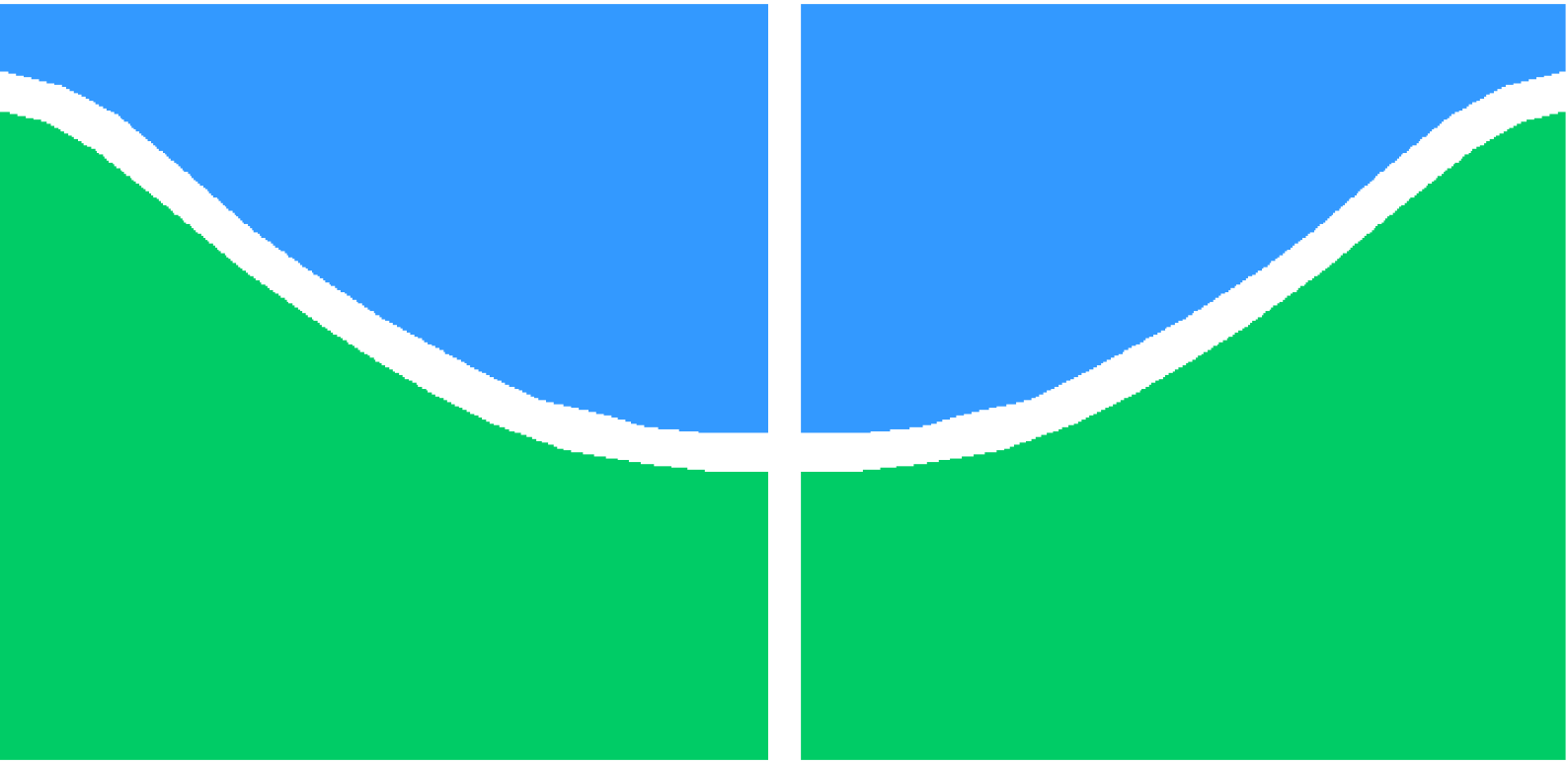

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Design Arquitetural de Software aplicado a Sistemas de Larga Escala: Revisão de Literatura Cinzenta

Autor: Gabriela Alves da Gama

Orientador: Dra. Carla Silva Rocha Aguiar

Brasília, DF

2021

Gabriela Alves da Gama

Design Arquitetural de Software aplicado a Sistemas de Larga Escala: Revisão de Literatura Cinzenta

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Dra. Carla Silva Rocha Aguiar

Brasília, DF

2021

Gabriela Alves da Gama

Design Arquitetural de Software aplicado a Sistemas de Larga Escala: Revisão de Literatura Cinzenta / Gabriela Alves da Gama. – Brasília, DF, 2021-
72 p.: il.

Orientador: Dra. Carla Silva Rocha Aguiar

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2021.

1. Arquitetura de Software. 2. Software Design. 3. Sistemas de larga escala I. Dra. Carla Silva Rocha Aguiar. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Design Arquitetural de Software aplicado a Sistemas de Larga Escala: Revisão de Literatura Cinzenta

CDU 02:141:005.6

Gabriela Alves da Gama

Design Arquitetural de Software aplicado a Sistemas de Larga Escala: Revisão de Literatura Cinzenta

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Trabalho aprovado. Brasília, DF, 01 de Janeiro de 2022:

Dra. Carla Silva Rocha Aguiar
Orientador

Dra. Milene Serrano
Convidado 1

Dr. Ricardo Ajax Dias Kosloski
Convidado 2

Brasília, DF
2021

Resumo

A Arquitetura de Software em Sistemas de Larga Escala é complexa, por isso há muitas estratégias sendo utilizadas pelos práticos da Engenharia de Software para lidar com essa complexidade. O objetivo desse trabalho é mapear essas estratégias por meio de uma Grey Literature Review, que trata-se de uma variante Revisão Sistemática de Literatura adequada para incluir materiais tais como artigos e documentações mantidos por profissionais da área em blogs e sites. A metodologia adotada justifica-se pelo objetivo de sintetizar dados que refletem ao que é feito no mercado de software, e esse tipo de informação é encontrada em materiais de literatura cinzenta devido a sua natureza não estruturada.

Palavras-chaves: arquitetura de software, software design, sistemas de larga escala, literatura Cinzenta

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

GLR	Grey Literature Review
RSL	Revisão Sistemática de Literatura
NALSD	Non-abstract Large System Design
DDD	Domain-drive Design
GT	Grounded Theory

Sumário

Introdução

A Arquitetura de Software é tão importante para o desenvolvimento de software quanto uma planta na construção de uma casa. Com o conhecimento do que deve ser feito, é possível aplicar um design arquitetural adequado para prover a solução. Isso envolve muito conhecimento técnico, pois tem grande potencial para ser uma etapa divisora do fracasso e sucesso do desenvolvimento de um sistema (QUORA, 2018e).

Com uma abordagem ágil, o planejamento arquitetural é incremental, vantagem principalmente para software de larga escala. Esse tipo de sistema, em geral, é caracterizado por suportar múltiplos e simultâneos acessos, em elevado volume, tornando o fluxo de dados complexo e com comunicação entre subsistemas dentro de uma rede distribuída.

Ser capaz de suportar um aumento de carga sem necessidade de ser completamente remodelado, tolerar falhas de seus subsistemas, estar inserido em um pipeline de desenvolvimento de infraestrutura complexa e contar com a participação de mais de uma equipe no desenvolvimento dos subsistemas e integração também são características observadas em software desse porte. No contexto de arquitetura de software de larga escala, a solução técnica evolui a medida que as necessidades negociais também evoluem. Além disso, é possível utilizar técnicas de arquitetura para aprimorar sistemas legados (CINTRA; VENDRAMEL, 2019).

Para suprir a necessidade de escalabilidade dos sistemas que são usados globalmente, os envolvidos no projeto arquitetural lidam com vários problemas. De modo geral, o projeto de um sistema assim envolve um número expressivo de stekholders, o que tem causado dificuldades no alinhamento de requisitos e regras de negócio. A resiliência dos subsistemas é uma grande preocupação, visto que falhas podem causar perdas ou inconsistências nos dados caso o sistema não esteja programado para lidar com isso.

A ideia da possibilidade de distribuir a responsabilidade de funcionalidades do sistema por várias equipes em cidades ou países diferentes é bem atrativa, mas a integração desses serviços é complexa o suficiente para ser inviabilizada caso não haja recursos suficientes a serem investidos. Boas ferramentas para automação de testes, deploy e integração contínua também são requeridas (SOFTWAREONE, 2020).

Como é uma área de conhecimento ainda emergente, parte do conhecimento e lições aprendidas atualmente sobre arquitetura de software de larga escala são amplamente divulgadas em blogs, canais, documentações, vídeos e fóruns. Essas fontes de conhecimento, sem processo de revisões, são conhecidos como literatura cinzenta, ou Grey literature. Essas informações são muito valiosas, pois parte desse conhecimento não está presente na literatura acadêmica ainda. Essa é a motivação para realizar o estudo desse trabalho.

O objetivo geral desse trabalho é identificar quais são as estratégias que os práticos da engenharia de software estão adotando no design arquitetural de sistemas de larga escala. Os objetivos específicos são identificar os maiores desafios enfrentados pelos práticos e coletar dados sobre as soluções usadas no mercado atual para vencer esses desafios.

A metodologia adotada é a de uma revisão sistemática de literatura adaptada para o uso de literatura cinzenta (WEN et al., 2020). A partir de boas práticas de Grounded Theory (STOL; RALPH; FITZGERALD, 2016), são usadas técnicas de *coding*, não afim de desenvolver uma teoria, mas para coletar dados sobre desafios, métodos, práticas e entre outros assuntos relacionadas ao modo como a Arquitetura de Software é pensada por práticos da área de Engenharia de Software, no contexto de software de larga escala, de forma sistemática.

A primeira seção desta monografia é destinada ao referencial teórico que contextualiza o conteúdo do estudo. Em seguida é apresentada a proposta, que explica a metodologia da GLR e detalha o protocolo. Por fim, os resultados obtidos, a discussão sobre as principais descobertas e a conclusão do trabalho.

Referencial Teórico

1.1 Arquitetura de Software

O termo “Arquitetura de Software” recebeu algumas definições ao longo dos anos, tais como “A organização fundamental de um sistema incorporada em seus componentes, o relacionamento entre eles e entre o ambiente, e os princípios que conduzem seu design e evolução” (IEEE. . . , 2000); “A arquitetura de software de um programa ou de um sistema computacional é a estrutura, ou as estruturas, do sistema a qual compreende elementos de software, as propriedades visíveis externamente desses elementos, e os relacionamentos entre eles” (BASS; CLEMENTS; KAZMAN, 2003); “A arquitetura de software de um programa ou sistema computacional é a representação do sistema que auxilia no entendimento de como ele vai se comportar” (UNIVERSITY, 2017). Porém, uma definição simples e aceitável feita pelo palestrante Ralph Johnson é que a 'Arquitetura se refere a algo importante, não importa o que seja'. Esse raciocínio é muito coerente com o que compõe uma das principais decisões arquiteturais, que é definir elementos fundamentais na composição de um software (FOWLER, 2019a).

Essas definições importantes devem ser tomadas de modo consciente, embasada nas grandes descobertas da comunidade de software e em sólida experiência. Um profissional com esse grau de expertise é capaz de atuar como Arquiteto de Software.

1.2 Arquiteto de Software

A descrição feita por (UFPE, 2006) sobre esse ou essa profissional acerca de suas responsabilidades cita a 'tomada das principais decisões técnicas, expressas como a arquitetura de software e o fornecimento de fundamentos para essas decisões, avaliando os interesses dos vários investidores, conduzindo os riscos técnicos e assegurando que as decisões sejam comunicadas, validadas e seguidas efetivamente'.

Em geral, um Arquiteto de Software começa sua carreira como um desenvolvedor de sistemas. Segundo (FOWLER, 2019a), o que faz com que esse progresso seja atingido é a habilidade de reconhecer elementos que sejam importantes do ponto de vista arquitetural e quais, eventualmente, podem resultar em sérios problemas se não forem controlados. Isso requer tempo, esforço e uma boa bagagem de experiência.

Bem como (MONSON-HAEFEL, 2009) cita, um arquiteto de software ocupa um lugar único na indústria de software. O autor pontua que um excelente arquiteto de software precisa dominar os dois lados da "moeda de arquitetura": negócio e tecnologia.

Logo, ele também alerta que esse domínio é um grande desafio.

Ao longo dos anos a comunidade de software percebe que a escolha de elementos tecnológicos apenas por serem aclamados é o mesmo que escolher tecnologia por ser tecnologia, ou seja, uma razão que não justifica a escolha. Dessa forma, os envolvidos com a arquitetura de software estão cada vez mais interessados em técnicas que os conduzam a não apenas o conhecimento sobre o domínio, mas a ter expertise na área.

1.3 Domain-Driven Design - DDD

Domain-Driven Design é uma abordagem para desenvolvimento de software que centraliza o desenvolvimento em um modelo de domínio que tem um entendimento de processos e regras de um domínio (FOWLER, 2020a). O tema veio a público em 2003 por meio de Evans em (EVANS, 2003) que em seu livro descreveu essa abordagem por meio de um catálogo de padrões. Desde então a comunidade de desenvolvedores usou as ideias de Evans e as expandiu, resultando em vários outros livros e cursos para treinamento na área. Essa abordagem é particularmente adequada para domínios complexos, onde várias lógicas desordenadas precisam ser organizadas, como por exemplo sistemas de larga escala (FOWLER, 2020a).

A ideia de que os sistemas de software precisam ser baseados num modelo de domínio bem desenvolvido não é tão recente. Muitos estudos em torno de banco de dados e orientação a objetos feitos pela comunidade de software entre as décadas de 80 e 90 a tiveram como parte chave (FOWLER, 2020a).

Uma das maiores contribuições de Evans foi criar um vocabulário para tratar essa abordagem, identificando conceitos chaves que foram além das várias notações de modelagem que dominaram a discussão na época (FOWLER, 2020a). O desenvolvimento de software para um domínio complexo necessita da construção de uma linguagem ubíqua que incorpora a terminologia do domínio dentro do software que será construído (FOWLER, 2020a). Isso significa que deve-se desenvolver uma linguagem comum e rigorosa entre desenvolvedores e usuários. Essa linguagem deve ser baseada no modelo de domínio usado no software e por isso deve ser rigorosa, já que requisitos de software não lidam bem com ambiguidade. Essa linguagem, e também o modelo, deve evoluir a medida que o entendimento do time sobre o domínio cresce (FOWLER, 2006a).

Modelo é um dos termos mais sobrecarregado de significados e ideias em software, mas possui propriedades em comum (NILSSON, 2006):

- É feito para um propósito definido;
- É um sistema de abstrações;

- É uma ferramenta cognitiva;
- Possui várias representações (linguagens, códigos, diagramas);
- Existem vários modelos em um sistema.

Todo software é relacionado a alguma atividade ou interesse do seu usuário. O conjunto de assuntos aos qual o usuário aplica um sistema computacional é o seu domínio. O modelo do domínio, mais conhecido em sua tradução para o inglês, Domain Model, é uma abstração rigorosamente organizada e seletiva do conhecimento de experts do domínio (FOWLER, 2020a).

Evans também introduz a noção de classificar objetos em Entidades, Objetos de valor e Objetos de serviço e identificar o conceito de Agregação, um design pattern importante quando se trata de uma aplicação complexa (FOWLER, 2020a). Outro conceito essencial incluído no DDD é a noção de Design Estratégico que é como organizar largos domínios numa network de Contexto Limitado (FOWLER, 2020a).

Contexto Limitado, ou Bounded Context é o padrão principal em Domain-Driven Design. DDD lida com largos modelos por dividi-los em diferentes Bounded Contexts e sendo explícito sobre seus relacionamentos (FOWLER, 2014a). Esse modelo é exemplificado na Figura 1.

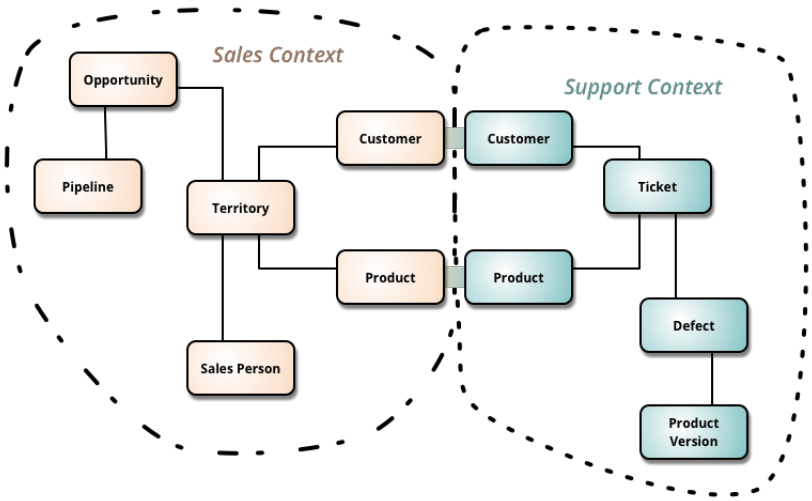

Figura 1 – O domínio de uma aplicação para vendas pode ser dividido em dois contextos (FOWLER, 2014a)

Atualmente, é aconselhado construir um modelo unificado para o negócio inteiro, mas o DDD reconhece que uma total unificação do modelo de domínio para um sistema de larga escala não é viável em termos de desenvolvimento e custo. Dessa forma, essa abordagem divide esse largo sistema em Bounded Contexts, e cada um deles tem um modelo unificado (FOWLER, 2014a).

1.4 Padrões Arquiteturais

Um dos elementos mais importantes de uma arquitetura de software é o padrão ou os padrões arquiteturais que a compõe. Esses padrões refletem estratégias que foram usadas, testadas e aprovadas para resolver problemas e que podem ser usados e combinados para resolver novos problemas. Muitos padrões estão disponíveis para comunidade de software, porém alguns são bem comuns no mercado. O modelo MVC está presente em muitos sistemas monolíticos, o padrão SOA pode considerado como o início da decomposição dos sistemas por serviços que então evoluiu para o padrão arquitetural que conhecemos por Microsserviços.

1.4.1 Model-View-Controller - MVC

Em português Modelo-Visão-Controle, o MVC é um conhecido padrão arquitetural que divide a aplicação em três partes lógicas: model, view e controller. Essas partes lógicas são definidas como camadas. A Figura 2 ilustra o fluxo de interações seguindo esse modelo. As requisições são processadas pela camada Controller, que por sua vez interage com a camada Model, que é responsável pela manipulação dos dados gravados na base de dados, para recuperar dados e/ou atualizá-los e assim apresentar os dados solicitados pela requisição HTTP que são tratados pela camada View antes de ser apresentado ao usuário. Já foi muito usado para interfaces gráficas de aplicações desktop, mas atualmente é usado para modelar apps mobile e aplicações web (SVIRCA, 2020).

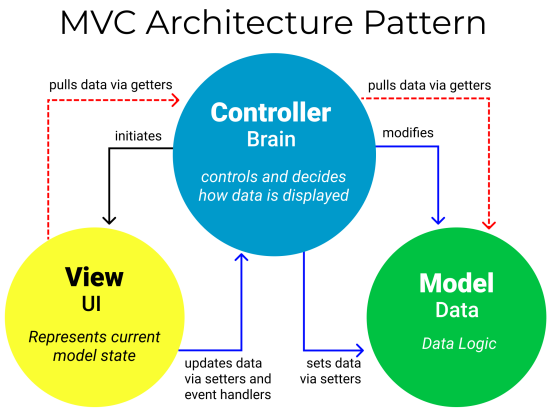

Figura 2 – Diagrama da estrutura do padrão MVC (FREECODECAMP, 2021)

A lógica por trás da descrição de cada camada do MVC pode parecer simples, porém a complexidade de implementação é alta, e essa é uma das suas principais desvantagens. Esse padrão foi pensado antes dos anos 80 e já conseguimos notar a busca por modularização para resolver problemas. Nas décadas seguintes as novas descobertas foram abrindo caminho para uma Arquitetura Baseada em Componentes, que é definida

por (VALIPOUR et al., 2009) como "um modelo de separação de funcionalidades, agrupando objetos que podem trabalhar juntos". Alguns frameworks usados atualmente foram criados tendo como base os conceitos por trás do MVC.

1.4.2 Service-Oriented-Architecture - SOA

O padrão SOA define uma maneira de reutilizar componentes de Software por meio de interface de serviços (IBM, 2021). Os serviços usam padrões de projeto e um padrão arquitetural, então podem ser rapidamente incorporados em novas aplicações. Isso simplifica o trabalho dos desenvolvedores por permitir reutilizar funcionalidades já implementadas (IBM, 2021).

Cada serviço em um SOA encorpora o código e dados necessários para executar uma função que satisfaz as regras de negócios (IBM, 2021). A interface do serviço provê baixo acoplamento por reduzir dependências entre eles. Esses serviços podem ser construídos do zero, mas geralmente são criados expondo funções de sistemas legados como interfaces de serviço (IBM, 2021).

Dessa forma, SOA representa um estágio importante na evolução no desenvolvimento e integração de aplicações nas últimas décadas (IBM, 2021). Antes do padrão SOA emergir no final dos anos 90, conectar uma aplicação a dados ou funcionalidades alocadas em outro sistema requeriam uma complexa integração de ponta a ponta (IBM, 2021). Uma integração que os desenvolvedores tinham que recriar, em parte ou completamente, para cada novo projeto. Expondo essas funções através de serviços SOA permitiu ao desenvolvedor simplesmente reutilizar recursos existentes e conectar por meio da arquitetura SOA Enterprise Service Bus (IBM, 2021).

Essa arquitetura ESB, é um padrão arquitetural onde um componente de software centralizado atua integrando aplicações (IBM, 2021). Isso é feito por transformações de modelo de dado, manipulando conectividade ou troca de mensagens, traçando rotas, convertendo protocolos de comunicação e gerenciando a composição de múltiplas requisições (IBM, 2021). É possível implementar um SOA sem um ESB, mas isso seria equivalente a apenas ter vários serviços. Cada aplicação precisaria se conectar diretamente a qualquer serviço que necessitasse e executar as transformações de dados necessárias para se adequar a cada interface do serviço (IBM, 2021).

Apesar do SOA e do Microservices compartilhem de várias palavras em comum, por exemplo "serviços"e "arquitetura", eles possuem poucas propriedades em comum de fato e operam em diferentes escopos (IBM, 2021).

1.4.3 Microservices - Microsserviços

O termo "Arquitetura Microsserviços"surgiu para descrever uma forma particular de projetar software como um conjunto de serviços que podem ter um pipeline para deploy independente (FOWLER, 2014c).

A Arquitetura Microsserviços é um estilo arquitetural que estrutura uma aplicação como uma coleção de serviços que são: Altamente manuteníveis e testáveis; Fracamente acoplados; Deploy independente entre eles; Organizados em torno de competências de negócios; e de responsabilidade de um pequeno time (RICHARDSON, 2021i). Essa arquitetura permite uma rápida, frequente e confiante entrega de aplicações complexas e de larga escala. Isso também permite que uma organização evolua seus conhecimentos sobre seu ramo tecnológico (RICHARDSON, 2021i).

Esses microsserviços são construídos entorno de competências de negócios e de seu processo de deploy completamente independente (FOWLER, 2014c). Dessa forma, é possível fazer um gerenciamento individual desses serviços, os quais podem ser escritos em linguagens de programação diferentes e usar diferentes tecnologias de armazenamento de dados (FOWLER, 2014c).

1.4.4 SOA vs Microservices

A diferença principal entre essas duas abordagens é o escopo. Basicamente, SOA tem um escopo a nível de organização, enquanto Microservices tem um escopo de aplicação (IBM, 2020), essa diferença está exemplificada na Figura 3. Muitos dos princípios essenciais de cada abordagem se torna incompatível quando essa diferença é negligenciada. Além disso, as duas podem potencialmente complementar uma a outra (IBM, 2020).

SOA é um estilo arquitetural de integração e um conceito que relaciona-se com toda empresa (IBM, 2021). Isso permite aplicações existentes serem expostas sobre interfaces com baixo acoplamento, cada uma correspondendo a sua função de negócios, que permite aplicações em uma parte de uma extensa empresa a reutilizar funcionalidades em outras aplicações (IBM, 2021).

Microservices é um estilo arquitetural de aplicação e um conceito a nível de aplicação. Permite que o interior de uma única aplicação possa ser decomposto em partes pequenas que podem ser mudadas independentemente, escaladas e administradas (IBM, 2021).

Numa empresa pode haver um sistema que utiliza globalmente a arquitetura SOA orquestrando vários microsserviços.

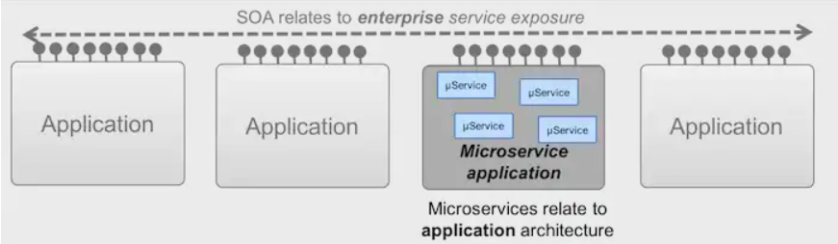

Figura 3 – Diferenças entre escopos das abordagens arquiteturais SOA e Microservices (IBM, 2020)

1.5 Padrões de Projeto

Popularmente conhecidos pela sua tradução em inglês, os Design Patterns seguem a mesma ideia dos padrões arquiteturais, porém com o objetivo de propor soluções mais específicas. Isso faz com que o número de padrões de projeto atualmente seja alto. Na literatura podemos encontrar dois grupos principais, GoF e GRASP.

Os padrões GRASP propõem uma abordagem sistemática de atribuição de responsabilidade entre as classes da aplicação (FACOM, 2021). Já os padrões *"Gang of Four"*, que foram criados por quatro autores do livro *"Design Patterns: Elements of Reusable Object-Oriented Software"* (GAMMA, 1995), que mesmo após mais de 20 anos de lançamento segue como um best seller no universo da computação, definem estratégias específicas usadas para lidar com problemas da programação orientada a objetos.

1.6 Sistemas de larga escala

O avanço tecnológico tem contribuído para que sistemas sejam cada vez mais abrangentes. Grandes empresas desenvolvem software com crescimento incremental, sem a possibilidade de estimar a escala limite desses sistemas, por isso a escalabilidade é uma necessidade de mercado. Além disso, a nossa economia e sociedade está se tornando cada vez mais dependente de sistemas complexos.

O fator chave que caracteriza sistemas de larga escala complexos é que esses sistemas são montados a partir de novos sistemas e já existentes, os quais são independentemente controlados e geridos. Práticos da área tem dado atenção a problemas envolvidos no desenvolvimento e manutenção desses sistemas (SOMMERVILLE et al., 2012).

A complexidade de sistemas deriva do número e tipo de relacionamentos entre os componentes do sistema e entre o sistema e o ambiente. Se há relativamente um baixo número de relacionamentos entre componentes do sistema e isso evolui de modo relativamente lento ao longo do tempo é possível desenvolver um modelo determinísticos do sistema e fazer predições de suas propriedades. No entanto, quando há várias relacionamentos dinâmicos entre elementos em um sistema então aí sim o sistema é complexo

(SOMMERVILLE et al., 2012).

Sistemas complexos não são determinísticos e suas características não podem ser preditas pela analise de seus sistemas constituintes. Essas características emergem quando o sistema inteiro é colocado em uso e mudam ao longo do tempo, dependendo de como o sistema é usado e em seu ambiente externo (SOMMERVILLE et al., 2012).

1.7 Non-Abstract Large System Design - NALSD

Baseado na experiência adquirida ao logo do desenvolvimento de sistemas da Google, foi desenvolvido um processo interativo de design de sistema chamado *Non-Abstract Large System Design* (NALSD) (Google, 2018). A necessidade de garantir confiabilidade aos sistemas de larga escala e ao mesmo tempo oferecer uma solução robusta e com baixo custo operacional motivou a proposta do NALSD (Google, 2018).

Quando se trata de sistemas de larga escala um fator crítico é a habilidade de avaliar, projetar e mensurar. O NALSD combina elementos de planejamento, isolar componentes e graceful degradation, que trata-se de tolerância a falhas que mantém o sistema ativo mesmo que algum componente falhe (BEYER et al., 2018). Essa combinação é crucial para atingir uma alta disponibilidade dos sistemas em ambiente de produção (BEYER et al., 2018).

O motivo de ser "Não abstrato"é oriundo da premissa de que um sistema deve sair do projeto e rodar em máquinas reais, networks reais e etc (BEYER et al., 2018). Os envolvidos no design dessas soluções devem sempre se certificar de que o que estão planejando pode ser traduzido para o mundo real.

1.7.1 Processo de Design

O processo de NALSD possui duas fazes (BEYER et al., 2018):

1. **Fase 1** O objetivo da fase 1 é criar uma solução que funcione. As seguintes perguntas devem ser respondidas:

   - *É possível?*

     como atender os requisitos, independente das restrições de hardware, conexão, entre outros.

   - *Podemos fazer melhor?*

     Pode-se pensar em tornar um sistema já existente ou um concorrente melhor, mais rápido, menor, mais eficiente e etc. Por exemplo, se um sistema resolve um problema levando O(N) de tempo, poderia ser projetado para fazer isso mais

rápido, algo como O(ln(N))? A inovação muitas vezes pode estar em tornar algo que não é novo em algo **melhor**.

2. **Fase 2** Essa próxima fase inicia-se a partir do design construído na fase 1. Com esse material em mãos partimos para mais três perguntas (BEYER et al., 2018):

   - *É viável?*

     O sistema pode envolver inúmeros requisitos como limite de memória RAM, CPU, largura de banda e por aí vai. Por isso, é importante pensar em algo que satisfaça requisitos que comprometem a viabilidade.

   - *É resiliente?*

     Como já foi citado, a tolerância a falhas é um requisito que não pode ser menosprezado em um projeto para um sistema de larga escala. Partes desse sistema pode estar espalhadas pelo mundo inteiro, em diversos provedores de núvem. Cada um desses componentes devem ser capazes de se recompor diante de uma adversidade. Além disso, mesmo que um sistema fique indisponível apenas por alguns instantes, isso pode comprometer a integridade e coerência dos dados. Outra situação seria a indisponibilidade do próprio datacenter por completo que também deve ser considerada como possibilidade.

   - *Podemos fazer melhor?*

     Já vimos que o design é possível e otimiza processos. Porém, é possível elevar o nível de melhorias? Essas perguntas levantam discussões que são feitas seguindo o fluxo de Fase 1 para Fase 2, depois retornando a Fase 1 para amadurecer novas ideias e logo seguindo para Fase 2 para refiná-las. Esse loop não tem um limite final arbitrário.

1.7.2 Requisitos inciais

Nessa fase é construída uma base que será evoluída para atingir os objetivos do projeto. Múltiplas reuniões são realizadas e técnicas de elicitação de requisitos são aplicadas. Nesse contexto é possível aplicar o que já foi apresentado sobre o DDD. É muito importante estar inteiramente ciente das regras de negócio e pensar em como uma abordagem NALSD poderia se encaixar ali.

Após a definição dos requisitos iniciais, os objetivos do projeto, que envolvem steakholders e usuários, podem ser divididos em: Service Level Agreement, Service Level Objectives, Service Level Indicators.

1.7.3 Service Level Agreement - SLA

Um SLA é um acordo entre um fornecedor e um cliente sobre métricas mensuráveis como por exemplo, tempo de resposta. Deixam claros entre as duas partes o escopo do que será desenvolvido, como deverá ser avaliado o que for entregue e os parâmetros que serão usados para isso (Atlassian, 2021).

Esses acordos são geralmente elaborados pelos envolvidos com o negócio e assuntos jurídicos entre as empresas, e representam o que deve ser feito e as consequências de falhas ou inconformidades, que podem resultar em penalidades financeiras, quebras de contrato e entre outras (Atlassian, 2021).

Para que as expectativas do cliente estejam alinhadas com o que pode ser realmente entregue, é necessário que os responsáveis pela parte técnica também participem na criação dos SLAs e colaborem com o desenvolvimento legal e de negócios para elaborar SLAs que atentam cenários do mundo real, não abstratos (Atlassian, 2021).

1.7.4 Service Level Objectives - SLO

Um SLO é um acordo dentro de um SLA sobre uma métrica especifica, como a já citada, tempo de resposta (Atlassian, 2021). Então, se o SLA é um acordo formal entre o fornecedor e o cliente, SLOs são os acordos individuais que o fornecedor está fazendo para esse cliente. São um conjunto de expectativas do cliente e diz às equipes de TI quais objetivos precisam atingir e mensurar de acordo com o acompanhamento de metas internas (Atlassian, 2021).

Quando SLAs são são relevantes apenas no caso de clientes responsáveis pelo pagamento, SLOs podem ser de grande ajuda para lidar com clientes internos e externos, mesmo que entre os envolvidos não inclua pagamentos (Atlassian, 2021).

1.7.5 Service Level Indicador - SLI

Um SLI mensura conformidade em um SLO. Então, por exemplo, se um SLA especifica que o sistema vai estar disponível 99, 95% do tempo, o SLI vai ajudar a cumprir o acordo (Atlassian, 2021). Como um SLO, o desafio de SLIs é os manter simples, escolhendo métricas corretar para acompanhar, e não complicar sem necessidade o trabalho da equipe de TI por acompanhar muitas métricas que na verdade não importa para os clientes (Atlassian, 2021).

Esses três services level estão resumidos na Figura 4.

Figura 4 – Acordos e metas estabelecidos no início do projeto

1.7.6 Uma máquina

O simples ponto de partida é considerar rodar a aplicação inteira em um único computador. Pensar em como seria todo o sistema em uma única instância (BEYER et al., 2018). Após percorrer paras as outras interações obviamente o resultado será que o sistema não cabe em apenas uma máquina e o one-machine design é inviável, porém esse passo não é uma perca de tempo e não deve ser evitado, pois com isso é possível descobrir informações valiosas sobre como raciocinar sobre as restrições do sistema de acordo com os requisitos iniciais (BEYER et al., 2018). Após isso, é necessário evoluir o design do sistema para mais de uma única instância.

1.7.7 Cálculos

Para determinar se uma solução é viável ou não é necessário verificar se é numericamente suficiente (BEYER et al., 2018). Um exemplo disso seria, abrir uma loja de livros em uma cidade de mais de 1 milhão de habitantes. O que fazer com a possibilidade de 1 milhão de possíveis leitores? Essa é a pergunta mais básica para iniciar o projeto para uma solução como essa.

No mundo de software existem ainda mais variáveis, tais como tempo, processamento, memória, restrições de hardware, atrasos e entre outras coisas. Então, a missão dos projetistas na fase de cálculos é usar dados reais negociais e técnicos alinhados para gerar uma estimativa que não seja irreal (BEYER et al., 2018).

1.7.8 Avaliações

Ignorando os cálculos por um momento e imaginando todo sistema em um design one-machine é importante se perguntar: "O que acontece quando esse componente falha?", pois assim é possível identificar uma longa lista de pontos únicos de falha, como por

exemplo CPU, memória, armazenamento, energia, network, cooler (BEYER et al., 2018). A pergunta final dessa linha de raciocínio é: É viável atender os SLOs se um desses componentes falhar? É vital estar seguro de que não haverá impactos significantes aos usuários (BEYER et al., 2018).

Após todos cálculos realizados, é feita uma avaliação da solução como um todo. É nessa etapa que o sistema de modo distribuído vai sendo moldado. Com cálculos e requisitos em mãos já é possível pensar em possíveis soluções (BEYER et al., 2018). Um exemplo, com a estimativa do volumes de dados é possível determinar quantos *datacenters*, como serão distribuídos, como os dados serão processados e etc. Nesse ponto, as descobertas feitas são valiosas informação que torna possível raciocinar sobre as restrições do sistema e seus requisitos iniciais (BEYER et al., 2018).

1.8 Literatura Cinzenta

Non-Abstract Large System Design e várias outras discussões sobre arquitetura de software de larga escala são pouco encontradas em materiais acadêmicos. Por isso, para analisar textos e artefatos que serão úteis e obter entendimento maior sobre esse tema, é possível examinar um tipo de material chamado Grey Literature (GL) (WEN et al., 2020). Isso inclui, relatórios técnicos, artigos, revistas, blogs e outros recursos que são desenvolvidos e mantidos por práticos do mercado de produção de software.

O desenvolvimentos de software amplamente escaláveis é tema de pesquisa tanto para a industria como para a academia. Porém, a maior e mais atualizada parte das informações são produzidas de forma não controlada por editores comerciais.

A natureza informal da GL pode ser uma de suas vantagens. É uma fonte líder para identificar tópicos e gaps ainda não explorados pela literatura acadêmica. Isso possibilita investigações mais atualizadas e informações emergentes, como NASLD, já que materiais acadêmicos formais sofrem com longos atrasos para a publicação devido ao processo de revisão por pares, conhecido como peer-review (WEN et al., 2020).

Apesar de haver boas recomendações para a inclusão de materiais GL em revisões de literatura no âmbito da Engenharia de Software, essa ainda não é uma metodologia ou guideline precisa para prescrever passos e restrições bem definidos para conduzir Grey Literature Reviews (GLR). Há muito conhecimento em fontes não acadêmicos formais, mas também há um vasto número de informações incorretas, inconsistentes ou inconclusivas. Os materiais podem não ser muito estruturados e com vocabulário não uniforme, já que não necessitam seguir rigorosos padrões de normas usadas em materiais acadêmicos. Por isso, é necessário um esforço extra para garantir a qualidade das informações e fontes incluídas em uma pesquisa.

2 Metodologia de Pesquisa

O objetivo geral desse trabalho é identificar quais são as estratégias que os práticos da engenharia de software estão adotando no design arquitetural de sistemas de larga escala. Nesse capítulo, são apresentados o método utilizado e os resultados preliminares.

2.1 Metodologia

Para execução dessa pesquisa, é adotada a metodologia de Grey Literature Review, que trata-se de uma Revisão Sistemática de Literatura utilizando materiais da Literatura Cinzenta. A principal referência usada é (WEN et al., 2020), que apresenta o passo a passo para o desenvolvimento de uma GLR baseado em referências e experiências relacionadas a esse tipo de revisão.

Uma Revisão Sistemática de Literatura (RSL) é uma metodologia que segue procedimentos rigorosos para sintetizar e avaliar as evidências sobre algum assunto. Combinar os métodos utilizados em uma RSL com uma GLR contribui para a descoberta de práticas importantes, utilizadas no mercado atual e que ainda não foram mapeadas por estudos convencionais em engenharia de software.

Assim como em uma RSL, o plano metodológico de uma GLR provê diretrizes, estrutura e transparência para métodos de pesquisa. Para reduzir viés de pesquisa, foram mapeados um data source, termos de pesquisa, critérios de seleção e limites de busca. O fluxo de pesquisa usado nessa GLR está representado na Figura 5. Para extrair conceitos dos dados não estruturados dos textos, foram utilizadas técnicas de coding.

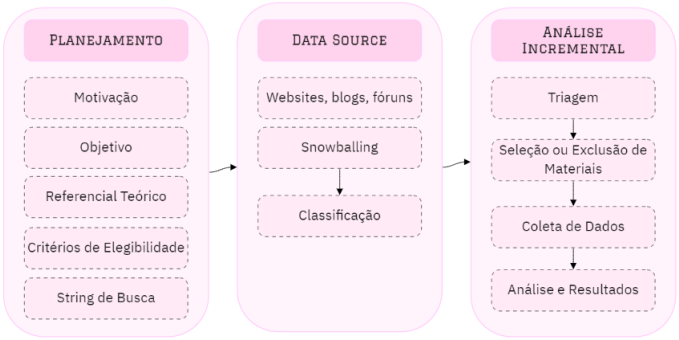

Figura 5 – Fluxo adotado para execução da GLR

2.1.1 Protocolo da GLR

A motivação para a realização dessa pesquisa é a importância da Arquitetura de Software em Sistemas de larga escala. O objetivo é identificar estratégias modernas que os práticos estão adotando no desenvolvimento desses software. A análise de materiais pode revelar boas estratégias usadas do design desse tipo de solução e melhorar o entendimento sobre a estrutura de grandes sistemas. A pergunta de pesquisa que guia essa revisão é

RQ: "Quais são as estratégias de design arquitetural adotadas em sistemas modernos de larga escala?".

Realizar uma revisão de literatura cinzenta possui alguns riscos, já que esses materiais não são revisados. Além disso, as evidências presentes nesses tipo de conteúdo podem ser baseadas em experiência e opinião pessoal, com risco de coleta de informações falsas e pontos de vista de pessoas não envolvidas de fato com o assunto pesquisado. Tendo em vista os benefícios desse tipo de estudo, esse trabalho utiliza essa metodologia definindo critérios de exclusão e inclusão para superar esse problema.

2.1.2 Fontes de dados - Critérios de Inclusão e Exclusão

A primeira etapa de uma GLR é a definição de quais fontes de dados serão utilizadas. Nessa revisão o Data Source, que é o repositório de materiais selecionados, é incrementado com materiais que satisfazem os seguintes critérios:

2.1.2.1 **Critérios de Inclusão**

1. **Intervalo de Publicação entre 2010 e 2021** - Tendo em vista o objetivo de elencar estratégias de design arquitetural modernas, materiais de mais de 11 anos de publicação serão desconsiderados. As referências de estudos realizados posteriormente a esse período definido mas que embasam discussões e práticas modernas estão incluídas no referencial teórico e não na revisão literária.

2. **Qualidade da Fontes de dados** - Serão analisadas publicações online de instituições de grande porte, como textos e documentos de grandes empresas que lidam com sistemas de larga escala. Também serão analisados blogs de arquitetos de software com grande experiência e consolidação no mercado e fóruns de discussão sobre arquitetura e design de software. Artigos de outros profissionais, que não sejam identificados como arquitetos de software, podem ser selecionados se o autor for bem recomendado e seu material for bem avaliado.

3. **Idioma** - O principal idioma a ser considerado na seleção de materiais será o inglês devido ao objetivo de identificar o que está sendo utilizado em dimensão global. Contudo, materiais que possuem boa qualidade e sejam em português não serão

descartados já que a comunidade brasileira de software também produz conteúdo sobre essa temática.

4. **Conteúdo** - Materiais que possuem experiências reais, apresentam estatísticas, exemplos práticos e ilustrações como diagramas, figuras, tabelas e outros recursos visuais que auxiliam numa maior compreensão do que está sendo abordado são os de maior relevância para esse trabalho.

2.1.2.2 Critérios de Exclusão

1. Não contribui de modo significativo para o objetivo desse estudo;
2. Respostas e comentários que não são confiáveis em fóruns;
3. Material apenas relacionado à implementação de software;
4. Material já analisado posteriormente;
5. Material não disponível gratuitamente;
6. Material escrito em algum idioma que não seja inglês ou português;
7. Fora do intervalo de tempo definido para inclusão.

2.1.3 Critérios de Inclusão e Exclusão para Mecanismos de Busca

Foram aplicados critérios de seleção e exclusão para cada mecanismo de busca considerado.

2.1.3.1 Critérios de Inclusão

1. Autores e usuários das fontes façam parte de uma grande comunidade
2. Sejam blogs ou fóruns de empresas tecnológicas ou de grandes autores
3. Posts ou respostas que tenham grande aprovação e abrangência, verificadas por meio do número de visualizações, curtidas, avaliações ou referências
4. Autores devem ser Arquitetos de Software ou Desenvolvedores experientes
5. Atender a pelo menos 3 critérios de inclusão

2.1.3.2 Critérios de Exclusão

1. Não possui materiais relevantes para a pesquisa
2. Difícil Navegabilidade
3. Não satisfaz os critérios de inclusão

2.1.4 Grounded Theory

Graunded Theory (GT) é uma metodologia de pesquisa qualitativa para o desenvolvimento de teorias a partir de dados não estruturados, por meio de indução. Existem algumas variantes desse tipo de teoria, e a variante adotada neste trabalho é a proposta sugerida pelos autores (STOL; RALPH; FITZGERALD, 2016), a chamada Straussian GT.

Ainda que haja diferenças entre as variantes de uma GT, elas compartilham de vários componentes fundamentais. O resultado desse trabalho não gera uma teoria, mas o uso das boas práticas de uma GT contribuiu com o objetivo de mapear, de forma sistemática, o comportamento de desenvolvedores e organizações no design arquitetural de software de larga escala. Alguns desses componentes, que foram aplicados nesse trabalho, são:

- **Limitar a exposição à Literatura**

  A maior razão para limitar o estudo da literatura é impedir o pesquisador de testar teorias existentes ou pensar em termos de conceitos estabelecidos. Ainda que o objetivo geral não é relacionado ao desenvolvimento de uma teoria, para a pesquisa realizada nesse trabalho foram estabelecidos critérios para seleção de fontes de dados e então foram feitas buscas utilizando o motor de busca interno de cada fonte.

- **Tratar tudo como dado**

  Para o autor Glaser, tudo ser dado significa que dados qualitativos, dados quantitativos, dados semi-estruturados, imagens, diagramas, vídeos e até teorias e literatura existentes são dados significativos. Os dados dessa pesquisa são derivados desse tipo de material, como especificado na Seção 1.8.

- **Análise de dados imediata e contínua**

  Em uma GT, o pesquisador começa analisando os dados imediatamente e não termina toda a coleta dos dados antes de começar a análise. A coleta e a análise de dados são feitas simultaneamente. Neste trabalho isso foi feito no processo de Análise Incremental do Fluxo de Execução da GLR apresentado na Figura 5.

- **Sensibilidade teórica**

  A habilidade do pesquisador de estabelecer relacionamentos e conexões entre conceitos é peça chave no desenvolvimento de uma GT. Nesse processo é importante ter criatividade. Esse foi um dos desafios dessa pesquisa.

- ***Coding***

  No desenvolvimento de uma GT, o pesquisador usa de lógica indutiva para desenvolver códigos analíticos e categorias teóricas a partir dos dados. Os códigos são identificadores de cada material e relacionados com uma ou mais categorias. O processo de *Coding* e todos os códigos desenvolvidos nessa pesquisa estão descritos na Seção 2.1.5. Exemplo: O trecho, "Based on Google's experience developing systems, we consider reliability to be the most critical feature of any production system. We find that deferring reliability issues during design is akin to accepting fewer features at higher costs."retirado de (GOOGLE, 2020) é referente a categoria "Confiabilidade do Sistema", e é mapeado com o código SRE-01, que relaciona o material à fonte de dados em que foi coletado. Dessa forma, o pesquisador fica ciente de que esse material trata sobre "Confiabilidade do Sistema"e que foi selecionado a partir da referência codificada. Essa rastreabilidade facilita de modo considerável a análise e coleta de dados.

- ***Memoing***

  Ao longo do desenvolvimento de uma GT, o pesquisador escreve anotações ou, no inglês, *memos*, como notas, diagramas ou esboços, para elaborar categoria a medida que elas emergem, descrever propriedades e relacionamentos preliminares e identificar gaps. Esses *memos* tem um papel importante no desenvolvimento da pesquisa. O autor Glaser cita que se o pesquisador não inclui esse processo na sua pesquisa, ele na verdade não está realizando uma Grounded Theory. Para essa pesquisa foram utilizados bloco de notas, planilhas e folhas de papel para registro de *memos* durante todo o processo de pesquisa.

- **Constante comparação**

  O pesquisador constantemente compara dados, anotações, códigos e categorias durante o estudo em uma GT. A definição de categorias e a interpretação dos dados segue evoluindo até a saturação teórica.

- **Saturação teórica**

  O pesquisador termina a coleta e análise dos dados quando a saturação teórica é alcançada. Essa saturação se refere ao ponto em que novos dados convergem a interpretações e inferências já feitas. Dessa forma, os dados não geram novas descobertas ou afetem os resultados obtidos. Esse trabalho não chega em saturação teórica.

2.1.5 Coding

O processo de *Coding* foi realizado através de *Open Coding* que é a geração de categorias comuns aos materiais analisados, *Axial Coding* que é identificar relacionamento entre as categorias e por fim *Selective Coding* que é definir uma categoria central que se relaciona com a maioria das demais categorias.

2.1.5.1 **Open Coding**

As categorias foram definidas a medida que o processo especificado pelo protocolo da GLR foi sendo executado. Elas estão inseridas dentro do contexto delimitado pela pergunta de pesquisa e objetivo do estudo. Após definidas as categorias, o próximo passo foi buscar materiais referentes a cada uma delas.

O processo de seleção de materiais foi incremental, tal qual especificado na fase de análise dentro do fluxo adotado para a execução da GLR, demonstrado na Figura 5. A medida que foram identificadas referências aos termos buscados, os materiais receberam um código padronizado da seguinte forma: Sigla para a Fonte de Dados, traço (-) e número ordenado de forma crescente para cada material. Exemplo, FD-01, sendo FD uma sigla para "Fonte de Dados"e 01 por ser o primeiro material selecionado.

Após a seleção inicial, cada material listado por categoria foi revisitado e reanalisado seguindo os critérios de seleção e exclusão. Dessa forma, o número de materiais que de fato apoiam o objetivo de cada categoria foi refinado. Durante esse processo, os códigos foram estruturados em uma planilha do Google Sheets (GOOGLE, 2021), onde foram feitas anotações e controle de aprovação de materiais e justificativas de exclusão. Também foram considerados, por meio de anotações, alguns trechos de materiais fortemente relevantes para a pesquisa e sinalização de materiais que mereciam mais atenção.

Tendo dados coletados e refinados, a fase de *Axial Coding* foi iniciada, partindo da elaboração de um 2.1.5.2 Guia de Relacionamento Condicional e a consolidação dos resultados por meio de uma 2.1.5.3 Matriz de Coding Reflexiva.

2.1.5.2 **Guia de Relacionamento Condicional**

Perguntas tais como "O que?", "Quando?", "Onde?", "Por que?", "Como?"e "Quais consequências?", sobre quais condições o fenômeno analisado ocorre, ajuda a descobrir ideias importantes. Responder a essa perguntas reúne o conjunto solto de categorias, que foram definidas durante a fase de *Open Coding*, e as classifica em um padrão coerente. Essa estrutura é utilizada no desenvolvimento de um Guia de Relacionamento Condicional durante a fase de *Axial Coding*.

O Guia de Relacionamento Condicional é uma tabela com o nome de cada categoria na coluna mais a esquerda e as demais categorias com as respostas das 6 perguntas

anteriormente citadas. O formato da tabela é desenhado para responder cada questão relacional sobre a categoria nomeada na coluna esquerda. Uma estratégia que pode ajudar a responder essas perguntas é refletir de modo estruturado da seguinte forma:

- Qual é [categoria]?
- Quando [categoria] ocorre?
- Onde [categoria] ocorre?
- Por que [categoria] ocorre?
- Como [categoria] ocorre?
- Quais consequências referentes [categoria] ocorrerem?

É notável que "Quando?", "Onde?"e "Porque?"são perguntas que identificam os limites e condicionais do contexto. A quinta pergunta, "Como?", identifica ações e interações entre as categorias. A última pergunta pode ser um desafio, porque isso requer uma reflexão do significado geral da categoria por completo, e depois transformá-lo em um conceito claro e conciso. Muitas vezes algumas categorias são listadas múltiplas vezes ao responder a pergunta sobre as Consequências. Esse processo contribui inclusive com o surgimento de novas ideias, embasadas pelos conceitos definidos nas respostas para essas 6 perguntas a cada categoria.

2.1.5.3 **Matriz de Coding Reflexiva**

Entender o relacionamento entre as categorias emergentes não é intuitivo. Para isso, (MCCASLIN, 1993) sugere o desenvolvimento de um Matriz de Coding Reflexiva neste ponto da análise. Em uma GT, um método mais especifico para entender o relacionamento entre as categorias pode ajudar o pesquisador. Tendo desenvolvido relacionamentos entre as categorias de estudo usando o 2.1.5.2 Guia de Relacionamento Condicional, o próximo passo, durante a fase *Axial Coding*, é aproximar esses relacionamento para criar padrões que sejam um suporte para reconhecer um fenômeno central. A Matriz de Coding Reflexiva é de ajuda para o desenvolvimento tanto de um fenômeno central quanto de uma story line, que exibe suas dimensões e condições. As consequências que emergem do Guia são as primeiras consideradas para desenvolver um fenômeno central, usando a Matriz de Codign Reflexiva. As categorias presentes no Guia que não possuem consequências são geralmente dimensões em uma Matriz de Coding Reflexiva.

A Subjetividade é uma razão para aplicar uma cuidadosa verificação dos relacionamentos emergentes com os dados que forma coletados de várias formas. Isso também é

uma razão para a altamente recomendada prática de *Memoing* (STOL; RALPH; FITZGERALD, 2016) em uma análise axial dos relacionamentos, indicados pelo processo de categorização localizado no 2.1.5.2 Guia de Relacionamento Condicional. Memos durante essa fase de analise são indispensáveis, tanto durante a análise quanto depois na formalização dos resultados.

O 2.1.5.2 Guia de Relacionamento Condicional identifica os relacionamento e interações entre as categorias, e também descreve como as consequências de cada categoria são entendidas. A Matriz de Coding Reflexiva é uma ferramenta para criar uma story line dos muitos padrões descobertos no 2.1.5.2 Guia.

O objetivo primário para a construção de uma Matriz de Coding Reflexiva, como uma hierarquia relacional, é desenvolver e contextualizar a categoria principal, que é o fenômeno central sobre cada uma das outras maiores e menores categorias relativas. Uma vez que a categoria principal é identificada, todas as outras categorias se tornam subcategorias. As subcategorias passam a ser descrições da categoria principal, como as propriedades, processos, dimensões, contextos, e modos para entender as consequências do fenômeno central de interesse. O método para identificar o que descreve a Matriz de Coding Reflexiva começa nos relacionamentos estabelecidos no 2.1.5.2 Guia.

Outro objetivo para o desenho dessa Matriz é para que ela sirva como uma "foto"do fenômeno central, o definindo e descrevendo de um modo suficiente, para que os dados coletados durante o estudo sejam apresentados como uma narrativa, ou história, que explica a teoria substantiva do fenômeno central. A categoria principal é uma forma de nomear o fenômeno central.

Como os dois objetivos citados são interessantes para a análise de uma possível tendência adotada por práticos, durante o processo de design arquitetural de software de larga escala, como um fenômeno, foi desenvolvida uma Matriz de Coding Reflexiva de acordo com as diretivas citadas. A Matriz está representada na Tabela 6.

2.1.5.4 **Selective Coding**

A 2.1.5.3 Matriz de Coding Reflexiva auxiliou na determinação da principal categoria, que representa o fenômeno central identificado por meio desse estudo. Por fim, na fase de Selective Coding, a saturação teórica é parcialmente atingida. Todas as informações coletadas convergiram para os mesmos resultados e então tornou-se possível mapear os conceitos consolidados, construindo assim uma Story line que narra os relacionamentos entre eles. Essa Story line é apresentada por meio de um Mapa Conceitual apresentado na Figura 9.

3 Resultados

3.1 String de Busca

Na literatura cinzenta há uma variedade de terminologias usadas por autores e instituições, já que palavras-chave populares em estudos de engenharia de software ainda não são comumente usadas por práticos da área. Por isso, é essencial definir um conjunto de termos de busca, baseados na pergunta de pesquisa e nos critérios de elegibilidade, que seja flexível, para lidar com esse problema usando termos mais informais e heterogêneos. O mecanismo de busca usado primariamente nesse trabalho foi o Google. Os termos usados para encontrar conteúdos alvo para incluir no Data Source de pesquisa estão presentes na Tabela 1. A string de busca principal é *"Software Architecture Design Large Scale"* e visando encontrar um número mais amplo de materiais foram criadas mais quatro variações estratégicas.

Estratégia	Termos				
1	Software	Architecture	Design		Large Scale OR String Vazia
2	Software	Architecture	Domain OR Non-Abstract Large System Design	Design	Large Scale OR String Vazia
3	Software	Architecture	Design	Process OR Methodology OR Guide OR Rules OR Issues OR Challenges OR Community OR Patterns OR Best Practices OR Examples OR Use Cases OR Problems OR Improvements OR Strategies OR Modern	Large Scale OR String Vazia
4	Arquitetura	Software	Design	Processos OR Metodologias OR Guia OR Regras OR Comunidade OR Desafios OR Padrões OR Melhores Práticas OR Exemplos OR Casos de Uso OR Problemas OR Melhorias OR Estratégias OR Moderna	Larga Escala OR String Vazia

Tabela 1 – Strings de Busca

Os termos considerados nas **Estratégias 3** e **4** são os termos-chave das categorias emergentes da fase de Open Coding. Durante o processo incremental de análise dos dados, obtidos a partir dos materiais selecionados, duas categorias primariamente definidas, as quais são *Melhorias no processo de Design* e *Metodologia de processo de Design*, foram desconsideradas, pois os materiais relacionados à essas categorias não ofereciam boa base para consolidar conceitos dentro do 2.1.5.2 Guia de relacionamento condicional. As

categorias remanescentes foram:

1. DevOps
2. Processos de negócio e desenvolvimento
3. Regras de Negócio e de Domínio
4. Problemas na implementação do Sistema
5. Desafios no desenvolvimento de Sistemas de Larga Escala
6. Comunidade de práticos da Engenharia de Software
7. Padrões de Projeto e Arquitetura
8. Melhores Práticas de Design e Arquitetura
9. Exemplos de implementação, design e metodologia
10. Casos de Uso de Sistemas de Larga Escala
11. Problemas no processo de Design
12. Estratégias de Design e Arquitetura
13. Sistemas modernos
14. Diretrizes estratégicas de Design e Arquitetura
15. Confiabilidade do sistema

Para materiais em Inglês, as categorias tiveram seus termos traduzidos assim como está especificado na **Estratégia 3**, que contém o termo-chave de cada categoria pesquisada.

Os códigos relacionam cada material a sua respectiva fonte de dados. Para esse segundo momento da pesquisa, foi adotada uma nova estratégia para mecanismos de busca. Os materiais foram selecionados dentre os resultados de pesquisas realizadas em mecanismos de pesquisas internos em alguns dos principais blogs e fóruns de discussão sobre Engenharia de Software.

3.2 Gray Literature Review

A primeira etapa do trabalho foi fazer uma revisão bibliográfica usando a técnica de *snowbawling*. Dessa forma, houve uma familiarização com o vocabulário, identificação das principais fontes de dados e principais autores da área.

Executando as duas primeiras etapas do protocolo, foi possível desenvolver um referencial teórico com as informações necessárias para pautar as análises feitas sobre os dados coletados e inseridos no Data Source. A primeira execução, usando a **Estratégia 1** da String de Busca, gerou os resultados, representados no gráfico na Figura 6, considerando a quantidade de materiais somados das duas variantes dessa estratégia em cada ano de publicação, dentro do intervalo aceito, de acordo com os critérios de inclusão e exclusão.

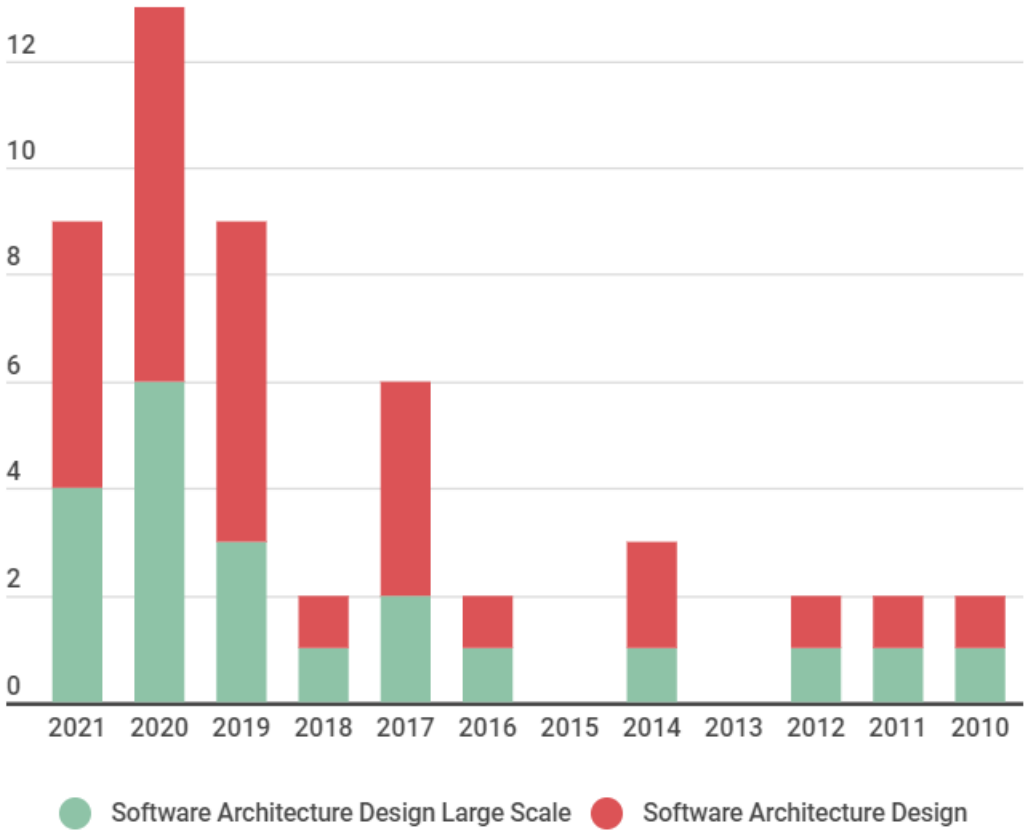

Figura 6 – Quantidade de materiais encontrados x Ano de publicação da Estratégia 1 da String de Busca

Os materiais coletados são referentes a busca pelos termos da Estratégia 1 no Google até a página 5, filtrando pelo intervalo de data de publicação de 2010 até 2021. Foi usada a ferramenta Pocket para salvar os links e dividi-los por tags. Primeiramente, foi salvo todos os links resultantes da busca sem muito critério, evitando apenas links de

anúncios, dos termos *"Software Architecture Design"*. Restaram 29 links após filtrar de acordo com os critérios de exclusão artigos sem data e links de lojas ou cursos de TI.

Adiante, foi feita a busca por *"Software Architecture Design Large Scale"* o que resultou em 31 links até a página 5. Após essa busca, restaram 21 links ao filtrar de acordo com os critérios de exclusão para artigos sem data e links de cursos de TI. Ao total foram encontrados 50 materiais executando o primeiro refinamento da busca, utilizando a **Estratégia 1**.

A primeira execução usando a **Estratégia 2**, incluindo a primeira variante ao termo *"Design"*, gerou os seguintes resultados representados no gráfico na Figura 7, considerando a quantidade de materiais de acordo com a data de publicação. Da mesma forma que o gráfico anterior, o gráfico seguinte representa a soma das duas variantes de pesquisa somadas em cada barra, e é ordenado de acordo com o intervalo de publicação aceitos nos critérios de inclusão e exclusão, definidos no protocolo.

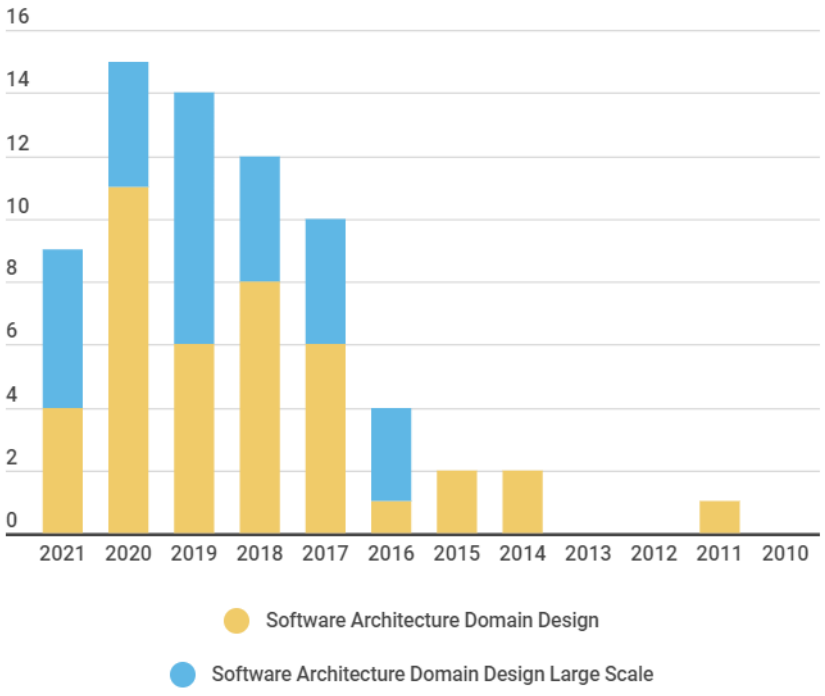

Figura 7 – Quantidade de materiais encontrados x Ano de publicação da primeira variante da Estratégia 2 da String de Busca

Os materiais coletados são referentes a busca pelos termos das primeiras variantes da **Estratégia 2** que são *"Software Architecture Domain Design"* e *"Software Architecture Domain Design Large Scale"*. Mais uma vez foi usada a ferramenta Pocket para salvar os links retornados pela busca dos termos no Google, filtrando pelo intervalo de data de publicação de 2010 até 2021 até a página 5. A busca pouco criteriosa por *"Software Architecture Domain Design"* retornou 58 links. Após ser feita uma filtragem excluindo links

de cursos de TI, artigos acadêmicos e perfis profissionais, restaram 41 links. A busca por *"Software Architecture Domain Design Large Scale"* resultou em 44 links até a página 5 e após a filtragem baseada nos critérios de exclusão de cursos e perfis profissionais, restaram 28 links. Ao todo foram encontrados 69 links considerando o primeiro refinamento.

Prosseguindo para a busca da segunda variante ao termo *"Design"*, temos o resultado representado pelo gráfico na Figura 8. Nesse caso, também temos a soma da quantidade de materiais das duas subvariantes em cada barra do gráfico ordenado pela linha temporal do período permitido pelos critérios de inclusão e exclusão presentes no protocolo.

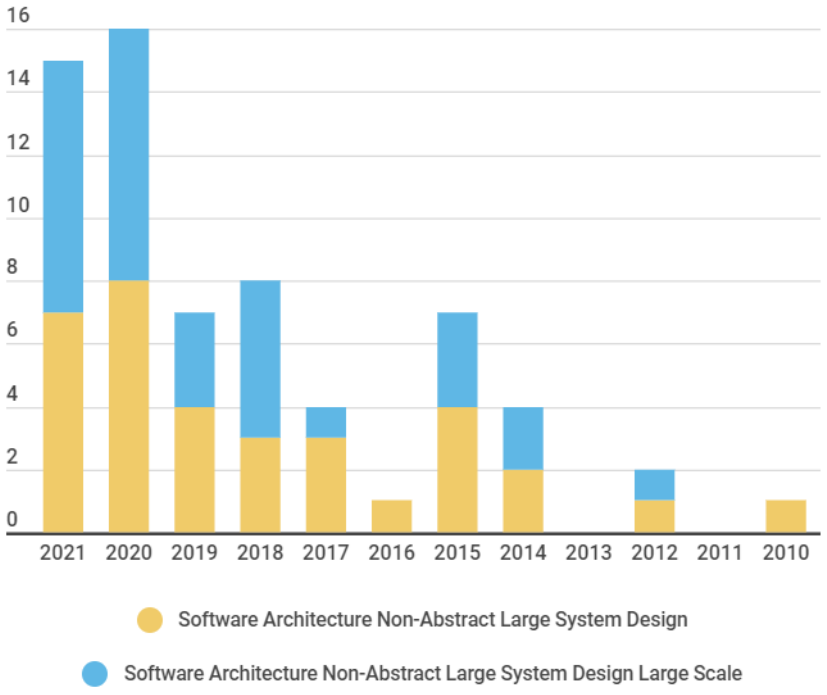

Figura 8 – Quantidade de materiais encontrados x Ano de publicação da segunda variante da Estratégia 2 da String de Busca

Os materiais encontrados para *"Software Architecture Non-Abstract Large System Design"* na busca no Google filtrando pelo intervalo de data de publicação de 2010 até 2021 até a página 5 somaram-se 53 links coletados com pouco critério. Após a filtragem excluindo materiais referentes a cursos, artigos acadêmicos e materiais sem data, de acordo com os critérios de exclusão e inclusão definidos no protocolo, restaram 34 links. Em seguida, os materiais encontrados para *"Software Architecture Non-Abstract Large System Design Large Scale"* até a página 5 do Google, e mais uma vez numa busca pouco criteriosa, somaram-se 66 links. Após a filtragem excluindo artigos acadêmicos, cursos e materiais sem data, restaram 31 links.

As estatísticas levantadas pelos primeiros refinamentos pouco revelam sobre a qualidade dos materiais e também não contemplam a duplicidade dos mesmos entre as estratégias adotadas para a String de Busca. Porém, é possível notar que o Design e Arquitetura de software se tornaram assuntos populares ao longo dos últimos anos A comunidade tem se movimentado para discutir e propor soluções vindas de novas descobertas no mercado. A colaboração global entre os desenvolvedores também é algo notório, incluindo disponibilização de ferramentas gratuitas, publicação de artigos pouco formais em blogs e sites destinados ao compartilhamento de conhecimento e trocas de dúvidas e respostas em grandes fóruns como o Stack Overflow e o Quora.

Quanto as estratégias que vem sendo adotadas no plano arquitetural, é possível observar que a solução geral culmina de algum modo para o uso dos microsserviços, onde entra a parte técnica da Arquitetura de Software. Alguns autores como (IBM, 2020) sugerem também o uso complementar dos microsserviços em uma arquitetura SOA.

Com relação ao Design, a tendência observada desde o livro de (EVANS, 2003) é dirigir o design por domínios. Estudos recentes estão sendo feitos como os de (VERNON, 2013) e (FOWLER, 2020a) a respeito, e a grande tendência atual é evoluir o trabalho do Evans para se adaptar cada vez mais a forma como é feita a agregação entre partes do sistemas baseados em seu contexto e também sobre a linguagem adotada por toda equipe de tal forma que seja possível dialogar com pessoas que não tem conhecimento técnico de tal maneira que ela se mantenha a par do andamento do projeto. Isso envolve não apenas conhecimento técnico da equipe, mas a forma de pensar e suas ações.

Outra característica observada é que o design de software atual hoje não é pautado apenas em padrões arquiteturais e ferramentas, envolve também a cultura organizacional. Por isso, não conhecemos, por exemplo, uma das abordagens mais importante ao lidar com sistemas de larga escala, o DevOps, como técnica ou método e sim como uma cultura (WILSENACH, 2015).

Apesar de estarem sendo feitos grandes esforços em estudar o design dirigido por domínios, a empresa Google tem ministrado workshops para apresentar o Non-Abstract Large System Design. O material disponível sobre o tema é bem limitado, porém é dito pela própria Google como uma solução para o design feito atualmente utilizando padrões arquiteturais (GOOGLE, 2020). A hipótese até o momento é que os arquitetos da empresa enfrentam desafios, que provavelmente sejam similares ou os mesmos que seus colegas de profissão no mundo todo, porém decidiram criar as suas próprias regras.

Como foi observado, há uma entrevista ministrada pela Google utilizando essa metodologia, apresentada na seção de Referencial Teórico (LüDTKE, 2019). Essa abordagem parece eficiente, porém os resultados são incertos, pois não são divulgados. E como não é algo compartilhado com a comunidade para que seja feita uma evolução conjunta, a empresa pode estar despendendo recursos e esforços que poderiam ser melhores aplicados se

contassem com as experiências e debates que grandes práticos fora do contexto da Google podem oferecer.

Na segunda fase da pesquisa, seguindo as diretivas de uma Grounded Theory, foram inseridas novas estratégias para coleta de dados. Dessa forma, primeiramente foi feito um estudos para listar os principais blogs e fóruns que tratam sobre Engenharia de Software, e dessa forma, foi possível identificar importantes fontes de informações sobre Arquitetura e Design de Software para sistemas de larga escala. Considerando todos os cinco critérios de inclusão e três de exclusão de mecanismos de busca, o resultado foi 12 fontes de dados identificadas, as quais são:

1. **Stack OverFlow** - Um grande fórum de tecnologia global;
2. **Medium** - Um grande blog colaborativo global;
3. **Code Project** - Fórum e Blog colaborativo que reúne uma grande comunidade de práticos da Engenharia de Software;
4. **Martin Fowler** - Um dos mais referenciados Arquitetos de Software e criador de conteúdo voltado para sua área de atuação;
5. **Code Ranch** - Fórum popular de tecnologia que reúne uma grande comunidade de práticos;
6. **Chris Richardson** - Um dos mais referenciados Arquitetos de Software e criador de conteúdo voltado para sua área de atuação;
7. **Reddit** - Um grande fórum de tecnologia global;
8. **IBM** - Blog de uma grande empresa de tecnologia global;
9. **SRE Google** - Blog de uma grande empresa de tenologia global;
10. **Quora** - Um grande fórum global, com uma vasta comunidade de práticos da Engenharia de Software atuando em respostas e perguntas;
11. **Slide Share** - Uma plataforma de disponibilização gratuita de slides usados em apresentações, incluindo apresentações sobre Arquitetura e design de Software;
12. **DDD Community** - Blog que reúne todos os principais conceitos sobre o Domain-Driven Design.

Dentre essas doze, quatro foram excluídas após serem refinadas pelos critérios de inclusão e exclusão. A Tabela 2 lista todas as fontes analisadas e os respectivos critérios de exclusão, que justificam as fontes que foram desconsideradas. Os critérios seguem a numeração definida na Subseção 2.1.3.

Fonte de Dados	Inclusão	Justificativa
Stack OverFlow	Não	01 e 03
Medium	Sim	
Code Project	Sim	
Fowler	Sim	
Code Ranch	Não	02 e 03
Richardson	Sim	
Reddit	Não	03
IBM	Sim	
SRE Google	Sim	
Quora	Sim	
SlideShare	Sim	
DDD Community	Não	02

Tabela 2 – Fonte de Dados Selecionadas

Com as Fontes de dados definidas, foram utilizados os respectivos mecanismos de busca internos para selecionar materiais, cujo tema faz referência a termos de interesse. Os materiais selecionados são listados na Tabela 3. A primeira coluna contém o código referente a cada material utilizando o padrão especificado na Subseção 2.1.5.1.

Foram analisados 84 materiais, dentre os quais 34 foram excluídos pelos critérios listados na coluna mais a direita da Tabela 3. Cada critério de exclusão segue a numeração especificada na Subseção 2.1.2. Foram incluídos 50 materiais. O intervalo de publicação é de 2011 até 2021.

3.2.1 Resultados da fase de Open Coding

Na fase de 2.1.5.1 *Open Codin* foram relacionados os materiais com as categorias, conforme o processo especificado na Subseção 2.1.5.1. A Tabela 4 contém todos os materiais selecionados apresentados por códigos, e relacionando tais códigos às categorias estabelecidas. Na coluna mais a direita da tabela, há o número de materiais que corresponde a cada categoria. As categorias mais citadas são "Padrões de Projeto e Arquitetura", no processo de Design"e "Exemplos de implementação, design e metodologia". Isso revela o engajamento por parte da comunidade de autores, que disponibilizam diversos exemplos que ilustram como aplicar os conceitos e expõem os problemas apresentados, bem como a importância que os Padrões de Projeto e Arquitetura ainda têm na resolução desses problemas atualmente.

3.2.2 Resultados da fase Axial Coding

Ao realizar os procedimentos recomendados por (SCOTT; HOWELL, 2008) para a fase de Axial Coding, foi elaborado o 2.1.5.2 Guia de Relacionamento Condicional apresentado na Tabela 5. Esse Guia auxiliou na elaboração da 2.1.5.3 Matriz de Coding Reflexiva, na Tabela 6, que apresenta a categorial principal identificada por meio dessa pesquisa, a qual é "Lidar com a complexidade de sistemas distribuídos de larga escala usando Domain-Driven Design".

3.2.3 Resultados da fase Selective Coding

Após a saturação teórica ser parcialmente atingida, os conceitos levantados durante o processo de Grounded Theory, atrelado ao fluxo de GLR executado, são apresentados no Mapa Conceitual, na Figura 9. Esse diagrama ilustra a Story Line criada, consolidando os principais conceitos e é possível notar que tudo que foi abordado durante o estudo está contemplado no diagrama de modo relacional com os demais conceitos. A Tabela 7 também lista os principais problemas, pontuados pelos práticos autores dos materiais analisados, para o Design Arquitetural de Sistemas de Larga Escala e as soluções sugeridas. Os resultados apontam para a utilização do padrão arquitetural Microsserviços, afim de garantir escalabilidade, para a preocupação com a confiabilidade do sistema e para o uso das diretrizes do Domain-Driven Design, apresentado por (EVANS, 2003). Os resultados contidos na Tabela 7 são analisados e comentados na seção 4.

Código	Título	Aprovado?	Critérios de Exclusão
MF-01 (FOWLER, 2020b)	DomainDrivenDesign	Sim	
MF-02 (FOWLER, 2019b)	Software Architecture Guide	Não	1
MF-03 (FOWLER, 2017)	What do you mean by “Event-Driven”?	Sim	
MF-04 (FOWLER, 2015)	Microservice Trade-Offs	Sim	
MF-05 (FOWLER, 2014b)	BoundedContext	Não	1
MF-06 (FOWLER, 2011)	CQRS	Sim	
MF-07 (FOWLER, 2006b)	UbiquitousLanguage	Não	7
QR-01 (QUORA, 2021)	Is domain-driven design still relevant?	Sim	
QR-02 (QUORA, 2020)	Is domain driven design worth it?	Não	2
QR-03 (QUORA, 2019)	Why is architectural design important in software engineering?	Sim	
QR-04 (QUORA, 2018d)	Is it possible to carry out a large software project without a software architect?	Sim	
QR-05 (QUORA, 2018f)	Where can you find a forum for discussing software architecture and design?	Não	2
QR-06 (QUORA, 2018a)	How do you learn and develop skills required for software architectural design?	Sim	
QR-07 (QUORA, 2018c)	Is Domain Driven Design by Eric Evans worth reading?	Não	2
QR-08 (QUORA, 2018b)	How relevant is domain-driven design?	Não	2
QR-09 (QUORA, 2017a)	How do microservices make use of domain-driven design?	Sim	
QR-10 (QUORA, 2017c)	Why is design so difficult in software architecture?	Sim	
QR-11 (QUORA, 2017d)	Where can I get advice about software architecture design?	Não	2
QR-12 (QUORA, 2017b)	How do you learn and develop skills required for software architectural design?	Não	4
QR-13 (QUORA, 2016a)	Is domain driven design really worth it? Or is it just to create buzz?	Sim	
QR-14 (QUORA, 2016e)	What is the relationship between microservices and domain-driven design?	Sim	
QR-15 (QUORA, 2016b)	What are your thoughts on the future of Software Architecture/Design?	Não	1 e 2
QR-16 (QUORA, 2016c)	What basic things need to know about design software architecture?	Não	1 e 2
QR-17 (QUORA, 2016d)	What is software architecture in architectural design?	Não	1
QR-18 (QUORA, 2015b)	How is software architecture related to the design and development approaches?	Não	1
QR-19 (QUORA, 2015b)	How is software architecture related to the design and development approaches?	Não	4
QR-20 (QUORA, 2015a)	How is software architecture designed in an agile process?	Não	1
IBM-01 (IBM, 2020)	SOA vs. Microservices: What’s the Difference?	Sim	
IBM-02 (IBM, 2021)	SOA (Service-Oriented Architecture)	Sim	
IBM-03 (IBM, 2018)	Use domain-driven design to architect your cloud apps	Sim	
SRE-01 (GOOGLE, 2018a)	Introducing Non-Abstract Large System Design	Sim	
SRE-02 (GOOGLE, 2018b)	Exercises for non-abstract large systems design	Sim	
CP-01 (CODEPROJECT, 2017)	Software Principles and Patterns Revisited	Sim	
CP-02 (CODEPROJECT, 2015)	Domain Driven Design - Reflecting Business in the Domain of the Software	Sim	
RC-01 (RICHARDSON, 2021j)	What are microservices?	Sim	
RC-02 (RICHARDSON, 2021a)	Domain event	Não	3
RC-03 (RICHARDSON, 2021d)	Monolithic Architecture	Sim	
RC-04 (RICHARDSON, 2021c)	Decompose by business capability	Sim	
RC-05 (RICHARDSON, 2021e)	Decompose by subdomain	Sim	
RC-06 (RICHARDSON,)	Self-contained service	Não	3
RC-07 (RICHARDSON, 2021g)	Service per team	Sim	
RC-08 (RICHARDSON, 2021h)	Strangler application	Sim	
RC-09 (RICHARDSON, 2021f)	Circuit Breaker	Não	3
RC-10 (RICHARDSON, 2021)	Health Check API	Não	3
RC-11 (RICHARDSON, 2021b)	API Gateway / Backends for Frontends	Não	3
SS-01 (HAMID, 2021)	Microservices Architecture - Cloud Native Apps	Sim	
SS-02 (HOSTEDBYCONFLUENT, 2021)	Reusing Kafka Data Structure Between Projects | Laura Schornack and Maureen Penzenik, Northern Trust	Sim	
SS-03 (HAMID, 2020)	Domain Driven Design	Sim	
SS-04 (KAO, 2019)	DDD Taiwan Community 2019 01-26-1st-meetup-why ddd matters	Sim	
SS-05 (FOWLER, 2006b)	DDD for real	Não	4
SS-06 (HAMID, 2020)	Domain Driven Design	Não	4
SS-07 (HAMID, 2018)	Microservices Architecture - Bangkok 2018	Sim	
SS-08 (RICHARDSON, 2017)	A Pattern Language for Microservices	Sim	
SS-09 (MARTRAIRE, 2017)	DDD patterns that were not in the book	Sim	
SS-10 (SOYKAN, 2017)	Domain Driven Design(DDD) Presentation	Sim	
SS-11 (THAWARE, 2015)	Software architecture and software design	Não	1
ME-01 (HUSEYNLI, 2021)	Software Architecture Patterns: 5 minute read	Não	1
ME-02 (KALKMAN, 2021)	How To Enhance Your Software Architecture Design	Não	5
ME-03 (TUNE, 2021)	Domain, Subdomain, Bounded Context, Problem/Solution Space in DDD: Clearly Defined	Sim	
ME-04 (CHATUEV, 2020)	Strategic Domain-Driven Design	Sim	
ME-05 (MARTINEZ, 2021)	Domain-Driven Design: Everything You Always Wanted to Know About it, But Were Afraid to Ask	Sim	
ME-06 (TUNE, 2020)	Legacy Architecture Modernisation With Strategic Domain-Driven Design	Sim	
ME-07 (LAINE, 2020)	Domain-Driven Design: Things to Remember When Building a Bounded Context	Não	5
ME-08 (DESHPANDE, 2020)	Modern-Day Architecture Design Patterns for Software Professionals	Não	5
ME-09 (LUGAVERE, 2019)	Micro-Service Design Pattern: Dependency Driven Decomposition (DDD)	Não	1
ME-10 (TUNE, 2019)	Uncovering Hidden Business Rules with DDD Aggregates	Sim	
ME-11 (LAINE, 2019)	Domain-Driven Design in the era of Microservices	Sim	
ME-12 (VEGREVILLE, 2019)	Expressive error handling in TypeScript and benefits for domain-driven design	Não	3
ME-13 (GILL, 2019)	Part 1: Domain Driven Design like a pro	Sim	
ME-14 (NORELUS, 2019)	Implementing Domain-Driven Design for Microservice Architecture	Sim	
ME-15 (CRIULANSCY, 2020)	Domain-Driven Design for javascript developers	Não	3
ME-16 (OZUNLU, 2018)	Domain Driven Design (DDD)	Não	4
ME-17 (OZUNLU, 2018)	Domain Driven Design (DDD)	Não	6
ME-18 (NIKLAS, 2018)	Modern Software Architecture (1): Domain Driven Design	Sim	
ME-19 (CHUNG, 2018)	Clean Domain-Driven Design in 10 minutes	Sim	
ME-20 (CHAUHAN, 2018)	Designing microservices using Domain Driven Design	Sim	
ME-21 (DOUGLASS, 2018)	Domain Driven Design and Agile — They Work Together!	Não	1
ME-22 (BüRCKEL, 2020)	Practical Tips on Software Architecture Design, Part One	Sim	
ME-23 (HAQ, 2019)	Top 10 System Design Interview Questions for Software Engineers	Sim	
ME-24 (GRAçA, 2017)	DDD, Hexagonal, Onion, Clean, CQRS, ... How I put it all together	Sim	
ME-25 (MENEZES, 2017)	Usando DDD e Event Storming para Remodelar o Domínio de um Módulo Legado	Sim	
ME-26 (SAMOKHIN, 2017)	DDD Strategic Patterns: How To Define Bounded Contexts	Sim	
ME-27 (ZHU, 2017)	GraphQL and DDD: the Missing Link	Não	3
ME-28 (LELONEK, 2015)	DDD building blocks	Sim	

Tabela 3 – Seleção de materiais

Categorias	Códigos	Número de Materiais
Devops	MF-04 (FOWLER, 2015); QR-14 (QUORA, 2016e); IBM-01 (IBM, 2020); IBM-02 (IBM, 2021); SS-01 (HAMID, 2021); SS-08 (RICHARDSON, 2017); ME-13 (GILL, 2019); ME-14 (NORELUS, 2019);	08
Processos de negócio e desenvolvimento	MF-01 (FOWLER, 2020b); MF-04 (FOWLER, 2015); QR-03 (QUORA, 2019); IBM-01 (IBM, 2020); IBM-02 (IBM, 2021); IBM-03 (IBM, 2018); SRE-01 (GOOGLE, 2018a); SRE-02 (GOOGLE, 2018b); CP-01 (CODEPROJECT, 2017); RC-04 (RICHARDSON, 2021c); RC-05 (RICHARDSON, 2021e); SS-01 (HAMID, 2021); SS-03 (HAMID, 2020); SS-07 (HAMID, 2018); ME-01 (HUSEYNLI, 2021); ME-04 (CHATUEV, 2020); ME-05 (MARTINEZ, 2021); ME-06 (TUNE, 2020); ME-10 (TUNE, 2019); ME-13 (GILL, 2019); ME-14 (NORELUS, 2019); ME-25 (MENEZES, 2017); ME-26 (SAMOKHIN, 2017);	23
Regras de Negócio e de Domínio	MF-01 (FOWLER, 2020b); QR-14 (QUORA, 2016e); SS-03 (HAMID, 2020); ME-03 (TUNE, 2021); ME-05 (MARTINEZ, 2021); ME-14 (NORELUS, 2019); ME-25 (MENEZES, 2017); ME-26 (SAMOKHIN, 2017);	08
Problemas na implementação do Sistema	MF-03 (FOWLER, 2017); MF-04 (FOWLER, 2015); IBM-01 (IBM, 2020); IBM-03 (IBM, 2018); SRE-01 (GOOGLE, 2018a) ; RC-01 (RICHARDSON, 2021j); RC-04 (RICHARDSON, 2021c); RC-05 (RICHARDSON, 2021e); SS-08 (RICHARDSON, 2017); ME-10 (TUNE, 2019); ME-11 (LAINE, 2019); ME-14 (NORELUS, 2019);	12
Desafios no desenvolvimento de Sistemas de Larga Escala	IBM-03 (IBM, 2018); SS-01 (HAMID, 2021); SS-04 (KAO, 2019); SS-10 (SOYKAN, 2017); ME-04 (CHATUEV, 2020); ME-05 (MARTINEZ, 2021); ME-06 (TUNE, 2020); ME-10 (TUNE, 2019); ME-13 (GILL, 2019); ME-18 (NIKLAS, 2018); ME-22 (BüRCKEL, 2020); ME-25 (MENEZES, 2017);	12
Comunidade de práticos da Engenharia de Software	MF-01 (FOWLER, 2020b); MF-04 (FOWLER, 2015); SS-01 (HAMID, 2021); ME-03 (TUNE, 2021); ME-06 (TUNE, 2020); ME-13 (GILL, 2019);	6
Padrões de Projeto e Arquitetura	MF-01 (FOWLER, 2020b); MF-03 (TUNE, 2021); MF-04 (FOWLER, 2015); MF-06 (FOWLER, 2011); QR-06 (QUORA, 2018a); IBM-01 (IBM, 2020); IBM-02 (IBM, 2021); IBM-03 (IBM, 2018); SRE-02 (GOOGLE, 2018b); CP-01 (CODEPROJECT, 2017); RC-01 (RICHARDSON, 2021j); RC-03 (RICHARDSON, 2021d); RC-04 (RICHARDSON, 2021c); RC-05 (RICHARDSON, 2021e); RC-07 (RICHARDSON, 2021g); RC-08 (RICHARDSON, 2021h); SS-01 (HAMID, 2021); SS-03 (HAMID, 2020); SS-04 (KAO, 2019); SS-07 (HAMID, 2018); SS-08 (RICHARDSON, 2017); SS-09 (MARTRAIRE, 2017); SS-10 (SOYKAN, 2017); ME-05 (MARTINEZ, 2021); ME-06 (TUNE, 2020); ME-13 (GILL, 2019); ME-14 (NORELUS, 2019); ME-18 (NIKLAS, 2018); ME-19 (CHUNG, 2018); ME-22 (BüRCKEL, 2020); ME-26 (SAMOKHIN, 2017);	31
Melhores Práticas de Design e Arquitetura	SRE-02 (GOOGLE, 2018b); SS-01 (HAMID, 2021); SS-07 (HAMID, 2018)	3
Exemplos de implementação, design e metodologia	MF-03 (FOWLER, 2017); MF-06 (FOWLER, 2011); QR-03 (QUORA, 2019); QR-06 (QUORA, 2018a); QR-13 (QUORA, 2016a); QR-14 (QUORA, 2016e); IBM-01 (IBM, 2020); IBM-02 (IBM, 2021); IBM-03 (IBM, 2018); SRE-01 (GOOGLE, 2018a); CP-01 (CODEPROJECT, 2017); CP-02 (CODEPROJECT, 2015); RC-01 (RICHARDSON, 2021j); RC-03 (RICHARDSON, 2021d); RC-04 (RICHARDSON, 2021c); RC-05 (RICHARDSON, 2021e); SS-01 (HAMID, 2021); SS-03 (HAMID, 2020); SS-09 (MARTRAIRE, 2017); ME-03 (TUNE, 2021); ME-04 (CHATUEV, 2020); ME-06 (TUNE, 2020); ME-09 (LUGAVERE, 2019); ME-10 (TUNE, 2019); ME-13 (GILL, 2019); ME-14 (NORELUS, 2019); ME-20 (CHAUHAN, 2018); ME-22 (BüRCKEL, 2020); ME-24 (GRAçA, 2017); ME-25; ME-26;	31
Casos de Uso de Sistemas de Larga Escala	IBM-01 (IBM, 2020); SS-01 (HAMID, 2021); SS-03 (HAMID, 2020); SS-07 (HAMID, 2018); SS-10 (SOYKAN, 2017); ME-06 (TUNE, 2020); ME-13 (GILL, 2019); ME-19 (CHUNG, 2018); ME-23 (HAQ, 2019); ME-24 (GRAçA, 2017);	10
Problemas no processo de Design	MF-04 (FOWLER, 2015); MF-06 (FOWLER, 2011); QR-10 (QUORA, 2017c); QR-14 (QUORA, 2016e); IBM-01 (IBM, 2020); IBM-03 (IBM, 2018); SRE-01 (GOOGLE, 2018a); SRE-02 (GOOGLE, 2018b); CP-02 (CODEPROJECT, 2015); RC-01 (RICHARDSON, 2021j); RC-03 (RICHARDSON, 2021d); RC-04 (RICHARDSON, 2021c); RC-05 (RICHARDSON, 2021e); RC-07 (RICHARDSON, 2021g); SS-03 (HAMID, 2020); SS-07 (HAMID, 2018); SS-08 (RICHARDSON, 2017); SS-09 (MARTRAIRE, 2017); SS-10 (SOYKAN, 2017); ME-03 (TUNE, 2021); ME-04 (CHATUEV, 2020); ME-05 (MARTINEZ, 2021); ME-10 (TUNE, 2019); ME-13 (GILL, 2019); ME-14 (NORELUS, 2019); ME-19 (CHUNG, 2018); ME-23 (HAQ, 2019); ME-25 (MENEZES, 2017); ME-26 (SAMOKHIN, 2017);	29
Estratégias de Design e Arquitetura	MF-04 (FOWLER, 2015); QR-13 (QUORA, 2016a); IBM-03 (IBM, 2018); SRE-01 (GOOGLE, 2018a); SS-01 (HAMID, 2021); SS-03 (HAMID, 2020); SS-04 (KAO, 2019); SS-07 (HAMID, 2018); ME-03 (TUNE, 2021); ME-04 (CHATUEV, 2020); ME-05 (MARTINEZ, 2021); ME-06 (TUNE, 2020); ME-13 (GILL, 2019); ME-14 (NORELUS, 2019); ME-20 (CHAUHAN, 2018); ME-25 (MENEZES, 2017);	16
Sistemas modernos	IBM-03 (IBM, 2018); CP-01 (CODEPROJECT, 2017); RC-08 (RICHARDSON, 2021h); SS-01 (HAMID, 2021); SS-08 (RICHARDSON, 2017); ME-06 (TUNE, 2020); ME-18 (NIKLAS, 2018);	7
Diretrizes estratégicas de Design e Arquitetura	QR-01 (QUORA, 2021); RC-04 (RICHARDSON, 2021c); RC-05 (RICHARDSON, 2021e); SS-03 (HAMID, 2020); ME-05 (MARTINEZ, 2021); ME-06 (TUNE, 2020); ME-14 (NORELUS, 2019);	7
Confiabilidade do sistema	SRE-01 (GOOGLE, 2018a); SRE-02 (GOOGLE, 2018b); RC-01 (RICHARDSON, 2021j); ME-10 (TUNE, 2019)	4

Tabela 4 – Categorias e Materiais

Guia de Relacionamento Condicional						
Categoria	**O que**	**Quando**	**Onde**	**Porque**	**Como**	**Consequência**
DevOps	Desenvolvimento, Infraestrutura, Cultura Organizacional	Parte fundamental do Design de um Software de Larga Escala Distribuído, Ajuda uma organização migrar seu sistema para outra Arquitetura	Sistemas de Larga Escala e Distribuídos	Lidar com Complexidade Operacional dos Microsserviços, Agilidade, Escalabilidade, Resiliência, Confiabilidade	Deploy Independente, Entrega Contínua, Automação, Integração Contínua, Combinado aos princípios do DDD	Colaboração entre todos os envolvidos na entrega de um software de larga escala
Processos	Processo de Modelagem do Domínio, Processos de Negócio, Processo de aprendizado sobre Design e Domínio	Modelagem do sistema por domínios, Dividir em múltiplos contextos limitados, Design	Processo de Design de Softwares de larga escala, Aplicar conceitos do DDD	O Design de sistemas distribuídos envolve vários processos	Processo interativo NALSD da Google, Event Storming	Surgimento de novas ideias, Design Arquitetural da solução e tomadas de decisão de modo incremental.
Regras	Regras de Domínio, Regras de Negócio	Relacionar domínios com regras de negócio	Modelagem proposta pelo DDD	Servem como validação e critério de seleção	Definir o conjunto de regras envolvidas nos domínios e subdomínios	Aplicação mais eficaz do Aggregate Pattern, Definição de Contextos Limitados
Problemas	Conjunto de dificuldades observadas pelos práticos da Engenharia de Software	Design, Desenvolvimento e Manutenção de Sistemas de Larga Escala	Todo o ciclo de vida de um Software de Larga Escala	Sistemas de Larga Escala são complexos	Design, Modelagem, Desenvolvimento, Cultura Organizacional, Comunicação entre equipes	Impedimentos no processo de desenvolvimento
Desafios	Decompor um sistema Monolítico em vários serviços, comunicação organizacional, Gerenciamento e Integridade de dados em sistema distribuído, Modelagem por Domínios	Migrar de uma Arquitetura Monolítica para um Sistema baseado em serviços.	Remodelagem de sistemas, Modelagem de sistemas complexos	Sistemas de Larga Escala Modernos não são viáveis em uma Arquitetura Monolítica	Decisões de Design, Arquiteturais e de Negócios	Aumento da complexidade do processo de desenvolvimento
Comunidade	Profissionais da área de tecnologia da informação que compartilham experiencias e conteúdos entre si sobre Design de Software	Discussões feitas dentro do intervalo definido pelos critérios de aceitação de materiais por período de publicação	Blogs, fóruns ou sites	Resolução de problemas em comum; Discutir assuntos não definidos pela academia; Compartilhar novas descobertas;	Perguntas e respostas; Posts e artigos;	Disseminação de conhecimento; Apresentação de novas ideias; Identificação de problemas; Ambiente para compartilhar opiniões e críticas;
Patterns	Soluções para problemas genéricos que desenvolvedores	Design do Sistema	Arquitetura e Design de Software	Muitos problemas dentro da Engenharia de Software são	Avaliar quais padrões encaixa bem como	Utilização de um modelo de implementação já

Tabela 5 – Guia de Relacionamento Condicional

Matriz de Coding Reflexiva				
Categoria Principal	*Lidar com a complexidade de sistemas distribuídos de larga escala usando Domain-Driven Design*			
Processos	**Arquitetura**	**Design**	**DevOps**	**Comunicação**
Propriedades	Padrões Arquiteturais e de Projeto	Modelagem de Sistemas Complexos	Automação	Linguagem
Dimensões	Sistemas de Larga Escala Modernos não são viáveis em uma Arquitetura Monolítica; Migrar de uma Arquitetura Monolítica para um Sistema baseado em serviços;	Remodelagem de sistemas legados; Modelagem de sistemas complexos; Relacionar domínios com regras de negócio;	Deploy Independente, Entrega Contínua, Automação, Integração Contínua; Partes do Sistema distribuídas por vários times diferentes; Times localizados em espaços diferentes;	Cultura Organizacional; Comunicação entre equipes; Linguagem comum entre equipe técnica e stakeholders;
Contextos	Microsserviços	Domain-Driven Design	Infraestrutura e Desenvolvimento	Linguagem Ubíqua
Modos de entender as consequências	Tomada de melhores decisões	Redução da Complexidade	Agilidade	Bom relacionamento

Tabela 6 – Matriz de Coding Reflexiva

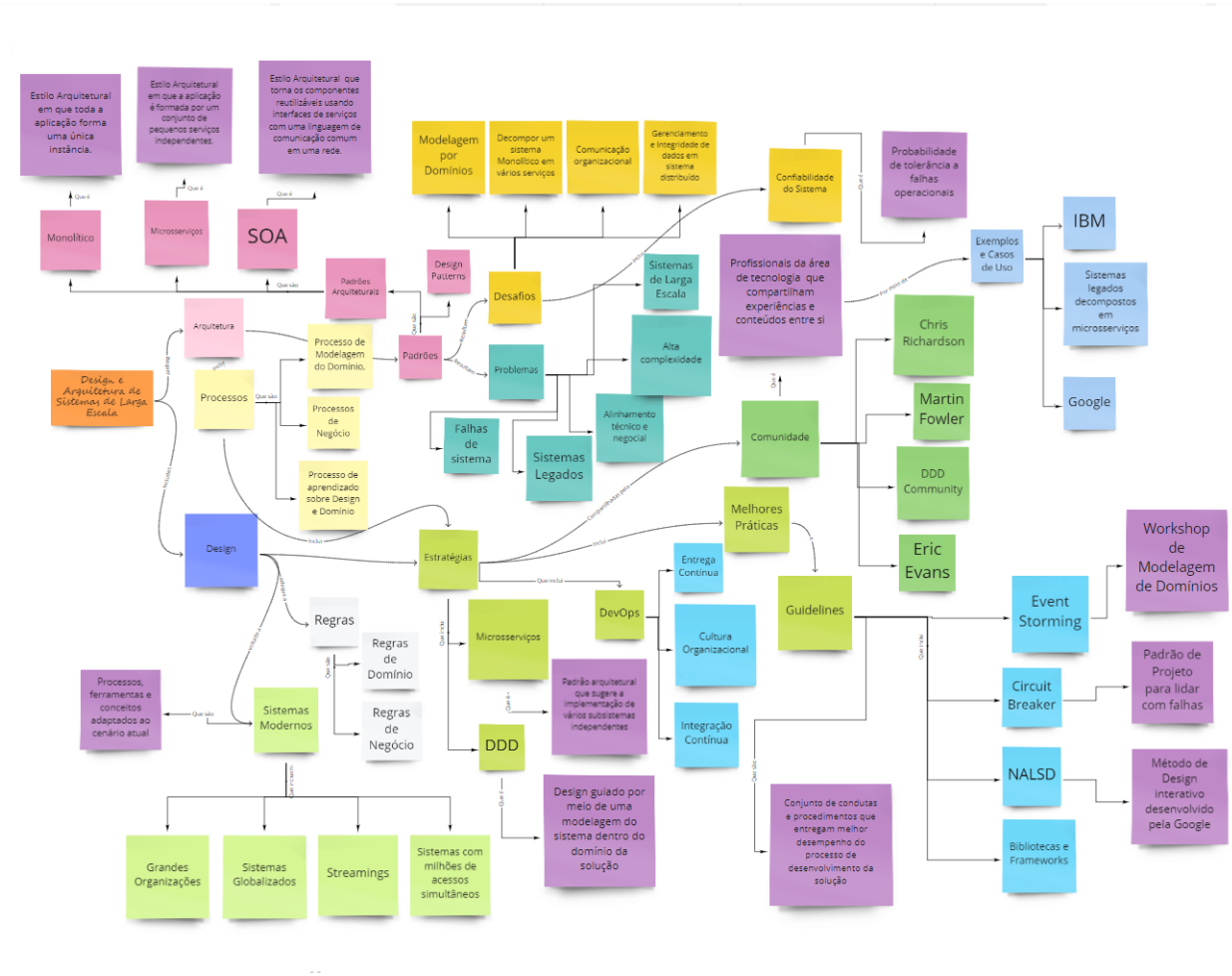

Figura 9 – Mapa Conceitual

Problemas	**Soluções**
Toda solução limitada ao uso de um único conjunto de tecnologias e ferramentas	- Uso da Arquitetura Microsserviços que viabiliza o uso das melhores ferramentas para cada caso particular; - Cada microsserviço, sendo independente, pode ser escrito em linguagens de programação diferentes e usar diferentes tecnologias para armazenamento de dados. (GILL, 2019) (RICHARDSON, 2021d)
Mudanças afetam toda a aplicação em um sistema Monolítico	- Uso da Arquitetura Microsserviços que viabiliza o redeploy de apenas serviços modificados, sem a necessidade de compilação e deploy de toda aplicação. (FOWLER, 2015)
Alto Risco para Falhas em cada serviço em um sistema em Microsserviços	- Automação na recuperação do sistema após falhas; - Monitoramento da aplicação em tempo real tanto em elementos arquiteturais, como em métricas negociais relevantes. (FOWLER, 2015) (GOOGLE, 2018a)
Dificuldade em dividir por serviços	- Os serviços devem ser componentes que possibilitam modificações que não afetem outros, ainda que tenham alguma interação. (FOWLER, 2015) (RICHARDSON, 2021c) (RICHARDSON, 2021e) (RICHARDSON, 2017) (CHATUEV, 2020) (SAMOKHIN, 2017)
Aumento da complexidade para sistema distribuído em um sistema em Microsserviços	- O DDD é um conjunto de ferramentas, práticas e padrões que ajudam no design dos sistemas e a lidar com complexidades; - Técnicas de DevOps reduzem a complexidade operacional de compilação, deploy e operação dos Microsserviços; - O DDD estrutura a complexidade lógica não só do código, mas no entendimento da aplicação como um todo. (FOWLER, 2015) (RICHARDSON, 2017) (MARTINEZ, 2021) (LAINE, 2019) (SAMOKHIN, 2017) (IBM, 2018)
Mudança em um serviço pode afetar outro em um sistema em Microsserviços	- Usando o conceito de contexto limitado, o DDD divide um domínio complexo em vários contextos e mapeia o relacionamento entre eles; - Os serviços devem ser desenhados para serem o mais tolerante possível a mudanças em seus serviços relacionados; - Os relacionamentos entre serviços devem ter poucas dependências de dados, como entidades autônomas. (FOWLER, 2015) (IBM, 2020) (LAINE, 2019)

Falhas de comunicação por não usar uma linguagem em comum, ou com ambiguidade, entre os Stakeholders	- A modelagem proposta pelo DDD atua como uma linguagem que facilita a comunicação; - A linguagem Ubíqua ajuda a alinhar pessoas com diferentes áreas de atuação e pontos de vista; - Os conceitos definidos por todos os Stakeholders devem refletir nos nomes das classes, variáveis e etc, do código do sistema. (QUORA, 2017c) (IBM, 2018) (MARTINEZ, 2021)
Sistema tem que ser seguro/confiável (Reliability)	- Pode-se testar a tolerância a falhas e o monitoramento por induzir os serviços e datacenters a falhas; - Monitoramento em tempo real de toda a aplicação; - Incluir perguntas tais como “É resiliente?”, “O que acontece quando um componente falha?” e “Como o sistema reage a uma falha de todo o datacenter?” no processo de design; - Um design interativo auxilia no exame de pontos fortes e fracos; - Uso do pattern Circuit Breaker. (GOOGLE, 2018a) (MARTINEZ, 2021) (SAMOKHIN, 2017)
Padrões não servem para todas os casos	- Bibliotecas são compartilhadas pela comunidade, fornecendo códigos que foram testados na prática e que oferecem a liberdade de usar outra abordagem para resolver problemas similares; - Ao invés de usar o conjunto de padrões definidos por artigos acadêmicos, times que desenvolvem microsserviços podem preferir produzir ferramentas que auxiliam em problemas similares. (GOOGLE, 2018b) (HAMID, 2020)
Várias limitações para larga escala um sistema Monolítico	- A Arquitetura Microsserviços é um padrão alternativo que supera as limitações arquiteturais monolíticas, como deploy continuo e escalabilidade. (RICHARDSON, 2021j) (RICHARDSON, 2021d)
Foco apenas no livro “Domain-Driven Design: Tackling Complexity in the Heart of Software” escrito por Eric Evans e publicado em 2003	- O livro de Eric Evans sugere a análise de outros padrões e inspirações não abordadas no livro; - Com o avanço tecnológico desde a publicação do livro de Eric Evans, novas ideias estão emergindo regularmente; - Os desenvolvedores podem compartilhar seus próprios padrões. (MARTRAIRE, 2017) (SOYKAN, 2017) (TUNE, 2021) (GILL, 2019)

Dificuldade em entender o domínio e modelá-lo	- Alinhar o software e os contextos organizacionais; - Para modelar usando o DDD pode-se usar qualquer coisa que mostre os conceitos, relacionamentos e regras do domínio, por exemplo com código ou bloco de notas; - Especialistas técnicos e de negócios modelam o domínio de forma colaborativa para identificar as reais regras de negócio; - Sempre envolver os experts do domínio, que são pessoas que possuem o conhecimento sobre porque as decisões estão sendo tomadas; - Às vezes é necessário excluir partes do domínio quando se tornam muito vastas e complexas; - O domínio pode ser decomposto em subdomínios para lidar com a complexidade e separar partes importantes do resto do sistema; - O Event Storming pode ser utilizado como um workshop feito para entender o domínio; - Adotar o design estratégico proposto pelo DDD. (RICHARDSON, 2017) (TUNE, 2021) (TUNE, 2019) (GILL, 2019) (NORELUS, 2019) (HAQ, 2019) (SAMOKHIN, 2017) (RICHARDSON, 2021c) (RICHARDSON, 2021e)
Tecnologia e código não refletem o domínio da solução e regras de negócio	- O modo de resolver isso é entender melhor o domínio do negócio; - Pode-se explorar modelos alternativos que adicionam complexidade técnica, mas oferecem benefícios para o negócio. (MARTINEZ, 2021) (TUNE, 2019) (CODEPROJECT, 2015)
Tratar todos os problemas como técnicos	- O DDD encoraja o desenvolvimento da solução constantemente envolvendo o modelo do domínio, deixando de lado detalhes inicialmente irrelevantes como linguagens de programação, tecnologias de infraestrutura e etc; - Os desenvolvedores trabalham com os experts do domínio de modo colaborativo com a intenção de constantemente refinar o Modelo de Domínio, assim são "forçados" a aprender detalhes importantes e princípios do problema negocial que estão tentando resolver, ao invés de só produzir código mecanicamente. (MARTINEZ, 2021) (TUNE, 2019) (CODEPROJECT, 2015) (FOWLER, 2015)

Tabela 7 – Principais problemas e soluções relatados pelos autores

4 Discussão

Essa pesquisa pontuou alguns problemas que são superados pela implementação do padrão arquitetural Microsserviços, tais como: Toda solução ser limitada ao uso de um único conjunto de tecnologias e ferramentas, mudanças pontuais que afetam a aplicação inteira e várias limitações de um sistema monolítico possui ao se tratar de escalabilidade. Certamente, várias técnicas, em uso pela comunidade de desenvolvedores de microsserviços, cresceram a partir de experiências daqueles que trabalham em sistemas de larga escala e em grandes organizações. Os resultados obtidos por esse estudo sintetizam boa parte do que vem sendo compartilhado (FOWLER, 2015).

Tendo Microsserviços como estilo arquitetural, é possível desenvolver uma única aplicação como um conjunto de pequenos serviços. Esses serviços podem ser construídos em torno de características do negócio, e com deploy independente e completamente automatizado, utilizando as práticas de DevOps. Dessa forma, já que o gerenciamento desses serviços é minimamente centralizado, cada um pode ser escrito em diferentes linguagens de programação, frameworks, banco de dados, padrões de projeto, arquiteturas e etc (FOWLER, 2015). Isso faz com que, diferentemente de um monolito, os microsserviços não sejam limitados por um único conjunto de ferramentas e tecnologias.

Outro fator que afeta a escalabilidade de um monolito é que qualquer mudança requer que seja feita compilação e deploy da aplicação inteira. Com o uso de microsserviços, no entanto, pode-se refazer o deploy apenas do serviço, ou serviços, que foi modificado (FOWLER, 2015).

De acordo com os práticos e autores dos materiais analisados, a Arquitetura em Microsserviços é uma melhor opção para soluções que exigem larga escalabilidade. No entanto, esse padrão arquitetural possuiu limitações e problemas que são base para várias discussões feitas pela comunidade, tais como o alto risco de ocorrer falhas nos microsserviços implementados, na dificuldade de definir o escopo de cada serviço, o aumento da complexidade de um sistema distribuído, lidar com mudanças em um serviço que podem afetar outro, e garantir confiabilidade do sistema. Além disso, há competências técnicas específicas, que são exigidas para o desenvolvimento de soluções complexas e para a implementação de um sistema utilizando a Arquitetura Microsserviços. Isso tem sido provado pelo fato de que o design de soluções de larga escala para sistemas distribuídos tende a ser uma parte padrão das entrevistas em engenharia de software atualmente, e a falta de experiência no desenvolvimento de sistemas de larga escala é notada por meio delas.(HAQ, 2019).

Para superar essas debilidades ao construir microsserviços, a tendência observada

durante esses estudo é o uso do 1.3 Domain-Driven Design (DDD), uma abordagem para design de software com alta complexidade. Essa abordagem foi apresentada por Eric Evans (EVANS, 2003) por volta do ano 2003 e ainda é muito utilizada em sistemas modernos. Os principais princípios são: A modelagem do sistema em frações que estejam alinhadas com as reais necessidades do negócio, e métodos para que o entendimento do domínio do negócio seja uma prioridade.

Um sistema distribuído, como proposto pela Arquitetura Microsserviços, eleva consideravelmente sua complexidade, porém a comunidade adepta ao DDD vem utilizando essa abordagem por meio da união de aspectos técnicos e não técnicos, e propondo um conjunto de ferramentas, práticas e padrões que ajudam no design de sistemas de larga escala, ajudam a lidar com complexidades e facilitam a construção de um sistema com sucesso (IBM, 2018) (MARTINEZ, 2021). O que Eric Evans criou foi uma maneira de estruturar a complexidade lógica, não só do código, mas no entendimento de toda a solução (LAINE, 2019).

Um conjunto de práticas que fortalecem o uso do DDD com sucesso fazem parte da cultura DevOps. Essa cultura pode ser usada para ajudar a organização na transição de uma arquitetura para outra, oferecendo suporte para necessidades específicas, como por exemplo a migração da arquitetura SOA para Microsserviços (IBM, 2020). Além disso, a cultura DevOps é uma forma de mitigar problemas que surgem com o aumento da complexidade operacional de um sistema distribuído, por meio de integração contínua, entrega contínua e uma comunicação eficiente e colaborativa entre desenvolvedores, operações e todos os envolvidos na entrega do software (HAMID, 2021). Uma mudança cultural é difícil, especialmente em largas organizações, porém dispensar as práticas de DevOps pode inviabilizar a transição de uma aplicação monolítica para microsserviços (FOWLER, 2015).

A Google, baseando-se em vasta experiência em desenvolvimento de sistemas, considera que a confiabilidade é o requisito mais crítico de um sistema em ambiente de produção. Por isso, a empresa sugere que o processo de design de software de larga escala seja interativo, como 1.7 NALSD, sempre considerando como prioridade a tolerância a falhas. Sabe-se que um sistema pode falhar a qualquer momento, então deve ser uma preocupação dos envolvidos no design do sistema sempre se perguntar "O que aconteceria se?"durante todo o processo de design (GOOGLE, 2018a). Ao utilizar a abordagem DDD, a escolha das agregações, que compõem partes do domínio relacionadas e delimitadas por um contexto, e que podem refletir no escopo de cada microsserviço ou não, deve ser feita com cuidado. Agregações largas e complexas ou vários processos assíncronos podem aumentar os custos de manutenção e afetar a confiabilidade das aplicações (TUNE, 2019).

Outra forma de garantir a confiabilidade do sistema é utilizar o padrão de projeto Circuit Breaker. A implementação envolve um service client que deve invocar um serviço

remoto via proxy, que funciona similarmente com um circuit breaker elétrico, conhecido como "disjuntor"em português. Quando o número de falhas consecutivas atinge um limiar, o circuit breaker passa a interromper a comunicação, e durante um período de timeout, todas as tentativas para invocar o serviço remoto falham imediatamente. Após o timeout expirar, o circuit breaker permite que um número limitado de requisições de teste passem por ele. Se existir uma falha, o periodo de timeout começa novamente. A Figura 10 mostra a implementação utilizando a lógica do Circuit Breaker escrita em Scala. O benefício do uso desse padrão é o controle de falhas de um serviço, tornando possível manipulá-las. Porém, esse padrão de projeto deve ser implementado com cautela, já que escolher valores para o período de timeout é um desafio, pois pode-se criar falsos positivos ou introduzir latência excessiva (RICHARDSON, 2021f).

```
@Component
class RegistrationServiceProxy @Autowired()(restTemplate: RestTemplate) extends RegistrationService {

  @Value("${user_registration_url}")
  var userRegistrationUrl: String = _

  @HystrixCommand(commandProperties=Array(new HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds", value="800")))
  override def registerUser(emailAddress: String, password: String): Either[RegistrationError, String] = {
    try {
      val response = restTemplate.postForEntity(userRegistrationUrl,
        RegistrationBackendRequest(emailAddress, password),
        classOf[RegistrationBackendResponse])
      response.getStatusCode match {
        case HttpStatus.OK =>
          Right(response.getBody.id)
      }
    } catch {
      case e: HttpClientErrorException if e.getStatusCode == HttpStatus.CONFLICT =>
        Left(DuplicateRegistrationError)
    }
  }
}
```

Figura 10 – Implementação de um Circuit Breaker (RICHARDSON, 2021f)

Os benefícios do uso do DDD em aplicações de larga escala e com arquitetura Microsserviços foram mapeados, mas ainda há algumas dificuldades em aderir essa abordagem. De acordo com o que foi analisado, a maior dificuldade em usar o DDD é entender o domínio e modelá-lo. É provável que o problema seja a mudança de paradigma ao se pensar na solução, direcionando o foco ao objetivo e contexto do que deve ser entregue, e não em apenas como deve ser feito.

Tecnologias emergem frequentemente, endereçando os maiores gargalos deixados por suas antecessoras. No entanto, o domínio dessas ferramentas não é suficiente para desenvolver uma solução com êxito duradouro. Essa é uma das filosofias por trás do DDD, que sugere que antes de mais nada sejam feitos esforços para entender e alinhar o entendimento do domínio entre todas as partes interessadas.

As soluções para enfrentar esses problemas fazem parte de longas discussões em fóruns, envolvendo muitas pessoas da comunidade DDD (TUNE, 2021). Uma delas é identificar os experts do domínio, aqueles que possuem conhecimento sobre todos os detalhes

do negócio. Unindo especialistas técnicos e de negócios, é possível modelar o domínio de forma colaborativa e assim identificar as reais regras de negócio.

Na realização desse estudo foram mapeados 41 exemplos e casos de uso que foram disponibilizados pelos autores para ajudar a comunidade a entender os conceitos apresentados e como aplicá-los. Um desses exemplos em (TUNE, 2019) ilustra como identificar regras de negócio que podem estar ocultas por meio das agregações propostas pelo DDD.

Nesse exemplo, o autor pontua que é importante identificar quais regras de negócio devem ter sucesso ou não juntas, porque isso resulta em implicações significativas tanto para o negócio quanto para o nível técnico. O cenário apresentado é o preenchimento do nome do usuário em uma criação de conta. O usuário desse sistema hipotético deve especificar seu primeiro, do meio e último nome antes da conta ser criada, e ela não pode ser criada sem esses três nomes. Quando um usuário insere seus dados, se ele não preenche todos os critérios, a criação de sua conta será rejeitada. A regra de negócio nesse caso é que o usuário não deve existir no sistema sem o primeiro, do meio e último nome, e essa seria uma Regra de Negócio Invariante. A partir da descoberta de regras potencialmente invariantes, pode-se usar perguntas tais como propostas pelo modelo interativo de design, 1.7 NALSD, da Google, como "O que aconteceria se?"e pensar em ações compensativas, dessa forma é possível fazer agregações de partes do negócio estratégicas.

O Event Storming, pode ajudar Modelar o domínio no qual o sistema a ser desenvolvido está inserido, pois promove uma profunda conversa sobre o domínio (TUNE, 2019). Esse método pode ser conduzido como um workshop, com o objetivo de modelar um novo sistema ou remodelar um sistema legado. É uma técnica de design que contribui para a agilidade do processo e engaja experts do domínio técnicos e de negócio, juntamente com desenvolvedores, a aprenderem sobre os processos do negócio e seu domínio. Dentre as vantagens estão: aprendizado acelerado do grupo, interação entre experts do domínio e equipe técnica, e a possibilidade de que ao final seja possível ter esboços de testes de aceitação, interfaces, atores do futuro sistema e entre outros elementos (MENEZES, 2017).

O Event Storming é composto por etapas, e cada etapa insere mais detalhes ao modelo final. Durante o processo são utilizados post-its, canetas, papel, entre outros materiais que possam servir como registro e estimulem a interação e surgimento de ideias. A expectativa é que, ao final, todos os envolvidos no workshop tenham conhecimento profundo do negócio, sem a barreira de abstrações técnicas. Além disso, que seja produzida a modelagem do sistema necessária, para tomadas de decisão mais assertivas na implementação de uma arquitetura Microsserviços (MENEZES, 2017).

Outro benefício da utilização de um workshop para modelar o domínio do sistema é a interação dos envolvidos na construção da solução. Um dos problemas pontuados pela comunidade de software no desenvolvimento de sistemas de larga escala é a falha

de comunicação entre os stakeholders, que possivelmente seja resultado da falta de uma linguagem comum entre esses profissionais. Para mitigar isso, o DDD propõe a utilização de uma linguagem Ubíqua, afim de alinhar pessoas de diferentes áreas de atuação e pontos de vista.

Para possibilitar essa colaboração entre times técnicos e de negócios, o Modelo de Domínio deve usar uma linguagem que une jargões técnicos e de negócios, e encontrar um equilíbrio no qual todos os membros do time podem entender e concordar no mesmo nível. Isso define bem o que é a linguagem Ubíqua.

Usando essa linguagem de forma bem definida, pode-se observar melhorias em cada interação entre times técnicos e de negócios, tornando-os menos ambíguos e mais efetivos (MARTINEZ, 2021). Uma importante parte do DDD é manter toda a comunicação em uma linguagem que faça sentido para o domínio, e assim, tendo todos os stakeholders completamente compreendendo uns aos outros, pode-se trabalhar mais eficientemente (GILL, 2019).

Após ter profundo conhecimento sobre o domínio, o time técnico deve inserir esses conhecimentos em suas decisões com relação ao código e as tecnologias utilizadas. Os conceitos definidos por todos os stakeholders devem refletir no nome das classes, variáveis e etc, no código do sistema (GILL, 2019). E algumas decisões podem ser feitas baseadas nos benefícios do negócio, ainda que eleve a complexidade técnica. Dessa forma, as tecnologias utilizadas e o código estarão alinhados com o domínio da solução e das regras de negócio. O DDD encoraja que o desenvolvimento da solução esteja constantemente envolvido com o Modelo do Domínio, não destacando detalhes menos importantes como qual linguagem de programação, tecnologias de infraestrutura e etc, usar (MARTINEZ, 2021).

De 2003 para os dias atuais muitas coisas mudaram dentro das organizações. Vários avanços tecnológicos foram realizados e o cenário de desenvolvimento de software globalizado reflete necessidades que não existiam anos atrás. Com esses argumentos, alguns autores criticam o que foi publicado por Eric Evans sobre o DDD (MARTRAIRE, 2017) (SOYKAN, 2017) (TUNE, 2021) (GILL, 2019). Em resumo, esses autores sugerem que novas ideias emergem com regularidade e novos padrões de projeto são criados a partir de experiencias com sistemas modernos, o que não exclui o uso do DDD, mas o incrementa.

O compartilhamento de informações atual é possivelmente maior do que em 2003, o que contribui para a propagação de conhecimento dentro da comunidade de software e tecnologia. Blogs, fóruns, sites, vídeos e outros meios estão carregados de fontes de aprendizado, trocas de experiências, discussões, exemplos e oportunidades. O avanço tecnológico é acelerado, por isso há uma preocupação em evoluir o que foi iniciado no passado para adequar a nova realidade. Dentro desse contexto, os desenvolvedores também chegaram a conclusão que existem alguns problemas relacionados a padrões sugeridos pela academia, e um dos maiores é a dificuldade de customização, já que padrões não servem

para todos os casos.

Uma forma de contornar a inflexibilidade de alguns padrões ou da dificuldade de implementá-los, adotada pela comunidade de desenvolvimento de software, é a disponibilização de bibliotecas gratuitamente em repositórios de código remotos, fornecendo assim implementações que foram testadas na prática e que oferecem liberdade na utilização de outras abordagens ao resolver problemas similares. Seguir a filosofia de compartilhar códigos e documentação resulta em várias ferramentas para o desenvolvimento de software de larga escala mais eficientes, tais como Node.js, Kubernetes, Docker e entre outros (KUBERNETES, 2021; Node.js, 2021; Docker, 2021).

5 Conclusão

O objetivo geral desse trabalho foi identificar quais são as estratégias que os práticos da engenharia de software estão adotando no design arquitetural de sistemas de larga escala. Foi possível observar que o estilo arquitetural mais adotado para sistemas dessa magnitude é a Arquitetura Microsserviços. Essa estratégia possui vários ganhos, porém muitos desafios precisam ser superados para que o resultado de seu uso não seja desastroso. O ponto mais crítico destacado foi garantir a confiabilidade de um sistema distribuído, e para que isso seja superado é necessário adotar uma estratégia de monitoramento de falhas em tempo real.

Além dos requisitos não funcionais do sistema, as regras de negócio de sistemas complexos também são difíceis de serem definidas e atendidas. Para lidar com a complexidade de sistemas de larga escala, as boas práticas publicadas pelo autor (EVANS, 2003) ainda são utilizadas, mesmo após vários anos de uso. O detalhe importante é que o Domain-Driven Design é atualmente aprimorado dentro da vasta comunidade de software. Novos padrões, novas bibliotecas, novas técnicas, novas abordagens e novas tecnologias emergem com muita frequência, e a disponibilização dessas informações e ferramentas de maneira livre contribui para um ambiente de colaboração global, que auxilia, de modo significativo, na forma como as organizações lidam com a complexidade de seus sistemas.

A cultura organizacional contribui consideravelmente para o sucesso do design arquitetural de sistemas de grande complexidade, como os de larga escala. As práticas de DevOps são vistas por muitos autores como indispensáveis, e a linguagem Ubíqua entre o time é um dos princípios chave dentro do conjunto de ferramentas propostos pela abordagem DDD. O objetivo é que todos envolvidos no negócio estejam completamente alinhados e que as decisões tomadas e tudo que é gerado, como o código, reflita o domínio da solução.

Tendo em vista o conjunto de conceitos que foram sintetizados durante essa pesquisa, uma evolução deste trabalho poderá contar com o desenvolvimento de uma ontologia de domínio, realizando um modelo de dados que representem esse conjunto de conceitos dentro do domínio do Design Arquitetural de Software e o relacionamento entre eles. O Mapa conceitual, Figura 9, é um vantajoso recurso para dar início ao desenvolvimento dessa ontologia.

Referências

Atlassian. *SLA vs. SLO vs. SLI - Differences*. 2021. Disponível em: <https://www.atlassian.com/incident-management/kpis/sla-vs-slo-vs-sli>. Citado na página 26.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. Pearson Education, 2003. (SEI Series in Software Engineering). ISBN 9780321680419. Disponível em: <https://books.google.com.br/books?id=ZY6UZTjBnGQC>. Citado na página 17.

BEYER, B. et al. (Ed.). *The site reliability workbook: practical ways to implement SRE*. Sebastopol, CA: O'Reilly Media, 2018. OCLC: on1029786800. ISBN 9781492029502. Citado 4 vezes nas páginas 24, 25, 27 e 28.

BüRCKEL, M. *Practical Tips on Software Architecture Design, Part One*. 2020. Disponível em: <https://medium.com/@mbue/practical-tips-on-software-architecture-design-part-one-1c6bb0167157>. Citado 2 vezes nas páginas 46 e 47.

CHATUEV, M. *DDD for microservices*. 2020. Disponível em: <https://medium.com/@chatuev/ddd-for-microservices-4778a363c071>. Citado 3 vezes nas páginas 46, 47 e 51.

CHAUHAN, A. *Designing microservices using Domain Driven Design*. 2018. Disponível em: <https://helloansh.medium.com/designing-microservices-using-domain-driven-design-e013caa0ac22>. Citado 2 vezes nas páginas 46 e 47.

CHUNG, T. *Clean Domain-Driven Design in 10 minutes*. 2018. Disponível em: <https://medium.com/hackernoon/clean-domain-driven-design-in-10-minutes-6037a59c8b7b>. Citado 2 vezes nas páginas 46 e 47.

CINTRA, R. d. C.; VENDRAMEL, W. Padrões de migração de sistemas legados para arquitetura baseada em microsserviços. 2019. Disponível em: <https://revistas.unifacs.br/index.php/rsc/article/download/5786/3805>. Citado na página 15.

CODEPROJECT. *Domain Driven Design - Reflecting Business in the Domain of the Software*. 2015. Disponível em: <https://www.codeproject.com/Articles/1020932/Domain-Driven-Design-Reflecting-Business-in-the-Do>. Citado 3 vezes nas páginas 46, 47 e 53.

CODEPROJECT. *CP-01 Software Principles and Patterns Revisited*. 2017. Disponível em: <https://www.codeproject.com/Articles/1169509/Software-Principles-and-Patterns-Revisited>. Citado 2 vezes nas páginas 46 e 47.

CRIULANSCY, P. *Domain-Driven Design for javascript developers*. 2020. Disponível em: <https://medium.com/spotlight-on-javascript/domain-driven-design-for-javascript-developers-9fc3f681931a>. Citado na página 46.

DESHPANDE, T. *Modern-Day Architecture Design Patterns for Software Professionals*. 2020. Disponível em: <https://betterprogramming.pub/modern-day-architecture-design-patterns-for-software-professionals-9056ee1ed977>. Citado na página 46.

Docker. 2021. Disponível em: <https://www.docker.com/>. Citado na página 60.

DOUGLASS, M. *Domain Driven Design & Agile — They Work Together!* 2018. Disponível em: <https://codeburst.io/domain-driven-design-agile-they-work-together-329f059923f5>. Citado na página 46.

EVANS, E. *Domain-driven design: tackling complexity in the heart of software*. Boston: Addison-Wesley, 2003. ISBN 9780321125217. Citado 5 vezes nas páginas 18, 42, 45, 56 e 61.

FACOM, F. d. C. U. F. D. U. *Padrões Grasp*. 2021. Disponível em: <http://www.facom.ufu.br/~bacala/ESOF/05a-Padr%C3%B5es%20GRASP.pdf>. Citado na página 23.

FOWLER, M. *UbiquitousLanguage*. 2006. Disponível em: <https://martinfowler.com/bliki/UbiquitousLanguage.html>. Citado na página 18.

FOWLER, M. *UbiquitousLanguage*. 2006. Disponível em: <https://martinfowler.com/bliki/UbiquitousLanguage.html>. Citado na página 46.

FOWLER, M. *CQRS*. 2011. Disponível em: <https://martinfowler.com/bliki/CQRS.html>. Citado 2 vezes nas páginas 46 e 47.

FOWLER, M. *BoundedContext*. 2014. Disponível em: <https://martinfowler.com/bliki/BoundedContext.html>. Citado 2 vezes nas páginas 7 e 19.

FOWLER, M. *BoundedContext*. 2014. Disponível em: <https://martinfowler.com/bliki/BoundedContext.html>. Citado na página 46.

FOWLER, M. *Microservices*. 2014. Publisher:. Disponível em: <https://martinfowler.com/articles/microservices.html>. Citado na página 22.

FOWLER, M. *Microservices*. 2015. Disponível em: <https://martinfowler.com/articles/microservices.html>. Citado 6 vezes nas páginas 46, 47, 51, 53, 55 e 56.

FOWLER, M. *What do you mean by "Event-Driven"?* 2017. Disponível em: <https://martinfowler.com/articles/201701-event-driven.html>. Citado 2 vezes nas páginas 46 e 47.

FOWLER, M. *Software Architecture Guide*. 2019. Disponível em: <https://martinfowler.com/architecture/>. Citado na página 17.

FOWLER, M. *Software Architecture Guide*. 2019. Disponível em: <https://martinfowler.com/architecture/>. Citado na página 46.

FOWLER, M. *DomainDrivenDesign*. 2020. Disponível em: <https://martinfowler.com/bliki/DomainDrivenDesign.html>. Citado 3 vezes nas páginas 18, 19 e 42.

FOWLER, M. *DomainDrivenDesign*. 2020. Disponível em: <https://martinfowler.com/bliki/DomainDrivenDesign.html>. Citado 2 vezes nas páginas 46 e 47.

FREECODECAMP. *The Model View Controller Pattern – MVC Architecture and Frameworks Explained*. 2021. Disponível em: <https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/>. Citado 2 vezes nas páginas 7 e 20.

GAMMA, E. (Ed.). *Design patterns: elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995. (Addison-Wesley professional computing series). ISBN 9780201633610. Citado na página 23.

GILL, A. *Part 1: Domain Driven Design like a pro*. 2019. Disponível em: <https://medium.com/raa-labs/part-1-domain-driven-design-like-a-pro-f9e78d081f10>. Citado 6 vezes nas páginas 46, 47, 51, 52, 53 e 59.

Google. *Google - Site Reliability Engineering*. 2018. Disponível em: <https://sre.google/workbook/non-abstract-design/>. Citado na página 24.

GOOGLE. *Google - Site Reliability Engineering*. 2018. Disponível em: <https://sre.google/workbook/non-abstract-design/>. Citado 5 vezes nas páginas 46, 47, 51, 52 e 56.

GOOGLE. *Join SRE Classroom NALSD workshops*. 2018. Disponível em: <https://cloud.google.com/blog/products/devops-sre/join-sre-classroom-nalsd-workshops/>. Citado 3 vezes nas páginas 46, 47 e 52.

GOOGLE. *Join SRE Classroom NALSD workshops*. 2020. Disponível em: <https://cloud.google.com/blog/products/devops-sre/join-sre-classroom-nalsd-workshops/>. Citado 2 vezes nas páginas 33 e 42.

GOOGLE. *Planilhas Google*. 2021. Disponível em: <https://accounts.google.com/ServiceLogin?service=wise&passive=1209600&continue=https://docs.google.com/spreadsheets/u/0/&followup=https://docs.google.com/spreadsheets/u/0/&ltmpl=sheets>. Citado na página 34.

GRAçA, H. *DDD, Hexagonal, Onion, Clean, CQRS, ... How I put it all together*. 2017. Disponível em: <https://medium.com/the-software-architecture-chronicles/ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together-f2590c0aa7f6>. Citado 2 vezes nas páginas 46 e 47.

HAMID, A. *Microservices Architecture - Bangkok 2018*. 2018. Disponível em: <https://www.slideshare.net/arafkarsh/microservices-architecture-bangkok-2018>. Citado 2 vezes nas páginas 46 e 47.

HAMID, A. *Domain Driven Design*. 2020. Disponível em: <https://www.slideshare.net/arafkarsh/domain-driven-design-238299677>. Citado 3 vezes nas páginas 46, 47 e 52.

HAMID, A. *Microservices Architecture - Cloud Native Apps*. 2021. Disponível em: <https://www.slideshare.net/arafkarsh/microservices-architecture-cloud-native-apps?qid=50dc9a1e-4c81-415b-a74b-ec849010545f&v=&b=&from_search=14>. Citado 3 vezes nas páginas 46, 47 e 56.

HAQ, F. u. *Top 10 System Design Interview Questions for Software Engineers*. 2019. Disponível em: <https://medium.com/hackernoon/top-10-system-design-interview-questions-for-software-engineers-8561290f0444>. Citado 4 vezes nas páginas 46, 47, 53 e 55.

HOSTEDBYCONFLUENT. *Reusing Kafka Data Structure Between Projects | Laura Schornack and M...* 2021. Disponível em: <https://www.slideshare.net/HostedbyConfluent/reusing-kafka-data-structure-between-projects-laura-schornack-and-maureen-penzenik-northern-trust?qid=190636e5-9344-4fe2-9edf-b5e86a682ff4&v=&b=&from_search=35>. Citado na página 46.

HUSEYNLI, O. *Software Architecture Patterns: 5 min read*. 2021. Disponível em: <https://orkhanscience.medium.com/software-architecture-patterns-5-mins-read-e9e3c8eb47d2>. Citado 2 vezes nas páginas 46 e 47.

IBM. *Use domain-driven design to architect your cloud apps*. 2018. Disponível em: <https://developer.ibm.com/tutorials/cl-domain-driven-design-event-sourcing/>. Citado 5 vezes nas páginas 46, 47, 51, 52 e 56.

IBM, I. C. E. *SOA vs. Microservices: What's the Difference?* 2020. Publisher:. Disponível em: <https://www.ibm.com/cloud/blog/soa-vs-microservices>. Citado 8 vezes nas páginas 7, 22, 23, 42, 46, 47, 51 e 56.

IBM, I. C. E. *soa*. 2021. Publisher:. Disponível em: <https://www.ibm.com/cloud/learn/soa>. Citado 4 vezes nas páginas 21, 22, 46 e 47.

IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. In: . [S.l.: s.n.], 2000. Citado na página 17.

KALKMAN, P. *How To Enhance Your Software Architecture Design*. 2021. Disponível em: <https://betterprogramming.pub/how-to-enhance-your-software-architecture-design-58668c3a5670>. Citado na página 46.

KAO, K. *DDD Taiwan Community 2019 01-26-1st-meetup-why ddd matters*. 2019. Disponível em: <https://www.slideshare.net/kimKao/ddd-taiwan-community-2019-01261stmeetupwhy-ddd-matters?qid=0acdf448-054f-4c32-ba9f-31a7a93dda31&v=&b=&from_search=24>. Citado 2 vezes nas páginas 46 e 47.

KUBERNETES. 2021. Disponível em: <https://kubernetes.io/pt-br/>. Citado na página 60.

LAINE, C. ***ME-11 Domain-Driven Design in the era of Microservices*. 2019. Disponível em: <https://medium.com/it-dead-inside/domain-driven-design-in-the-era-of-icroservices-de2be01821ed>. Citado 4 vezes nas páginas 46, 47, 51 e 56.

LAINE, C. *Domain-Driven Design: Things to Remember When Building a Bounded Context*. 2020. Disponível em: <https://medium.com/it-dead-inside/

domain-driven-design-things-to-remember-when-building-a-bounded-context-62ed1d0cb2ae>. Citado na página 46.

LELONEK, K. *DDD building blocks*. 2015. Disponível em: <https://blog.lelonek.me/ddd-building-blocks-for-ruby-developers-cdc6c25a80d2>. Citado na página 46.

LUGAVERE, B. *Micro-Service Design Pattern: Dependency Driven Decomposition (DDD)*. 2019. Disponível em: <https://medium.com/swlh/micro-service-design-pattern-dependency-driven-decomposition-ddd-d2d28df2fedc>. Citado 2 vezes nas páginas 46 e 47.

LüDTKE, D. *My Path to Site Reliability Management*. 2019. Disponível em: <https://danrl.com/srm/>. Citado na página 42.

MARTINEZ, P. *Domain-Driven Design: Everything You Always Wanted to Know About it, But Were Afraid to Ask*. 2021. Disponível em: <https://medium.com/ssense-tech/domain-driven-design-everything-you-always-wanted-to-know-about-it-but-were-afraid-to-ask-a85e7 Citado 7 vezes nas páginas 46, 47, 51, 52, 53, 56 e 59.

MARTRAIRE, C. ****SS-09 DDD patterns that were not in the book*. 2017. Disponível em: <https://www.slideshare.net/cyriux/ddd-patterns-that-were-not-in-the-book>. Citado 4 vezes nas páginas 46, 47, 52 e 59.

MCCASLIN, M. L. The nature of leadership within rural communities: A grounded theory. *ETD collection for University of Nebraska - Lincoln*, p. 1–233, jan. 1993. Disponível em: <https://digitalcommons.unl.edu/dissertations/AAI9415981>. Citado na página 35.

MENEZES, G. *Usando DDD e Event Storming para Remodelar o Domínio de um Módulo Legado*. 2017. Disponível em: <https://medium.com/involves-rocks/usando-ddd-e-event-storming-para-remodelar-o-domínio-de-um-módulo-legado-aa12a364616e>. Citado 3 vezes nas páginas 46, 47 e 58.

MONSON-HAEFEL, R. (Ed.). *97 things every software architect should know: collective wisdom from the experts*. Sebastopol, Calif. ; Farnham: O'Reilly Media, 2009. OCLC: ocn263978531. ISBN 9780596522698. Citado na página 17.

NIKLAS, K. *Modern Software Architecture (#1): Domain Driven Design*. 2018. Disponível em: <https://medium.com/modern-software-architecture/modern-software-architecture-1-domain-driven-design-f06fad8695f9>. Citado 2 vezes nas páginas 46 e 47.

NILSSON, J. *Applying domain-driven design and patterns: with examples in C# and .NET*. Upper Saddle River, NJ: Addison-Wesley, 2006. OCLC: ocm63703904. ISBN 9780321268204. Citado na página 18.

Node.js. *Node.js*. 2021. Disponível em: <https://nodejs.org/en/>. Citado na página 60.

NORELUS, E. *Implementing Domain-Driven Design for Microservice Architecture*. 2019. Disponível em: <https://medium.com/design-and-tech-co/implementing-domain-driven-design-for-microservice-architecture-26eb0333d72e>. Citado 3 vezes nas páginas 46, 47 e 53.

OZUNLU, A. *Domain Driven Design (DDD)*. 2018. Disponível em: <https://medium.com/@avniozunlu/domain-driven-design-ddd-151c90472914>. Citado na página 46.

QUORA. *How is software architecture designed in an agile process?* 2015. Disponível em: <https://www.quora.com/How-is-software-architecture-designed-in-an-agile-process>. Citado na página 46.

QUORA. *How is software architecture related to the design and development approaches? - Quora*. 2015. Disponível em: <https://www.quora.com/How-is-software-architecture-related-to-the-design-and-development-approaches>. Citado na página 46.

QUORA. *Is domain driven design really worth it? Or is it just to create buzz?* 2016. Disponível em: <https://www.quora.com/Is-domain-driven-design-really-worth-it-Or-is-it-just-to-create-buzz>. Citado 2 vezes nas páginas 46 e 47.

QUORA. *What are your thoughts on the future of Software Architecture/Design?* 2016. Disponível em: <https://www.quora.com/What-are-your-thoughts-on-the-future-of-Software-Architecture-Design>. Citado na página 46.

QUORA. *What basic things need to know about design software architecture?* 2016. Disponível em: <https://www.quora.com/What-basic-things-need-to-know-about-design-software-architecture>. Citado na página 46.

QUORA. *What is software architecture in architectural design?* 2016. Disponível em: <https://www.quora.com/What-is-software-architecture-in-architectural-design>. Citado na página 46.

QUORA. *What is the relationship between microservices and domain-driven design?* 2016. Disponível em: <https://www.quora.com/What-is-the-relationship-between-microservices-and-domain-driven-design>. Citado 2 vezes nas páginas 46 e 47.

QUORA. *How do microservices make use of domain-driven design?* 2017. Disponível em: <https://www.quora.com/How-do-microservices-make-use-of-domain-driven-design>. Citado na página 46.

QUORA. *How to learn and develop skills required for software architectural design - Quora*. 2017. Disponível em: <https://www.quora.com/How-do-you-learn-and-develop-skills-required-for-software-architectural-design>. Citado na página 46.

QUORA. *QR-10 Why is design so difficult in software architecture?* 2017. Disponível em: <https://www.quora.com/Why-is-design-so-difficult-in-software-architecture>. Citado 3 vezes nas páginas 46, 47 e 52.

QUORA. *Where can I get advice about software architecture design?* 2017. Disponível em: <https://www.quora.com/

Where-can-I-get-advice-about-software-architecture-design>. Citado na página 46.

QUORA. *How do you learn and develop skills required for software architectural design?* 2018. Disponível em: <https://www.quora.com/How-do-you-learn-and-develop-skills-required-for-software-architectural-design>. Citado 2 vezes nas páginas 46 e 47.

QUORA. *How relevant is domain-driven design?* 2018. Disponível em: <https://www.quora.com/How-relevant-is-domain-driven-design>. Citado na página 46.

QUORA. *Is Domain Driven Design by Eric Evans worth reading?* 2018. Disponível em: <https://www.quora.com/Is-Domain-Driven-Design-by-Eric-Evans-worth-reading>. Citado na página 46.

QUORA. *Is it possible to carry out a large software project without a software architect?* 2018. Disponível em: <https://www.quora.com/Is-it-possible-to-carry-out-a-large-software-project-without-a-software-architect>. Citado na página 46.

QUORA. *Is it possible to carry out a large software project without a software architect? - Quora.* 2018. Disponível em: <https://www.quora.com/Is-it-possible-to-carry-out-a-large-software-project-without-a-software-architect>. Citado na página 15.

QUORA. *Where can you find a forum for discussing software architecture and design?* 2018. Disponível em: <https://www.quora.com/Where-can-you-find-a-forum-for-discussing-software-architecture-and-design>. Citado na página 46.

QUORA. *Why is architectural design important in software engineering?* 2019. Disponível em: <https://www.quora.com/Why-is-architectural-design-important-in-software-engineering>. Citado 2 vezes nas páginas 46 e 47.

QUORA. *Is domain driven design worth it?* 2020. Disponível em: <https://www.quora.com/Is-domain-driven-design-worth-it>. Citado na página 46.

QUORA. *Is domain-driven design still relevant?* 2021. Disponível em: <https://www.quora.com/Is-domain-driven-design-still-relevant>. Citado 2 vezes nas páginas 46 e 47.

RICHARDSON. *Health Check*. 2021. Disponível em: <https://microservices.io/patterns/observability/health-check-api.html>. Citado na página 46.

RICHARDSON, C. *Microservices Pattern: Self-contained service*. Disponível em: <http://microservices.io/patterns/decomposition/self-contained-service.html>. Citado na página 46.

RICHARDSON, C. *A Pattern Language for Microservices*. 2017. Disponível em: <https://www.slideshare.net/chris.e.richardson/a-pattern-language-for-microservices>. Citado 4 vezes nas páginas 46, 47, 51 e 53.

RICHARDSON, C. *Domain event*. 2021. Disponível em: <https://microservices.io/patterns/data/domain-event.html>. Citado na página 46.

RICHARDSON, C. *Microservices Pattern: API gateway pattern*. 2021. Disponível em: <http://microservices.io/patterns/apigateway.html>. Citado na página 46.

RICHARDSON, C. *Microservices Pattern: Decompose by business capability*. 2021. Disponível em: <http://microservices.io/patterns/decomposition/decompose-by-business-capability.html>. Citado 4 vezes nas páginas 46, 47, 51 e 53.

RICHARDSON, C. *Microservices Pattern: Monolithic Architecture pattern*. 2021. Disponível em: <http://microservices.io/patterns/monolithic.html>. Citado 4 vezes nas páginas 46, 47, 51 e 52.

RICHARDSON, C. *RC-05 Decompose by subdomain*. 2021. Disponível em: <https://microservices.io/patterns/decomposition/decompose-by-subdomain.html>. Citado 4 vezes nas páginas 46, 47, 51 e 53.

RICHARDSON, C. *RC-09 Circuit Breaker*. 2021. Disponível em: <https://microservices.io/patterns/reliability/circuit-breaker.html>. Citado 3 vezes nas páginas 7, 46 e 57.

RICHARDSON, C. *Service per team*. 2021. Disponível em: <https://microservices.io/patterns/decomposition/service-per-team.html>. Citado 2 vezes nas páginas 46 e 47.

RICHARDSON, C. *Strangler application*. 2021. Disponível em: <https://microservices.io/patterns/refactoring/strangler-application.html>. Citado 2 vezes nas páginas 46 e 47.

RICHARDSON, C. *What are microservices?* 2021. Disponível em: <http://microservices.io/index.html>. Citado na página 22.

RICHARDSON, C. *What are microservices?* 2021. Disponível em: <http://microservices.io/index.html>. Citado 3 vezes nas páginas 46, 47 e 52.

SAMOKHIN, V. *DDD Strategic Patterns: How To Define Bounded Contexts*. 2017. Disponível em: <https://codeburst.io/ddd-strategic-patterns-how-to-define-bounded-contexts-2dc70927976e>. Citado 5 vezes nas páginas 46, 47, 51, 52 e 53.

SCOTT, K. W.; HOWELL, D. Clarifying Analysis and Interpretation in Grounded Theory: Using a Conditional Relationship Guide and Reflective Coding Matrix. *International Journal of Qualitative Methods*, v. 7, n. 2, p. 1–15, jun. 2008. ISSN 1609-4069. Disponível em: <https://doi.org/10.1177/160940690800700201>. Citado na página 45.

SOFTWAREONE. *Saiba o que é a automatização de deploy*. 2020. Disponível em: <https://www.softwareone.com/pt-br/blog/artigos/2020/02/03/saiba-o-que-e-a-automatizacao-de-deploy>. Citado na página 15.

SOMMERVILLE, I. et al. Large-scale complex it systems. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 55, n. 7, p. 71–77, jul. 2012. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/2209249.2209268>. Citado 2 vezes nas páginas 23 e 24.

SOYKAN, O. *****SS-10 Domain Driven Design(DDD) Presentation*. 2017. Disponível em: <https://www.slideshare.net/OuzhanSoykan/domain-driven-designddd-presentation?qid=0acdf448-054f-4c32-ba9f-31a7a93dda31&v=&b=&from_search=45>. Citado 4 vezes nas páginas 46, 47, 52 e 59.

STOL, K.-J.; RALPH, P.; FITZGERALD, B. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2016. p. 120–131. ISSN: 1558-1225. Citado 3 vezes nas páginas 16, 32 e 36.

SVIRCA, Z. *Everything you need to know about MVC architecture*. 2020. Disponível em: <https://towardsdatascience.com/everything-you-need-to-know-about-mvc-architecture-3c827930b4c1>. Citado na página 20.

THAWARE, S. G. *Software architecture and software design*. 2015. Disponível em: <https://www.slideshare.net/Swapnilthaware1/swap-sda-copy?qid=e55a7011-8900-49dc-a305-246607889a1e&v=&b=&from_search=3>. Citado na página 46.

TUNE, N. *Uncovering Hidden Business Rules with DDD Aggregates*. 2019. Disponível em: <https://medium.com/nick-tune-tech-strategy-blog/uncovering-hidden-business-rules-with-ddd-aggregates-67fb02abc4b>. Citado 5 vezes nas páginas 46, 47, 53, 56 e 58.

TUNE, N. *Legacy Architecture Modernisation With Strategic Domain-Driven Design*. 2020. Disponível em: <https://medium.com/nick-tune-tech-strategy-blog/legacy-architecture-modernisation-with-strategic-domain-driven-design-3e7c05bb383f>. Citado 2 vezes nas páginas 46 e 47.

TUNE, N. *Domain, Subdomain, Bounded Context, Problem/Solution Space in DDD: Clearly Defined*. 2021. Disponível em: <https://medium.com/nick-tune-tech-strategy-blog/domains-subdomain-problem-solution-space-in-ddd-clearly-defined-e0b49c7b586c>. Citado 6 vezes nas páginas 46, 47, 52, 53, 57 e 59.

UFPE, C. de Informática da. *Função: Arquiteto de software*. 2006. Disponível em: <https://www.cin.ufpe.br/~gta/rup-vc/core.base_rup/roles/rup_software_architect_DB0BF177.html>. Citado na página 17.

UNIVERSITY, C. M. *Software architecture*. 2017. Disponível em: <https://www.sei.cmu.edu/our-work/projects/display.cfm?customel_datapageid_4050=21328>. Citado na página 17.

VALIPOUR, M. H. et al. *A Brief Survey of Software Architecture Concepts and Service Oriented Architecture*. [S.l.: s.n.], 2009. Citado na página 21.

VEGREVILLE, B. *Expressive error handling in TypeScript and benefits for domain-driven design*. 2019. Disponível em: <https://medium.com/inato/expressive-error-handling-in-typescript-and-benefits-for-domain-driven-design-70726e061c86>. Citado na página 46.

VERNON, V. *Implementing domain-driven design*. Upper Saddle River, NJ: Addision-Wesley, 2013. OCLC: ocn830008330. ISBN 9780321834577. Citado na página 42.

WEN, M. et al. Understanding FLOSS through community publications: strategies for grey literature review. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. Seoul, South Korea: Association for Computing Machinery, 2020. (ICSE-NIER '20), p. 89–92. ISBN 9781450371261. Disponível em: <https://doi.org/10.1145/3377816.3381729>. Citado 3 vezes nas páginas 16, 28 e 29.

WILSENACH, R. *DevOpsCulture*. 2015. Disponível em: <https://martinfowler.com/bliki/DevOpsCulture.html>. Citado na página 42.

ZHU, R. *GraphQL and DDD: the Missing Link*. 2017. Disponível em: <https://medium.com/hackernoon/graphql-and-ddd-the-missing-link-4e992a26b711>. Citado na página 46.