CEFET/RJ - CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA
CELSO SUCKOW DA FONSECA

O Título do Seu Trabalho Bem Bonito

Seu Nome Completo

Prof. Orientador:
Prof. D.Sc. Carlos Eduardo Pantoja//Prof. Se
Houver

Rio de Janeiro,
Junho de 2025

CEFET/RJ - CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA
CELSO SUCKOW DA FONSECA

O Título do Seu Trabalho Bem Bonito

Seu Nome Completo

Projeto final apresentado em cumprimento às normas do Departamento de Educação Superior do Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, CEFET/RJ, como parte dos requisitos para obtenção do título de Bacharel em Sistemas de Informação.

Prof. Orientador:
Prof. D.Sc. Carlos Eduardo Pantoja//Prof. Se Houver

Rio de Janeiro,
Junho de 2025

Obtenha a ficha catalográfica junto a biblioteca.
Substitua o arquivo ficha.pdf pela versão obtida lá.

PEDRO HENRIQUE DA COSTA CANTANHÊDE

Desenvolvimento de Biblioteca Front-end para Acessibilidade em Aplicações Web

Trabalho de Conclusão de Curso, apresentado ao Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, como parte das exigências para a obtenção do título de Bacharel em Sistemas de Informação.

Rio de Janeiro, 13 de fevereiro de 2025.

BANCA EXAMINADORA

_____
Prof. M.Sc. Diego Cardoso Borda Castro
Centro Federal de Educação Tecnológica (CEFET/RJ)

_____
Prof. D.Sc. Leonardo Pio Vasconcelos
Universidade Federal Fluminense

_____
Prof. D.Sc. Carlos Eduardo Pantoja
Centro Federal de Educação Tecnológica (CEFET/RJ)

"Não importa como você bate e sim o quanto aguenta apanhar e continuar lutando; o quanto pode suportar e seguir em frente. É assim que se ganha."

(Rocky Balboa)

RESUMO

O resumo é uma parte essencial do TCC, pois apresenta de forma concisa os principais pontos do trabalho. Deve ser escrito em um único parágrafo, com aproximadamente 150 a 500 palavras, e deve conter as seguintes informações: Primeiramente, é importante apresentar o tema e o contexto do trabalho, explicando brevemente em que área ele se insere e qual problema ou motivação o justifica. Em seguida, o aluno deve deixar claro qual é o objetivo principal da pesquisa ou do projeto desenvolvido. Depois, deve-se descrever, de forma sucinta, a metodologia adotada — ou seja, quais foram os métodos, ferramentas, técnicas ou etapas utilizadas para atingir o objetivo proposto. O resumo também deve apresentar os principais resultados obtidos, ainda que sejam parciais, destacando as evidências mais relevantes. Por fim, deve-se incluir a conclusão principal do trabalho, ressaltando sua contribuição ou impacto. O resumo deve ser informativo, escrito em linguagem clara e objetiva, sem o uso de citações, siglas não definidas ou fórmulas matemáticas. Ele deve ser compreensível por qualquer leitor, mesmo que não tenha acesso ao restante do trabalho. Ao final do resumo, o aluno deve indicar cinco palavras-chave, separadas por ponto e vírgula, que representem os principais temas abordados no TCC.

Palavras-chaves: Palavra-chave 1; Palavra-chave 2; Palavra-chave 3; Palavra-chave 4; Palavra-chave 5

ABSTRACT

O mesmo conteúdo do Resumo mas em Inglês.

Keywords: Keyword 1; Keyword 2; Keyword 3; Keyword 4; Keyword 5

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A contextualização é a primeira parte da introdução e tem como objetivo situar o leitor sobre o tema do trabalho. É aqui que o aluno deve apresentar o cenário geral da área abordada, explicando os principais conceitos, tendências, desafios ou avanços relacionados ao tema escolhido.

Deve-se começar com uma visão ampla, destacando aspectos relevantes da área de estudo e, gradualmente, direcionar o texto até chegar ao foco específico do trabalho. O aluno pode mencionar dados estatísticos, referências teóricas, aplicações práticas ou acontecimentos recentes que justifiquem a importância do tema. A ideia é responder à pergunta: por que este tema é relevante hoje?

Além disso, é importante destacar se o tema é atual, inovador, pouco explorado ou se existe alguma lacuna ou demanda prática que ainda não foi suficientemente atendida. Sempre que possível, a contextualização deve ser fundamentada com referências bibliográficas confiáveis, preferencialmente recentes e pertinentes à área.

Essa parte deve gerar no leitor interesse pelo trabalho e preparar o terreno para a apresentação do problema de pesquisa na seção seguinte.

1.1 Problema

Nesta seção, o aluno deve descrever o problema central que motivou o desenvolvimento do trabalho. Isso envolve contextualizar a situação, mostrar lacunas existentes na área ou desafios enfrentados, e justificar a importância de tratar essa problemática. A redação deve ser objetiva, baseada em fatos e, sempre que possível, sustentada por referências bibliográficas.

Aqui devem ser formuladas uma ou mais perguntas que o trabalho busca responder. As questões de pesquisa ajudam a delimitar o foco do estudo e orientam a construção dos objetivos. Elas devem ser claras, específicas e relacionadas ao problema descrito anteriormente.

QP 1: Colocar as questões de pesquisa de forma clara e direta?

1.2 Objetivo

Esta seção apresenta o objetivo geral do trabalho, ou seja, a finalidade principal que se pretende alcançar. O objetivo deve estar diretamente ligado ao problema identificado e às questões de pesquisa. A redação deve começar com verbos no infinitivo, como desenvolver, analisar, avaliar, propor, entre outros.

1.2.1 Objetivos Específicos

Os objetivos específicos detalham as etapas intermediárias ou metas que serão cumpridas para atingir o objetivo geral. Cada objetivo específico deve ser claro, mensurável e realizável dentro do escopo do trabalho.

1. **O primeiro objetivo** e uma breve descrição.

1.3 Metodologia

Neste item, o aluno deve descrever o caminho metodológico adotado no trabalho. Isso inclui os métodos de pesquisa (qualitativa, quantitativa, experimental etc.), técnicas utilizadas (levantamento bibliográfico, estudo de caso, modelagem, implementação, testes etc.), ferramentas empregadas (softwares, linguagens, frameworks), e o delineamento geral do processo. O nível de detalhamento deve ser suficiente para que outro pesquisador possa compreender e, eventualmente, replicar o estudo.

1.4 Contribuições

Nesta seção, o aluno deve explicitar quais são as principais contribuições do trabalho. Isso pode incluir avanços teóricos, práticos ou tecnológicos; o desenvolvimento de uma ferramenta; uma nova abordagem metodológica; ou a solução de um problema específico. A ênfase deve ser no valor agregado pelo trabalho dentro de seu contexto acadêmico ou científico.

1.5 Estrutura do trabalho

Aqui se apresenta a organização dos capítulos do TCC. Deve-se descrever o que é abordado em cada capítulo, dando ao leitor uma visão geral do conteúdo e da lógica do documento

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta os principais conceitos, práticas e tecnologias que sustentam a construção de sistemas distribuídos resilientes baseados em microsserviços. São abordadas as diferentes abordagens arquiteturais, bem como os mecanismos e ferramentas essenciais para garantir a escalabilidade, a robustez e a confiabilidade dessas aplicações. Essa fundamentação destaca as relações de dependência entre essas tecnologias e os trade-offs envolvidos em sua aplicação, estabelecendo assim a base conceitual e técnica para o desenvolvimento e validação do modelo de maturidade proposto.

2.1 Arquitetura de Software

A arquitetura de software ocupa um papel estratégico na engenharia de software, pois influencia diretamente decisões de design, desempenho, manutenibilidade e escalabilidade ao longo do ciclo de vida de um sistema. De acordo com a norma ANSI/IEEE 1471-2000 [2007], a arquitetura de um sistema é definida como sua organização fundamental, incluindo componentes, interações entre eles e princípios que orientam seu projeto e evolução. Essa compreensão vai além de aspectos técnicos, pois a arquitetura também reflete fatores organizacionais e práticas de desenvolvimento.

2.1.1 Arquitetura Monolítica

Historicamente, a Arquitetura Monolítica surgiu como o modelo dominante para o desenvolvimento de aplicações. Nesse modelo, o aplicativo é construído como um único projeto, utilizando um único sistema de compilação e resultando em um binário executável que reúne diversos módulos voltados a recursos técnicos ou de negócios [Mendes, 2021]. Essa abordagem se consolidou pela sua simplicidade inicial: ao manter todas as funcionalidades em um único processo, as equipes têm menos preocupações com a comunicação entre módulos e podem focar na entrega de funcionalidades de forma rápida [Pereira and Diniz, 2022].

A Figura 1 ilustra como essa arquitetura agrupa a interface do usuário, o núcleo da aplicação e o banco de dados em um único bloco coeso. Esse modelo reduz a complexidade inicial de integração e facilita o monitoramento, já que todas as operações são executadas dentro do mesmo ambiente [Salaheddin and Ahmed, 2022].

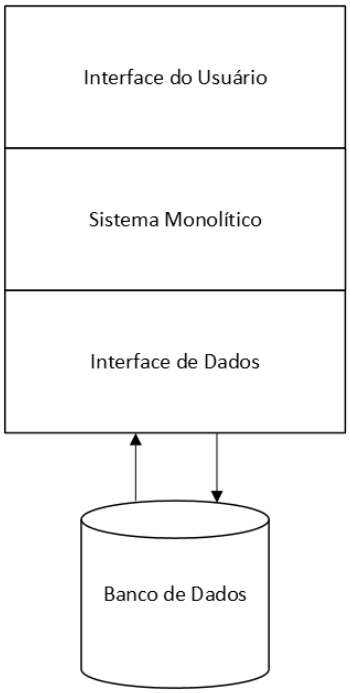

Figura 1: Diagrama representativo de uma arquitetura monolítica.

Contudo, segundo [Fowler, 2015], embora a arquitetura monolítica seja um excelente ponto de partida, ela apresenta desafios à medida que o sistema cresce e se torna mais complexo. Problemas de escalabilidade e pontos únicos de falha surgem quando um único binário precisa atender a múltiplas demandas ao mesmo tempo. Isso gera a necessidade de soluções mais flexíveis e adaptáveis para lidar com a evolução constante das aplicações e das demandas do mercado.

2.1.2 Arquitetura de Microsserviços

É nesse cenário de evolução e necessidade de adaptação que a Arquitetura de Microsserviços ganha destaque. Essa abordagem visa superar as limitações do monólito, dividindo a aplicação em pequenos serviços autônomos e independentes, cada um responsável por uma funcionalidade específica e executando seu próprio processo [Fowler and Lewis, 2014]. Esses microsserviços se comunicam entre si por meio de APIs (Interfaces de Programação de Aplica-

ções) baseadas em HTTP (Hypertext Transfer Protocol), formando um ecossistema distribuído e colaborativo.

A Figura 2 ilustra essa transição arquitetural: cada retângulo representa um microsserviço, com seu próprio banco de dados isolado, mostrando como a separação de responsabilidades facilita o desenvolvimento e a manutenção. A interface do usuário atua como ponto de entrada, direcionando requisições aos microsserviços especializados.

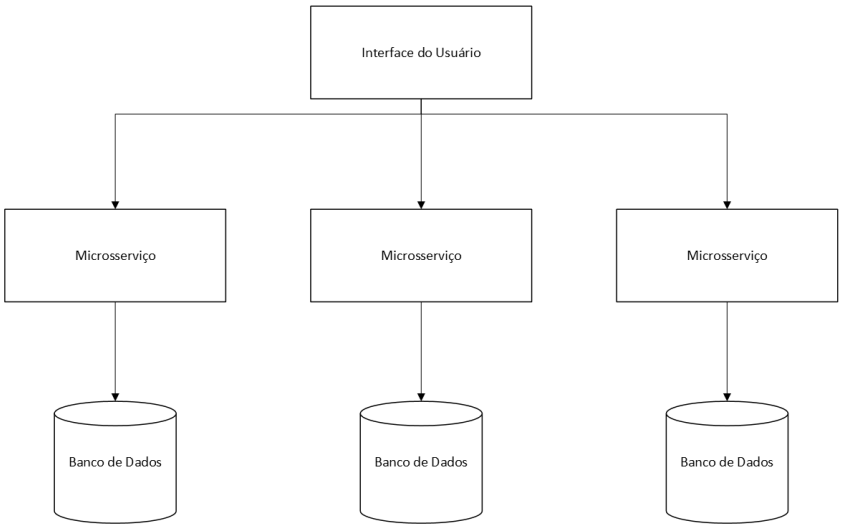

Figura 2: Diagrama de microsserviços com interface, serviços e bancos de dados independentes.

Essa descentralização traz ganhos significativos: cada serviço pode ser escalado horizontalmente de forma independente, adaptando-se às demandas específicas de carga sem impactar outros serviços. Isso aumenta a flexibilidade e a resiliência do sistema como um todo. No entanto, também introduz novos desafios, como a necessidade de ferramentas de observabilidade para monitorar métricas, logs e verificações de saúde de cada serviço [Newman, 2015]. Além disso, migrar de um sistema monolítico para uma arquitetura distribuída exige estratégias cuidadosas de orquestração e coreografia, garantindo que os serviços colaborem de forma eficiente e confiável [Baskarada et al., 2018].

Assim, a passagem de uma arquitetura monolítica para microsserviços não é apenas uma mudança técnica, mas também um reflexo de como as organizações se adaptam à complexidade e à necessidade de resiliência em sistemas distribuídos modernos. Esse movimento de evolução arquitetural fundamenta o modelo de maturidade proposto neste trabalho, que visa apoiar organizações na transição para ambientes mais flexíveis e robustos, aproveitando os benefícios dos microsserviços sem ignorar os desafios operacionais que acompanham essa jornada.

2.1.3 Arquitetura Orientada a Eventos

A necessidade de lidar com cenários de alta demanda e picos de acesso, bem como de garantir maior adaptabilidade, impulsionou o surgimento de abordagens que complementam a arquitetura de microsserviços. Nesse contexto, a Arquitetura Orientada a Eventos (EDA, do inglês *Event-Driven Architecture*) desponta como uma solução poderosa para desacoplar componentes e permitir que eles reajam a mudanças e estímulos de forma assíncrona [Michelson, 2006].

Na EDA, eventos representam ocorrências significativas no domínio do negócio ou do sistema, indicando mudanças de estado, anomalias ou oportunidades. Esses eventos são capturados e processados por componentes modulares e independentes, que executam tarefas específicas de maneira desacoplada e assíncrona. Essa característica fortalece a escalabilidade e a adaptabilidade do sistema, pois cada componente atua de forma isolada, sem precisar conhecer os demais ou saber quem irá reagir aos eventos gerados [Richards, 2015; Lazzari and Farias, 2021].

A Figura 2 — que ilustra a arquitetura de microsserviços — ajuda a entender como a EDA complementa essa abordagem. Enquanto os microsserviços são responsáveis por funcionalidades bem definidas, a EDA permite que esses serviços interajam de forma reativa, respondendo a eventos que circulam entre eles. Esse modelo reduz ainda mais o acoplamento e aumenta a flexibilidade operacional, criando fluxos de processamento mais dinâmicos e resilientes.

A EDA pode ser implementada seguindo duas topologias principais:

- **Mediator Topology:** utiliza um mediador central para orquestrar eventos complexos com múltiplos passos[1]. É ideal para fluxos que exigem coordenação explícita [Richards, 2015].

- **Broker Topology:** dispensa o mediador e conecta processadores de eventos diretamente por meio de um *broker*. Cada processador atua de forma autônoma, consumindo eventos, executando lógicas específicas e publicando novos eventos, o que favorece a escalabilidade e a flexibilidade [Richards, 2015]. Um exemplo clássico é o registro de estatísticas e

[1] O termo "Mediator"também é conhecido como um padrão de projeto do GoF (Gang of Four), descrito no livro *Design Patterns: Elements of Reusable Object-Oriented Software* (1994). Embora compartilhem o princípio de centralizar a comunicação para evitar acoplamento direto, no contexto da EDA o Mediator atua em nível arquitetural, coordenando eventos em sistemas distribuídos, enquanto no padrão GoF atua em nível de objetos dentro de um único processo.

recomendações em plataformas de streaming de vídeo, onde diversos componentes atuam simultaneamente de maneira descentralizada.

Ambas as topologias eliminam dependências diretas entre os componentes, alinhando-se ao princípio de que "emissores não precisam conhecer os receptores" [Lazzari and Farias, 2021]. Essa separação entre quem emite e quem consome eventos fortalece a flexibilidade e a adaptabilidade da arquitetura, pois cada componente ou serviço pode evoluir, ser substituído ou escalado de maneira independente, sem impacto direto nos demais.

Além de facilitar a manutenção e a evolução dos sistemas, esse baixo acoplamento reduz os pontos de falha e aumenta a resiliência do ecossistema como um todo. Cada serviço ou processador de eventos tem a liberdade de reagir de acordo com sua lógica de negócios e com a disponibilidade de recursos, sem depender de uma sequência rígida de chamadas ou de um fluxo de controle centralizado. Assim, mesmo em situações adversas ou de alta demanda, os componentes continuam a operar de forma autônoma e coordenada, garantindo a continuidade dos serviços e a robustez do sistema.

Essa capacidade de resposta independente e dinâmica, proporcionada pela Arquitetura Orientada a Eventos, torna-se especialmente valiosa em ambientes distribuídos e complexos, onde a previsibilidade é limitada e a adaptabilidade é essencial para lidar com falhas e variações na carga de trabalho.

2.2 Escalabilidade Horizontal e Vertical

A necessidade de manter um desempenho estável em ambientes distribuídos exige que sistemas modernos incorporem mecanismos de escalabilidade adequados. Essa escalabilidade pode ser classificada em duas formas principais: horizontal e vertical.

A primeira, conhecida também como *scaling out*, consiste em adicionar novas instâncias ao sistema, distribuindo a carga entre diversos recursos computacionais. Já a escalabilidade vertical — ou *scaling up* — está relacionada ao aumento da capacidade de uma única instância, adicionando mais poder de CPU, memória ou armazenamento. Ambas as abordagens visam melhorar o desempenho e a capacidade de resposta dos sistemas, sendo a escolha entre elas dependente das características da aplicação e do ambiente de implantação [Blinowski et al., 2022].

2.3 Load Balancer

Conforme as aplicações evoluíram de sistemas monolíticos para microsserviços e arquiteturas orientadas a eventos, a necessidade de lidar com grandes volumes de requisições e garantir escalabilidade e desempenho contínuos se tornou ainda mais crítica. Nesse cenário, o balanceamento de carga surge como uma técnica fundamental para manter a disponibilidade e a confiabilidade dos sistemas distribuídos.

O balanceamento de carga consiste em distribuir de forma equilibrada o tráfego de requisições entre diferentes servidores, evitando que apenas um nó seja sobrecarregado enquanto outros permanecem subutilizados [Mohapatra et al., 2013]. Esse processo não apenas otimiza o uso de recursos e reduz o tempo de resposta, mas também reforça a resiliência geral da aplicação, permitindo que ela suporte picos de acesso e situações de alta demanda sem comprometer a experiência do usuário.

Na prática, o balanceador de carga costuma ser posicionado entre os usuários e o conjunto de servidores que executam as aplicações. Embora essa não seja uma configuração obrigatória, ela é amplamente adotada porque permite que todo o tráfego seja gerenciado de maneira centralizada e transparente, sem que o usuário perceba as complexidades envolvidas [Kumar Mondal et al., 2016]. Atuando como uma ponte de intermediação, o balanceador intercepta as requisições e as distribui para os servidores disponíveis, ajustando-se dinamicamente à carga de trabalho e ao estado de saúde de cada servidor [Bourke, 2001].

A Figura 3 ilustra de forma clara esse fluxo de comunicação em um sistema distribuído que utiliza um Load Balancer. O diagrama mostra os usuários, que são responsáveis por gerar as solicitações; o Load Balancer, que as recebe e decide qual servidor irá processá-las; e, finalmente, os três servidores (Server 1, Server 2 e Server 3), que atuam como nós de processamento dessas requisições. Essa representação evidencia a separação de responsabilidades e a função essencial do balanceador para manter o sistema equilibrado e funcional, mesmo diante de variações imprevisíveis na demanda.

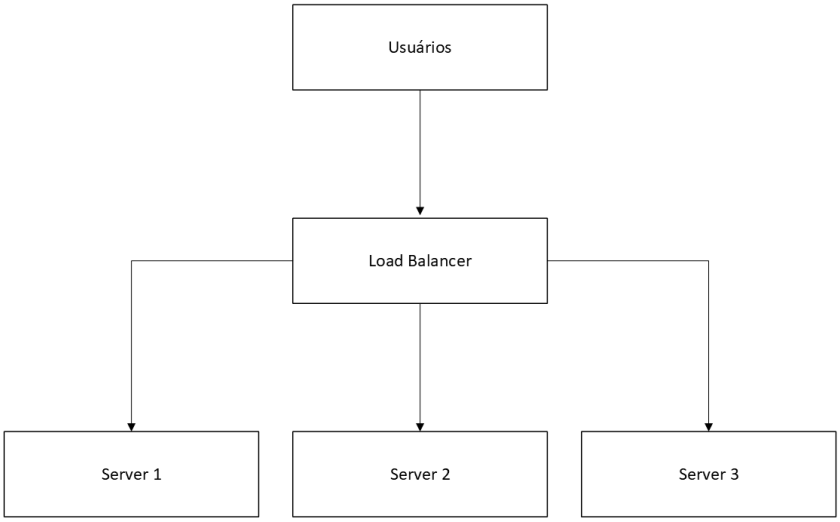

Figura 3: Diagrama de funcionamento de um Load Balancer, ilustrando a distribuição de requisições dos usuários para três servidores distintos.

Essa estratégia de distribuição eficiente permite o melhor aproveitamento dos recursos disponíveis, mantendo a performance e a disponibilidade dos serviços mesmo em momentos de pico de acesso. No entanto, apesar de seus benefícios, a implementação do balanceamento de carga também apresenta desafios técnicos e operacionais. Um dos principais problemas está relacionado à distribuição desigual de carga, causada, por exemplo, pelo armazenamento em cache dos registros DNS, que pode direcionar um volume excessivo de requisições para apenas um ou alguns servidores [Bourke, 2001]. Esse desequilíbrio pode gerar gargalos e prejudicar o desempenho do sistema como um todo.

Outro desafio recorrente é a necessidade de manter a persistência de sessão em aplicações que exigem consistência no atendimento ao usuário. Em sistemas de e-commerce, por exemplo, é fundamental que o usuário seja sempre redirecionado ao mesmo servidor durante uma sessão, o que aumenta a complexidade da configuração do balanceador e exige mecanismos adicionais de gerenciamento de sessão [Bourke, 2001].

Assim, o balanceamento de carga se estabelece como um elemento central no ecossistema de sistemas distribuídos modernos, complementando as estratégias de resiliência, escalabilidade e flexibilidade discutidas anteriormente.

2.4 Observabilidade em Sistemas Distribuídos

À medida que sistemas distribuídos se tornaram mais complexos, com múltiplos serviços atuando de forma autônoma e interdependente, cresceu também a necessidade de compreender

como esses componentes funcionam e interagem em tempo real. Nesse contexto, surge a observabilidade como um pilar essencial para manter a confiabilidade e o desempenho dos sistemas.

Originalmente definida por Kálmán (1960) na teoria de controle, a observabilidade descreve a capacidade de inferir o estado interno de um sistema a partir de suas saídas externas [Majors et al., 2022]. Aplicada à computação, essa ideia se adapta ao desafio de acompanhar aplicações formadas por microsserviços e arquiteturas orientadas a eventos, onde as falhas podem ser sutis e as causas nem sempre são evidentes.

A observabilidade se apoia em três pilares principais: logs, métricas e traces. Os logs são registros de eventos que ajudam a reconstruir o histórico de execução de cada serviço. As métricas fornecem dados quantitativos sobre o desempenho e a utilização de recursos ao longo do tempo, enquanto os traces mapeiam o caminho que as requisições percorrem entre diferentes serviços, expondo gargalos ou dependências inesperadas [Kuusijärvi, 2024].

Ao contrário do monitoramento tradicional, que se limita a acompanhar indicadores pré-definidos, a observabilidade permite responder a perguntas novas e inesperadas sobre o sistema. Essa flexibilidade é crucial para lidar com a complexidade e a incerteza de arquiteturas distribuídas modernas, onde pequenos problemas em um microsserviço podem desencadear efeitos em cascata por toda a aplicação.

Dessa forma, a observabilidade fecha o ciclo iniciado pela adoção de microsserviços e do balanceamento de carga. Se por um lado essas estratégias permitem maior escalabilidade e resiliência, por outro, tornam essencial a capacidade de entender o comportamento interno do sistema — não apenas quando ele está operando como esperado, mas também quando surgem falhas e degradações de desempenho. Essa integração entre arquitetura técnica e visibilidade operacional forma a base para a resiliência de sistemas distribuídos, permitindo que equipes de desenvolvimento e operação atuem com segurança e confiança, mesmo em ambientes dinâmicos e de alta demanda.

2.5 Padrões de Projeto

Ao longo da evolução dos sistemas de software, desenvolvedores perceberam que muitos problemas enfrentados em diferentes projetos se repetem de maneira recorrente. Foi assim que surgiram os padrões de projeto, que funcionam como microarquiteturas: soluções consolidadas e reutilizáveis que ajudam a resolver esses desafios de forma elegante e previsível, sem impor um modelo único e completo para toda a aplicação [Gamma et al., 1993].

Os padrões são, intencionalmente, pequenos e genéricos. Essa característica os torna adaptáveis a diversos domínios, garantindo que possam ser combinados ou modificados conforme a necessidade de cada projeto. Dessa forma, eles oferecem uma base sólida para a construção de sistemas mais coesos e confiáveis, independentemente das particularidades de cada negócio.

Um dos benefícios mais relevantes dos padrões de projeto é a capacidade de tornar o sistema mais resiliente a mudanças. À medida que novas funcionalidades surgem ou requisitos evoluem, padrões bem aplicados evitam a necessidade de grandes reformulações. Eles permitem que as partes críticas do sistema sejam ajustadas de forma mais controlada, reduzindo o risco de efeitos indesejados e mantendo a estabilidade da aplicação [Aversano et al., 2007].

Assim, os padrões de projeto vão muito além da simples reutilização de código: eles capturam boas práticas de design e fornecem um guia para decisões arquiteturais mais assertivas e eficientes. Funcionam como uma ponte entre o conhecimento consolidado da engenharia de software e as demandas concretas de um sistema em constante evolução, contribuindo para a criação de aplicações mais robustas, compreensíveis e preparadas para os desafios de longo prazo.

2.5.1 Retry

Em sistemas distribuídos, falhas de comunicação ou indisponibilidade temporária de serviços são inevitáveis. Para lidar com esses problemas e evitar que erros transitórios causem interrupções definitivas, surge o padrão *Retry*. Ele funciona como uma "segunda chance" para operações que falham, permitindo que o sistema faça novas tentativas de forma controlada antes de desistir completamente [Mendonça et al., 2020].

A Figura 4 ilustra o fluxo de comunicação em um sistema clássico sem o padrão *Retry*. Nesse cenário, o *ServicoCliente* envia requisições diretamente ao *ServicoAlvo*, recebendo respostas de sucesso (*OK*) ou falha (*erro*). Caso ocorra uma falha, a operação é imediatamente encerrada, comprometendo a continuidade do serviço.

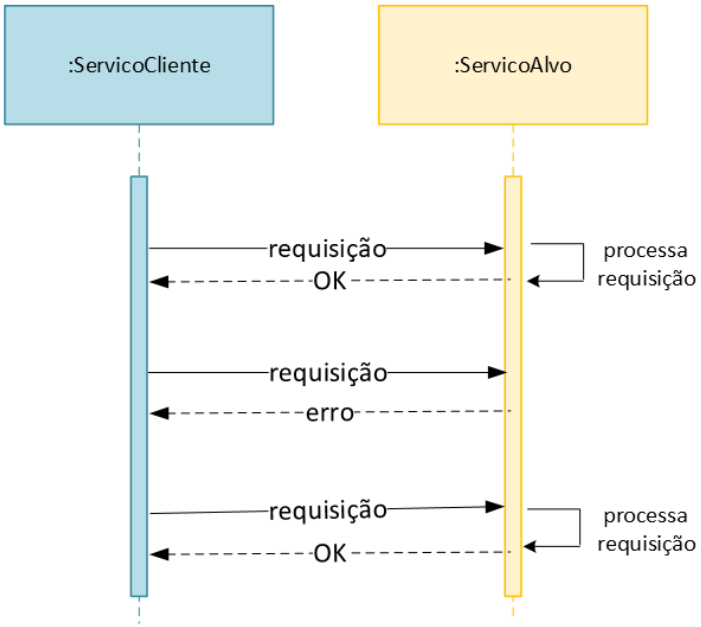

Figura 4: Fluxo de comunicação em um sistema clássico sem mecanismo de Retry [Mendonça et al., 2020].

A Figura 5 apresenta o fluxo quando o padrão *Retry* é introduzido. Nesse caso, um componente intermediário — o *MecanismoRetry* — gerencia as tentativas. Quando o *ServicoAlvo* retorna um erro, o *MecanismoRetry* realiza novas requisições de acordo com uma política pré-definida, que pode incluir intervalos de espera crescentes e estratégias de espera exponencial. Essa abordagem aumenta as chances de sucesso, especialmente em falhas temporárias, e evita que o sistema falhe definitivamente já na primeira tentativa.

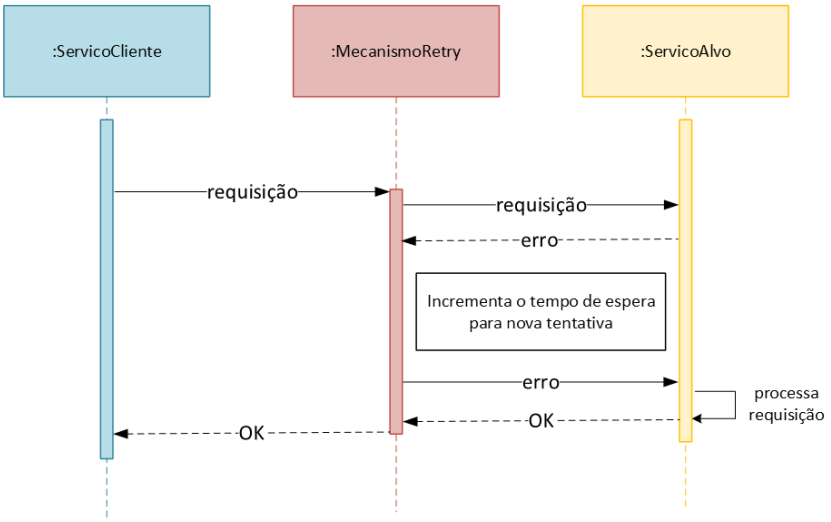

Figura 5: Fluxo de comunicação em um sistema com o padrão Retry [Mendonça et al., 2020].

Determinar o intervalo correto entre as tentativas de nova chamada é um ponto crítico. Um

intervalo muito curto pode levar à sobrecarga do serviço, enquanto um intervalo muito longo pode comprometer a responsividade do sistema [Kostenko, 2023]. Para lidar com esse desafio, adota-se a estratégia de espera exponencial.

Na espera exponencial, o tempo entre as tentativas aumenta de forma exponencial a cada falha consecutiva. Por exemplo, após a primeira falha, o sistema espera 1 segundo; após a segunda falha, espera 2 segundos; depois 4 segundos e assim por diante. Essa abordagem reduz a frequência das tentativas e permite que o serviço em falha tenha tempo para se recuperar, evitando sobrecargas desnecessárias.

Além disso, é comum aplicar um elemento de *jitter* — uma variação aleatória no tempo de espera — para evitar que múltiplos clientes sincronizem suas tentativas ao mesmo tempo, o que poderia criar um novo pico de carga. Dessa forma, ao adotar a espera exponencial com jitter, o padrão *Retry* promove sistemas distribuídos mais resilientes e capazes de se recuperar de falhas transitórias de forma controlada.

2.5.2 Circuit Breaker

Em aplicações modulares, nas quais diferentes serviços ou módulos colaboram para fornecer funcionalidades completas, podem surgir momentos em que um componente fique indisponível ou sobrecarregado. Esses cenários podem comprometer a estabilidade geral do sistema e afetar a experiência do usuário, exigindo estratégias para impedir que falhas pontuais se propaguem.

Nesse contexto, o padrão arquitetural *Circuit Breaker* atua como uma medida de proteção. Ele estabelece limites de controle — como a taxa de falhas tolerada ou o tempo máximo de resposta — para decidir quando bloquear temporariamente as requisições a um serviço com problemas [Kostenko, 2023]. Esses “limites pré-definidos” funcionam como barreiras automáticas que ajudam a evitar sobrecargas e permitem que o serviço falho tenha um período de recuperação antes de ser requisitado novamente [Suprapto et al., 2021].

Assim como um disjuntor elétrico corta o fluxo em caso de curto-circuito, o *Circuit Breaker* atua como um “intermediador” entre o cliente e o serviço remoto. Ele alterna entre três estados principais, descritos a seguir:

Estado	Descrição
Fechado (Closed)	Todas as requisições são encaminhadas ao serviço normalmente, até que a taxa de falhas ultrapasse o limite definido.
Aberto (Open)	O circuito bloqueia as requisições, enviando mensagens de erro ao cliente ou acionando um procedimento alternativo (fallback).
Meio-aberto (Half-open)	O circuito libera um número limitado de requisições de teste para verificar se o serviço voltou a responder corretamente. Se as tentativas forem bem-sucedidas, o circuito volta a fechado; caso contrário, reabre.

Tabela 1: Estados do padrão Circuit Breaker [Suprapto et al., 2021]

Visualmente, o *Circuit Breaker* pode ser representado como uma máquina de estados que alterna dinamicamente entre esses três modos de operação:

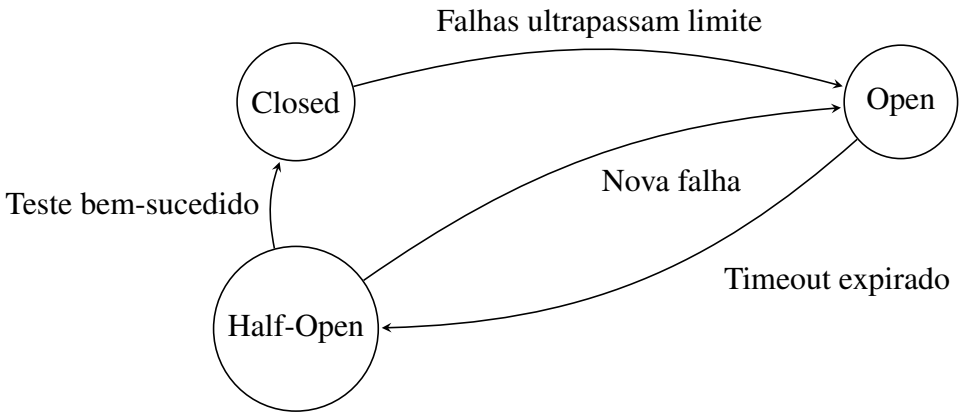

Figura 6: Modelo de estados do padrão Circuit Breaker

Quando ocorre uma falha e o circuito se abre, o parâmetro *timeout* define o período de espera antes de permitir novas tentativas. Após esse período, o estado *Half-Open* libera requisições de teste: se bem-sucedidas, o circuito fecha novamente; se falharem, o circuito permanece aberto para proteger o serviço de uma nova sobrecarga.

A configuração desses limites e tempos de espera é delicada: valores muito baixos podem cortar requisições antes do necessário, enquanto valores altos demais podem manter um serviço falho ativo por muito tempo. Assim como nos padrões *Retry* e *Timeout*, recomenda-se iniciar com valores razoáveis e ajustá-los com base na observabilidade e nas métricas reais do sistema [Newman, 2015].

2.5.3 Timeout

O padrão *Timeout* constitui uma estratégia fundamental para reforçar a resiliência de sistemas modernos [Kostenko, 2023]. Nesse contexto, o *timeout* define o período máximo de espera do microserviço solicitante pela resposta do microserviço requisitado. Se a resposta não chegar dentro desse intervalo, o serviço requisitado é considerado indisponível ou falho [Alashqar and Kurdya, 2022].

Além disso, a adoção de *timeouts* adequados promove o isolamento de falhas, de forma que possíveis problemas em outros sistemas, subsistemas ou dispositivos não precisem necessariamente se propagar e afetar todo o ambiente. Quando o *timeout* é configurado de maneira apropriada, falhas são isoladas; um problema em outro sistema não precisa, obrigatoriamente, tornar-se o seu problema. [Nygard, 2018]

Conexões em redes instáveis podem resultar em falhas e lentidão, afetando operações síncronas e, em alguns casos, levando a *deadlocks* — situações em que dois ou mais processos ou *threads* ficam bloqueados esperando recursos indefinidamente [Troubitsyna, 2019]. Para prevenir esses problemas, mecanismos de *timeout* delimitam o período máximo de espera por uma resposta. Sem tais mecanismos, um serviço corre o risco de aguardar indefinidamente [Kostenko, 2023].

Conforme ilustra a Figura 7, o *Serviço Consumidor* inicia uma contagem de tempo configurada ao enviar a requisição para o *Serviço Provedor*. Se não houver resposta dentro do intervalo estipulado, o mecanismo de *timeout* é acionado, encerrando a espera e reportando uma falha temporária. Isso evita que o consumidor permaneça indefinidamente bloqueado na expectativa de uma resposta que talvez nunca chegue.

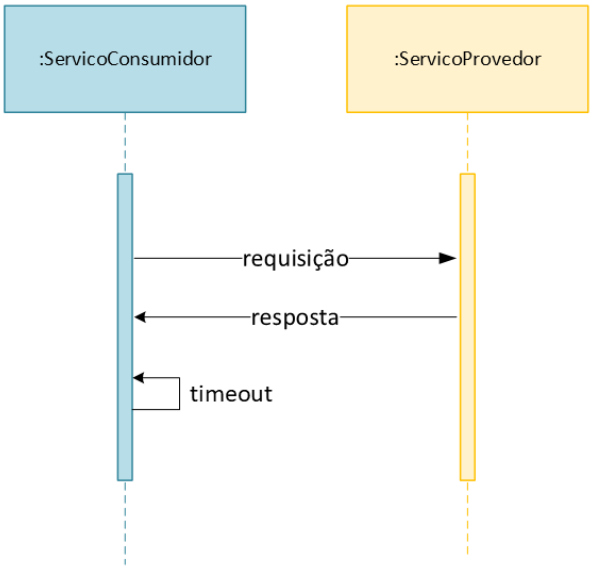

Figura 7: Exemplo de Interação do Padrão Timeout [Troubitsyna, 2019]

É comum também que *timeouts* sejam aplicados em conjunto com novas tentativas de execução (*retries*). O software busca repetir as operações que excederam o tempo limite, proporcionando mais chances de sucesso em condições de rede instáveis [Nygard, 2018].

Em aplicações que recebem muitas requisições simultâneas, o uso do padrão *Timeout* pode dificultar o processamento, pois algumas *threads* podem ficar bloqueadas. Além disso, definir um valor de tempo limite apropriado para cada tipo de serviço dependente tende a tornar o sistema mais complexo, já que cada serviço pode exigir configurações específicas [Kostenko, 2023].

2.5.4 Bulkhead

Imagine um navio cruzando o oceano. Para proteger a embarcação de um possível naufrágio, engenheiros navais projetaram compartimentos estanques — chamados *bulkheads* — que isolam partes do casco. Assim, mesmo que ocorra uma ruptura e a água invada um compartimento, os demais permanecem intactos, evitando que o dano se espalhe e leve toda a embarcação ao fundo.

Inspirado nessa estratégia de contenção, o padrão *Bulkhead* aplica o mesmo princípio aos sistemas de software: segmentar e isolar recursos críticos para impedir que falhas em uma parte comprometam toda a aplicação [Nygard, 2018]. A ideia é particionar o ambiente em segmentos autônomos, de modo que cada um possa falhar ou ser sobrecarregado sem arrastar o restante consigo.

No contexto de aplicações modernas, o padrão *Bulkhead* é fundamental para evitar sobrecargas quando diversos serviços ou módulos são executados simultaneamente. Ele promove a criação de "compartimentos" de recursos — como conexões, memória ou threads — para cada serviço, reduzindo a possibilidade de um pico de tráfego ou uma falha em um serviço afetar toda a operação [Kostenko, 2023]. Isso envolve, por exemplo, a utilização de *pools* de conexões, que atuam como reservatórios compartilhados e configuráveis para gerenciar de forma eficiente o uso de conexões com bancos de dados. A separação desses *pools* por serviço reduz o risco de um serviço monopolizar todas as conexões disponíveis, comprometendo o desempenho de outros módulos [Sobri et al., 2022]

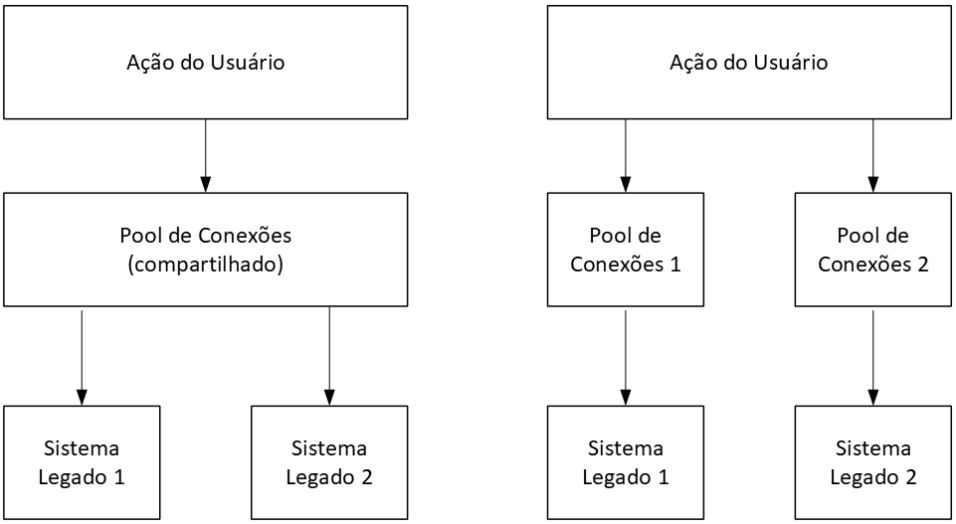

Figura 8: Exemplo de Bulkhead [Alashqar and Kurdya, 2022]

À esquerda, vemos um único *pool* de conexões compartilhado, no qual dois sistemas legados — Sistema Legado 1 e Sistema Legado 2 — competem pelos mesmos recursos. Nesse cenário, uma falha ou lentidão em um dos sistemas pode consumir todas as conexões disponíveis, impactando diretamente o outro sistema.

À direita, em contrapartida, o padrão *Bulkhead* cria divisórias entre os sistemas, alocando *pools* de conexões separadas. Dessa forma, mesmo que o Sistema Legado 1 enfrente problemas, o Sistema Legado 2 mantém acesso aos seus próprios recursos, permanecendo isolado das consequências do incidente.

Embora eficaz, a aplicação do *Bulkhead* apresenta desafios relacionados à otimização de desempenho e ao equilíbrio entre segmentação e aproveitamento de recursos. Isso envolve analisar e ajustar múltiplas variáveis interdependentes, como o número de instâncias, a quantidade de threads e a memória disponível, sempre considerando as necessidades e restrições de cada

serviço [Nygard, 2018].

2.5.5 Fallback

Fallbacks são mecanismos acionados quando um serviço remoto apresenta falhas ou está indisponível, direcionando a busca de informações para um serviço alternativo que esteja funcionando adequadamente [Meiklejohn, 2024].

Em vez de retornar uma exceção diretamente ao usuário quando algo dá errado, o padrão fallback recorre a ações alternativas, como buscar dados em uma fonte secundária ou enfileirar a solicitação para processamento futuro. Dessa forma, o usuário não recebe um erro imediato, mas pode ser informado de que sua solicitação será atendida posteriormente, garantindo uma experiência mais fluida e confiável [Sanchez and Carnell, 2021].

Por exemplo, em um sistema de monitoramento climático, ao solicitar dados meteorológicos, o sistema inicialmente tenta recuperar as informações de uma API principal. Caso essa fonte falhe, o padrão fallback pode redirecionar a consulta para uma base de dados secundária, como registros históricos. Se nenhuma dessas alternativas estiver disponível, a solicitação é enfileirada, e o usuário é notificado de que os dados serão fornecidos assim que possível.

O padrão fallback pode ser arriscado em situações em que a segurança do sistema depende de respostas críticas, como na verificação de fraudes, podendo levar à aprovação de transações fraudulentas.

2.6 Cache

O *cache* é um mecanismo de armazenamento baseado no modelo *key-value*, que mantém dados em memória sem a necessidade de gravação permanente em arquivos. Esse método reduz a quantidade de leituras e escritas realizadas diretamente no disco, proporcionando um desempenho muito superior ao aproveitar a velocidade da memória *RAM* [Falkevych and Lisniak, 2023].

Uma das soluções mais populares para implementar *cache* em aplicações *web* é o *Remote Dictionary Server* (*Redis*) . O *Redis* é um banco de dados que funciona inteiramente em memória e adota o modelo *key-value* para armazenamento. Além de sua velocidade e simplicidade, destaca-se por oferecer mecanismos avançados de persistência e suporte a diversas estruturas de dados, o que o torna especialmente útil em sistemas distribuídos que requerem alta disponibilidade [Kanthed, 2023].

A Figura 9 ilustra o funcionamento do *cache* em um cenário onde o *Redis* atua como intermediário entre o usuário e o banco de dados principal. Esse processo ocorre em quatro etapas principais:

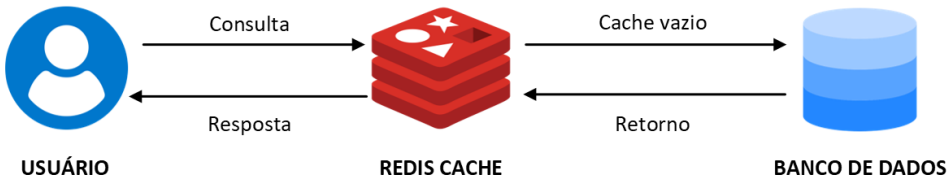

Figura 9: Fluxo de funcionamento do *Redis* como camada de *cache*, intermediando requisições entre o usuário e o banco de dados.

Nesse cenário, quando o usuário faz uma solicitação de dados, o sistema primeiro consulta o *Redis* para verificar se a informação já está armazenada em memória. Caso os dados estejam disponíveis, o *Redis* responde diretamente ao usuário, reduzindo a latência e poupando recursos do banco de dados. Entretanto, se o *cache* não contiver a informação — situação conhecida como *cache vazio* —, o sistema encaminha a requisição ao banco de dados principal para obter a resposta. Em seguida, além de fornecer a resposta ao usuário, o sistema armazena esses dados no *Redis*, de modo que consultas futuras sejam atendidas diretamente pela camada de *cache*.

Esse modelo, chamado *Cache-Aside* (ou *Lazy Loading*), permite que as informações mais relevantes sejam mantidas em memória, otimizando o tempo de resposta sem comprometer a consistência dos dados [Pamula et al., 2014]. Apesar de todos os benefícios proporcionados por esse padrão, é essencial que a configuração e o gerenciamento do *cache* sejam realizados de maneira criteriosa, para evitar problemas como o *cache stampede* — ou *dog-piling* —, em que múltiplas requisições simultâneas tentam acessar ou atualizar um dado expirado, sobrecarregando o banco de dados [Falkevych and Lisniak, 2023; Vattani et al., 2015]. Dessa forma, o uso adequado do *Redis* como camada de *cache* é um fator determinante para garantir alta performance e escalabilidade em sistemas modernos.

2.7 Discussão

A análise dos padrões arquiteturais e tecnologias apresentadas revela um ecossistema interdependente, no qual a resiliência, escalabilidade e eficiência de sistemas distribuídos dependem da combinação estratégica de múltiplas abordagens.

Padrão/Mecanismo	Vantagens	Desvantagens
Retry	Permite novas tentativas em falhas transitórias, aumentando a resiliência.	Pode causar sobrecarga ou latência se mal configurado.
Circuit Breaker	Protege o sistema de sobrecarga ao isolar falhas.	Difícil de configurar os limites adequados.
Timeout	Evita bloqueios indefinidos e isola falhas.	Complexidade em determinar tempos limites ideais.
Bulkhead	Isola recursos, prevenindo que a sobrecarga de um serviço afete outros.	Requer particionamento cuidadoso dos recursos.
Fallback	Oferece alternativas em caso de falhas, garantindo continuidade.	Pode comprometer segurança em cenários críticos.
Cache	Reduz a latência e melhora o desempenho ao evitar consultas repetitivas ao banco de dados.	Risco de inconsistência nos dados armazenados e problemas como *cache stampede*.
Load Balancer	Distribui carga entre servidores, otimizando o uso de recursos e aumentando disponibilidade.	Pode gerar distribuição desigual se não configurado corretamente, além de desafios em persistência de sessão.

Tabela 2: Comparação entre padrões arquiteturais e mecanismos de otimização

Embora a Tabela 2 sintetize as vantagens e desafios de cada solução, a real efetividade desses padrões só emerge quando contextualizada em cenários práticos, considerando sinergias entre eles, práticas de desenvolvimento e o ambiente operacional de cada sistema.

Em sistemas distribuídos, falhas são inevitáveis e podem ocorrer em diferentes níveis — desde a comunicação entre microsserviços até a sobrecarga de servidores ou a indisponibilidade temporária de componentes críticos. Por isso, a escolha entre esses padrões não é excludente, mas sim complementar, pois cada abordagem atua de forma distinta, tratando problemas específicos sem necessariamente substituir as demais.

Por exemplo, em ambientes de alta demanda e acesso concorrente, o padrão de *cache* surge como solução para reduzir a latência e aliviar a carga do banco de dados, como observado no uso do *Redis*, que armazena dados em memória para agilizar o atendimento de requisições subsequentes. Esse mecanismo, ao atuar como intermediário entre o usuário e o banco de dados, colabora para a resiliência ao mitigar gargalos de leitura em cenários críticos (Kanthed, 2023).

Entretanto, esses mesmos recursos de cache introduzem desafios adicionais, como riscos de inconsistência e fenômenos de *cache stampede*, que surgem em cenários de grande volume de requisições concorrentes [Falkevych and Lisniak, 2023].

Além disso, como indicam as observações do texto de referência, padrões como *Retry*, *Circuit Breaker* e *Fallback* atuam de forma complementar para lidar com diferentes tipos de falhas. O *Retry* permite novas tentativas de comunicação em falhas transitórias, mas pode sobrecarregar o sistema caso não seja limitado adequadamente. Nesse ponto, a integração com *Circuit Breaker* se mostra essencial, pois ele bloqueia temporariamente chamadas que excedam um limiar de falhas, prevenindo sobrecarga e evitando que um serviço instável agrave a situação. Essa combinação inteligente entre padrões fortalece a resiliência e a eficiência do sistema.

No contexto do modelo de maturidade proposto por esta pesquisa, esses resultados reforçam que a resiliência em sistemas distribuídos não depende de um único padrão ou ferramenta, mas sim de um ecossistema que integra práticas de monitoramento, adaptação e aprendizado. O ajuste contínuo de parâmetros, como tempos de *timeout* e limites de falha do *Circuit Breaker*, aliado a práticas de observabilidade como *tracing* e análise de métricas, confirma a relevância de alinhar a arquitetura técnica com as necessidades do domínio e as características dinâmicas do ambiente [Newman, 2015; Kostenko, 2023].

Em suma, a maturidade em resiliência, tema central desta pesquisa, não se alcança pela simples aplicação de padrões isolados, mas sim pela capacidade de integrá-los de forma coesa, considerando seus *trade-offs* (as compensações necessárias para equilibrar vantagens e desvantagens em cada decisão arquitetural) e ajustando-os às necessidades específicas de cada serviço. Essa integração, combinada com uma cultura organizacional de aprendizado e adaptação, garante que tecnologias como *cache Redis* e *Bulkhead* atuem como elementos de um ecossistema resiliente, preparado para lidar com falhas inevitáveis.

Capítulo 3

Trabalhos Relacionados

Capítulo 4

Metodologia

Capítulo 5

Estudo de Caso

Capítulo 6

Experimentação

Referências Bibliográficas

2007. *ISO/IEC Standard for Systems and Software Engineering – Recommended Practice for Architectural Description of Software-Intensive Systems*. 07 . ISBN 978-0-7381-5660-6. doi: 10.1109/IEEESTD.2007.386501.

Abdelkareem M. Alashqar and Zaki Kurdya. Examining the design patterns of microservices for achieving performance quality tactics. *International Journal of Academic Information Systems Research (IJAISR)*, 6(12):4–13, December 2022. ISSN 2643-9026.

L. Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Grosso, and Massimiliano Di Penta. An empirical study on the evolution of design patterns. pages 385–394, 09 2007. doi: 10.1145/1287624.1287680.

Sasa Baskarada, Vivian Nguyen, and Andy Koronios. Architecting microservices: Practical opportunities and challenges. *Journal of Computer Information Systems*, 60:1–9, 09 2018. doi: 10.1080/08874417.2018.1520056.

Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022. doi: 10.1109/ACCESS.2022.3152803.

Tony Bourke. *Server load balancing*. O'Reilly & Associates, Inc., USA, 2001. ISBN 0596000502.

Vitalii Falkevych and A.O. Lisniak. Methodology of cache invalidation in microservices architecture of the web applications. *Scientific Notes of Taurida National V.I. Vernadsky University. Series: Technical Sciences*, 1:131–135, 2023. doi: 10.32782/2663-5941/2023.1/20.

Martin Fowler. Monolithfirst, 2015. URL `https://martinfowler.com/bliki/MonolithFirst.html`.

Martin Fowler and James Lewis. Microservices. 2014. URL `http://martinfowler.com/articles/microservices.html`.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction

and reuse of object-oriented design. pages 406–431, 01 1993. ISBN 978-3-642-63970-8. doi: 10.1007/978-3-642-48354-7_15.

Surbhi Kanthed. Redis vs. memcached in microservices architectures: Caching strategies. *International Journal of Multidisciplinary Research and Growth Evaluation.*, 4:1084–1091, 01 2023. doi: 10.54660/.IJMRGE.2023.4.3.1084-1091.

BC ILLIA Kostenko. Antifragile microservice systems. 2023.

Ranjan Kumar Mondal, Payel Ray, and Debabrata Sarddar. Load balancing. 4:1–21, 01 2016.

Iiro Kuusijärvi. Introduction to modern observability, 2024. URL https://www.theseus.fi/handle/10024/858551. Degree: Bachelor of Business Administration.

Luan Lazzari and Kleinner Farias. Event-driven architecture and rest: An exploratory study on modularity, 10 2021.

C. Majors, L. Fong-Jones, and G. Miranda. *Observability Engineering*. O'Reilly Media, 2022. ISBN 9781492076414. URL https://books.google.com.br/books?id=KGZuEAAAQBAJ.

Christopher Meiklejohn. Resilient Microservice Applications, by Design, and without the Chaos. 6 2024. doi: 10.1184/R1/25901422.v1. URL https://kilthub.cmu.edu/articles/thesis/Resilient_Microservice_Applications_by_Design_and_without_the_Chaos/25901422.

Iasmin Santos Mendes. Arquitetura monolítica vs microsserviços: uma análise comparativa, 2021. URL https://bdm.unb.br/handle/10483/30715. Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Software) — Universidade de Brasília, Brasília, 2021.

Nabor Mendonça, Carlos Aderaldo, Javier Cámara, and David Garlan. Model-based analysis of microservice resiliency patterns. 02 2020. doi: 10.1109/ICSA47634.2020.00019.

Brenda M. Michelson. Event-driven architecture overview: Event-driven soa is just part of the eda story. Technical report, Patricia Seybold Group, jan 2006. URL http://dx.doi.org/10.1571/bda2-2-06cc.

Subasish Mohapatra, Subhadarshini Mohanty, and K.Smruti Rekha. Analysis of different variants in round robin algorithms for load balancing in cloud computing. *International Journal of Computer Applications*, 69:17–21, 05 2013. doi: 10.5120/12103-8221.

Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 1st edition, February 2015. ISBN 978-1491950357.

M.T. Nygard. *Release It!: Design and Deploy Production-ready Software*. Pragmatic programmers. Pragmatic Bookshelf, 2018. ISBN 9781680502398. URL `https://books.google.com.br/books?id=UW7jAQAACAAJ`.

Narendra Babu Pamula, K. Jairam, and B. Rajesh. Cache-aside approach for cloud design pattern. *International Journal of Computer Science and Information Technologies (IJCSIT)*, 5(2):1423–1426, 2014. URL `https://www.ijcsit.com`. V.KR., V.N.B & A.G.K College of Engineering, Andhra Pradesh, India.

Lucas Gabriel Nerys Pereira and Luciana Mara Freitas Diniz. Comparativo entre arquitetura monolítica e arquitetura de microsserviços. Trabalho de conclusão de curso, Pontifícia Universidade Católica de Minas Gerais, Betim, MG, 2022. URL `https://bib.pucminas.br/pergamumweb/vinculos/000018/00001811.pdf`. Trabalho de Conclusão de Curso (Sistemas de Informação) - PUC Minas, Unidade Betim, 2º semestre de 2022.

Mark Richards. *Software Architecture Patterns*. O'Reilly Media, Inc., 2015. ISBN 9781491925409.

Nada Salaheddin and Nuredin Ahmed. Microservices vs. monolithic architectures [the differential structure between two architectures]. *MINAR International Journal of Applied Sciences and Technology*, 4:484–490, 10 2022. doi: 10.47832/2717-8234.12.47.

Illary Huaylupo Sanchez and John Carnell. *Spring Microservices in Action*. Manning Publications, Shelter Island, NY, 2. ed. edition, 2021.

Nur Sobri, Mohamad Abas, Megat Syahirul Amin Megat Ali, Noorita Tahir, Azlee Zabidi, and Zairi Rizman. A study of database connection pool in microservice architecture. *JOIV : International Journal on Informatics Visualization*, 6:566, 08 2022. doi: 10.30630/joiv.6.2-2.1094.

Falahah Suprapto, Kridanto Surendro, and Wikan Sunindyo. Circuit breaker in microservices: State of the art and future prospects. *IOP Conference Series: Materials Science and Engineering*, 1077:012065, 02 2021. doi: 10.1088/1757-899X/1077/1/012065.

Elena Troubitsyna. Model-driven engineering of fault tolerant microservices. *Åbo Akademi University*, 2019.

Andrea Vattani, Flavio Chierichetti, and Keegan Lowenstein. Optimal probabilistic cache stampede prevention. *Proceedings of the VLDB Endowment*, 8:886–897, 04 2015. doi: 10.14778/2757807.2757813.