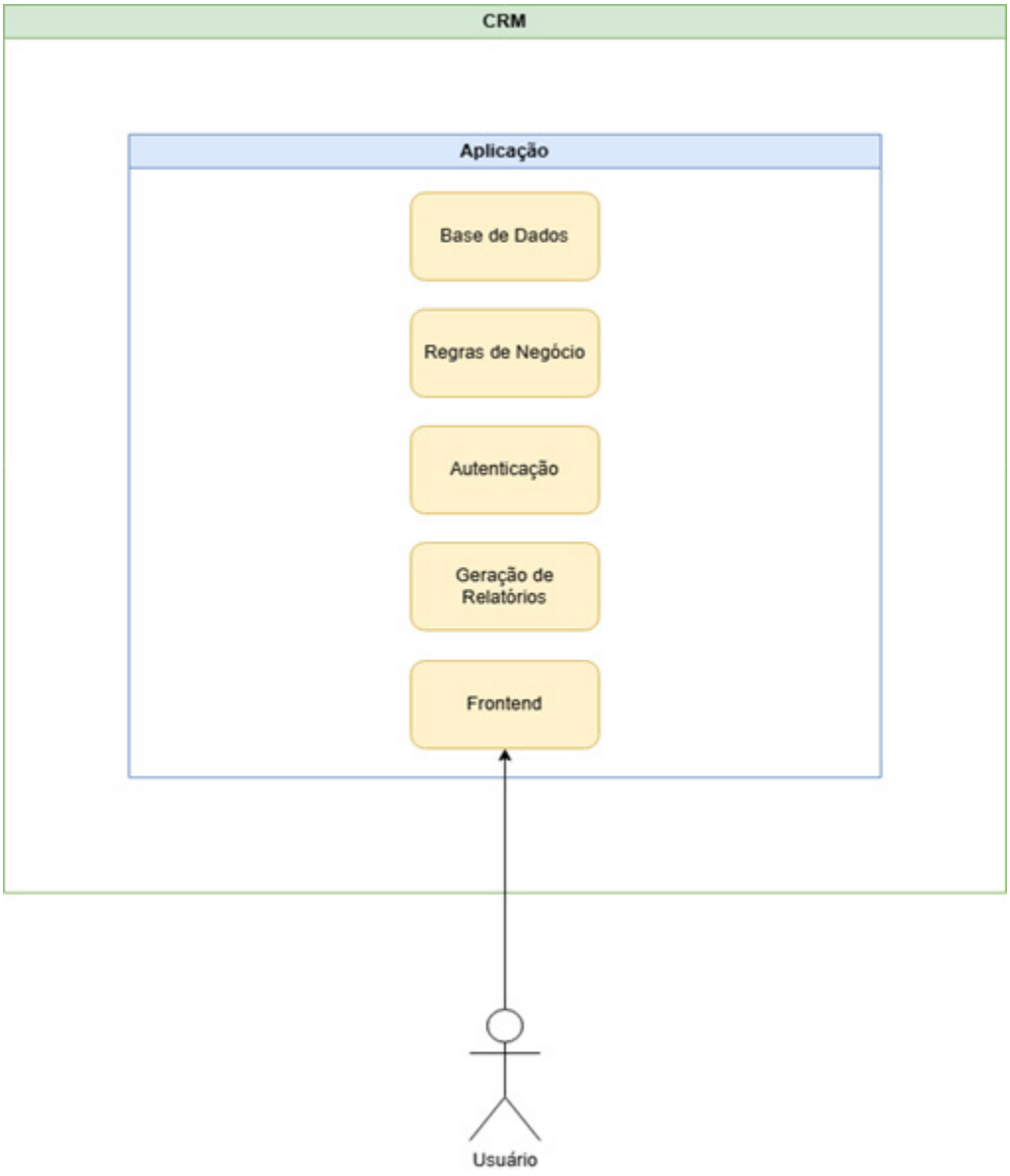
Análise e relatórios é um módulo de extrema importância, segundo a RD Station (2024), é através dele que os usuários podem ver como estão os relacionamentos com seus clientes, quantidades de vendas, prospecções, insights de e-mails de marketing e muitos outros dados analíticos.

A migração desse módulo é muito simples também, ela está por último pois precisa dos dados de todos os outros módulos do sistema. Para migrar esse módulo, os mesmos princípios abordados anteriormente devem ser seguidos, com atenção para telemetria e logs de uso; esses dados também são de extrema importância para o sistema.

MIGRAÇÃO FINALIZADA

Após concluir a migração do sistema, podemos observar uma mudança arquitetural significativa. Vários microsserviços irão existir, cada um com sua responsabilidade e recursos necessários para funcionar corretamente. Abaixo, segue um exemplo dessa mudança, na qual a primeira imagem é da antiga arquitetura e a segunda é a nova arquitetura:

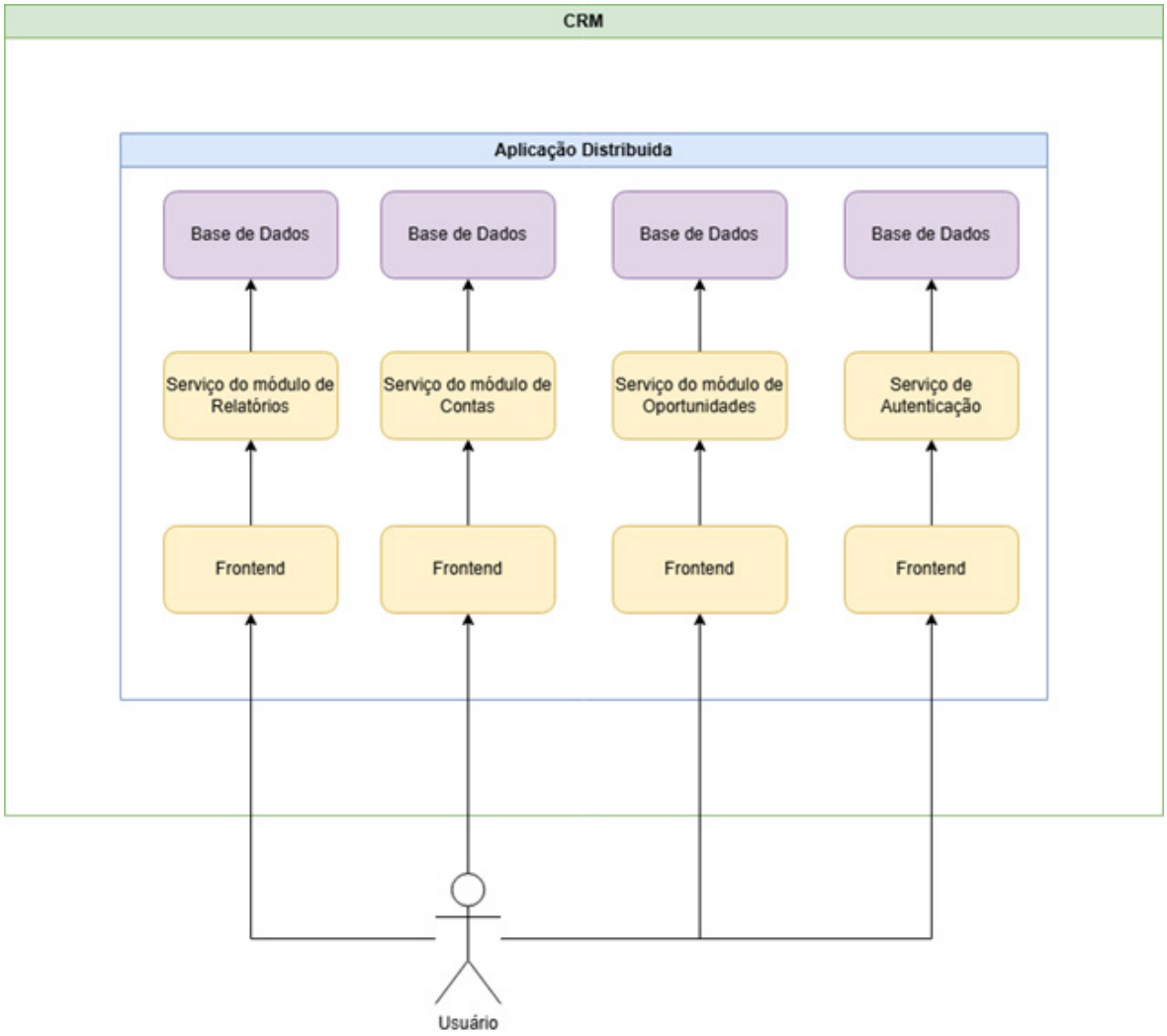
Figura 1. Diagrama de uma arquitetura monolítica



Fonte: o autor.

Observação referente à figura 1: os retângulos amarelos na imagem representam partes de uma mesma aplicação. Cada um dos retângulos pode ser responsável por um módulo, uma regra, autenticação etc.

Figura 2. Diagrama de uma arquitetura de microsserviços



Fonte: o autor.

Observação referente à figura 2: os retângulos amarelos na imagem representam microsserviços individuais, os roxos representam as bases de dados de cada serviço. Cada um dos retângulos é individualmente funcional, eles podem ser responsáveis por um módulo, uma regra, autenticação etc.

CONSIDERAÇÕES FINAIS

A migração de um sistema de CRM monolítico para uma arquitetura de microsserviços representa uma transformação essencial para empresas que buscam escalabilidade, flexibilidade e maior agilidade no desenvolvimento de suas soluções. Conforme abordado neste artigo, embora a arquitetura monolítica seja mais simples em estágios iniciais, suas limitações tornam-se evidentes à medida que os sistemas crescem, dificultando a manutenção e o desenvolvimento de novas funcionalidades.

A adoção da arquitetura de microsserviços oferece vantagens como a independência dos componentes, escalabilidade sob demanda, resiliência a falhas e maior velocidade nas entregas. No entanto, essa transição exige um planejamento cuidadoso, sendo o *Strangler Pattern* a metodologia mais indicada por permitir uma migração gradual e de baixo risco. Iniciar a migração por módulos menos críticos e mais desacoplados, como o de cadastros básicos, possibilita validar a nova arquitetura antes de avançar para módulos mais complexos, como o de contas e oportunidades.

Durante o processo de migração, a sincronização de dados entre o sistema legado e os novos microsserviços é essencial para garantir a continuidade operacional. Além disso, a comunicação entre os serviços deve ser cuidadosamente planejada, utilizando APIs RESTful para interações síncronas e mensageria para comunicações assíncronas. Testes rigorosos de desempenho, carga e integração são fundamentais para assegurar a estabilidade da nova arquitetura.

Conclui-se que, ao adotar uma arquitetura de microsserviços, as empresas ganham não apenas em escalabilidade e desempenho, mas também em competitividade, ao responder de forma mais ágil às demandas do mercado. O sucesso dessa transformação depende de uma execução estratégica e incremental, permitindo que a empresa aproveite os benefícios da nova arquitetura sem comprometer a operação durante o processo de transição.

REFERÊNCIAS

ABGAZ, Y. *et al*. Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. **IEEE Transactions on Software Engineering**, v. 49, n. 8, p. 4213-4242, ago. 2023. Disponível em: https://ieeexplore.ieee.org/document/10160171. Acesso em: 20 set. 2024.

AMAZON WEB SERVICES. **Diferença entre arquitetura monolítica e microsserviços**. AWS, 2024. Disponível em: https://aws.amazon.com/pt/compare/the-difference-between-monolithic-and-microservices-. Acesso em: 2 out. 2024.

ATLASSIAN. Microservices vs. Monólito. **Atlassian**, 2024. Disponível em: https://www.atlassian.com/br/microservices/microservices-architecture/microservices-v. Acesso em: 11 out. 2024.

BLINOWSKI, G.; OJDOWSKA, A.; PRZYBYŁEK, A. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. **IEEE**, 2022. Disponível em: https://ieeexplore.ieee.org/abstract/document/9717259. Acesso em: 9 set. 2024.

CLICKUP. Componentes de CRM. **ClickUp**, 2024. Disponível em: https://clickup.com/pt-BR/blog/139401/componentes-de-crm. Acesso em: 29 set. 2024.

DRAGONI, N. *et al*. **Microservices**: Science and Engineering. 1. ed. Cham: Springer, 2022.

FERRER-ESTÉVEZ, M.; CHALMETA, R. Sustainable customer relationship management. **Marketing Intelligence & Planning**, v. 41, n. 2, p. 244-262, 2023. Disponível em: https://www.emerald.com/insight/content/doi/10.1108/mip. Acesso em: 21 set. 2024.

GIACOM, J. G. S. **A contribuição da atividade de relações públicas nas startups**: diretrizes de relacionamento e seu impacto na reputação do negócio. 2021. Trabalho de Conclusão de Curso (Graduação) – Faculdade Cásper Líbero, São Paulo, 2021. Disponível em: abrapcorp.org.br. Acesso em: 17 fev. 2025.

GOOGLE CLOUD. Refatoração de monólitos para microsserviços. **Google Cloud**, 2024. Disponível em: https://cloud.google.com/architecture/microservices-architecture-refactoring-monoliths?hl=pt-br. Acesso em: 19 out. 2024.

IBM. O que são microsserviços? **IBM**, 2024. Disponível em: https://www.ibm.com/br-pt/topics/microservices. Acesso em: 11 out. 2024.

IMASTERS. Strangler Pattern: migrar monolito para microsserviços. **iMasters**, 2024. Disponível em: https://imasters.com.br/apis-microsservicos/strangler-pattern-migrar-mon. Acesso em: 4 nov. 2024.

MATHLIMMA. Microsserviços: um estudo de caso - Amazon e Netflix. **Medium**, 2021. Disponível em: https://mathlimma.medium.com/microsservi%C3%A7os-um-estudo-de-caso-amazon-e-netflix-3582648540a0. Acesso em: 17 fev. 2025.

MICROSOFT. Strangler Fig pattern. **Microsoft Learn**, 2024. Disponível em: https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig. Acesso em: 19 out. 2024.

RD STATION. CRM. **RD Station**, 2024. Disponível em: https://www.rdstation.com/produtos/crm/. Acesso em: 4 nov. 2024.

RED HAT. What are microservices? **Red Hat**, 2024. Disponível em: https://www.redhat.com/en/topics/microservices/what-are-microservices. Acesso em: 21 out. 2024.

SALESFORCE. CRM: Gestão de Relacionamento com o Cliente. **Salesforce**, 2024. Disponível em: https://www.salesforce.com/br/crm/. Acesso em: 29 set. 2024.

SHRESTHA, D. **Navigating Software Architecture**: Evaluating Monolithic Architectures in Modern Development. 2024. Trabalho de Conclusão de Curso (Graduação) – Centria University of Applied Sciences, [*s. l.*], 2024. Disponível em: https://www.theseus.fi/handle/10024/858838. Acesso em: 21 set. 2024.

STRADOLINI, C. J. **Migração de sistemas monolíticos para microsserviços**: Estudo de caso de migração de um módulo de pagamentos de e-Commerce. 2022. Trabalho de Conclusão de Curso (Graduação) – Universidade Federal do Rio Grande do Sul, Porto Alegre, 2022. Disponível em: lume.ufrgs.br. Acesso em: 17 fev. 2025.

TECHTARGET. Monolithic architecture. **TechTarget**, 2024. Disponível em: https://www.techtarget.com/whatis/definition/monolithic-architecture. Acesso em: 21 out. 2024.