UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

ANDRÉ TORQUATO SILVA

MICROFRONT-ENDS COMO UMA EVOLUÇÃO ARQUITETURAL? UMA ANÁLISE COMPARATIVA COM APLICAÇÕES MONOLÍTICAS

QUIXADÁ

2025

ANDRÉ TORQUATO SILVA

MICROFRONT-ENDS COMO UMA EVOLUÇÃO ARQUITETURAL? UMA ANÁLISE COMPARATIVA COM APLICAÇÕES MONOLÍTICAS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

Orientadora: Profa. Ma. Lana Mesquita.

QUIXADÁ

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S578m Silva, André Torquato.
Microfront-ends como uma evolução arquitetural? : Uma análise comparativa com aplicações monolíticas / André Torquato Silva. – 2025.
64 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Sistemas de Informação, Quixadá, 2025.
Orientação: Profa. Ma. Lana Mesquita.

1. Microfront-ends. 2. Arquitetura de sofware. 3. Front-end. 4. React.js. 5. Vue.js. I. Título.
CDD 005

ANDRÉ TORQUATO SILVA

MICROFRONT-ENDS COMO UMA EVOLUÇÃO ARQUITETURAL? UMA ANÁLISE COMPARATIVA COM APLICAÇÕES MONOLÍTICAS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

Aprovada em: 18 de Dezembro de 2025

BANCA EXAMINADORA

Profa. Ma. Lana Mesquita (Orientadora)
Universidade Federal do Ceará (UFC)

Prof. Dr. Jeferson Kenedy Morais Vieira
Universidade Federal do Ceará (UFC)

Prof. Dr. Sidartha Azevedo Lobo de Carvalho
Universidade Federal do Ceará (UFC)

Huerbet Melgaço Morais
TCE Ceará

AGRADECIMENTOS

Em primeiro lugar, agradeço a Deus, pela força diária, pela saúde e pela clareza que precisei em cada etapa deste trabalho. Sei que sem essa força, este trabalho não seria possível.

Agradeço à minha família, pelo apoio incondicional, pela compreensão durante as longas horas de estudo e pelas palavras de incentivo quando o cansaço parecia maior que o progresso. Tudo o que conquistei até aqui carrega um pouco de cada um de vocês.

À minha orientadora, Prof. Ma. Lana Mesquita, deixo meu reconhecimento e gratidão pela paciência, pelas orientações e pela confiança ao longo do desenvolvimento deste trabalho. Agradeço também aos professores da banca, Prof. Dr. Sidartha Azevedo e Prof. Dr. Jeferson Kenedy Morais, pelas contribuições valiosas, e à Universidade Federal do Ceará (UFC), por proporcionar a formação, os recursos e o ambiente acadêmico que tornaram este estudo possível.

Aos meus amigos, que acompanharam essa trajetória de perto, agradeço por cada conversa, conselho e mensagem de apoio, principalmente nos momentos em que o acúmulo de demandas parecia tornar tudo mais difícil. A presença de vocês tornou o processo mais leve.

Por fim, agradeço a todos que contribuíram indiretamente para a realização deste trabalho, seja compartilhando conhecimentos, revisando algum trecho, oferecendo feedback, ou simplesmente torcendo pelo meu sucesso.

"O objetivo do trabalho é o jardim que se planta, ou a casa que se constrói, ou o livro que se escreve..." (Rubem Alves)

RESUMO

Nos últimos anos, a crescente complexidade e os desafios de manutenção das arquiteturas monolíticas tradicionais têm impulsionado a busca por soluções mais escaláveis e modulares. Inspirado nos princípios da arquitetura de microsserviços para o back-end, surge o conceito de microfront-ends (MFE), que visa aplicar a modularidade e a independência ao desenvolvimento da interface do usuário. Este trabalho investiga se os benefícios da arquitetura de microfront-ends justificam seus desafios operacionais e técnicos em comparação com a abordagem monolítica. O objetivo principal é compreender os impactos técnicos e operacionais da adoção de microfront-ends, analisando sua influência na eficiência dos pipelines de integração contínua (CI) e entrega contínua (CD) e nas métricas de desempenho com base nos indicadores de Web Vitals. A metodologia empregada é um estudo de análise comparativa, no qual uma aplicação de streaming de vídeo, desenvolvida em duas versões distintas: uma com arquitetura monolítica e outra baseada em microfront-ends, utilizando uma divisão vertical por rotas. Ambas as implementações contam com pipelines CI/CD automatizadas via GitHub Actions. A avaliação das métricas, realizada a partir dos dados e artefatos gerados pelas ferramentas de integração contínua, permite analisar indicadores como o tempo de execução das pipelines, o tamanho dos pacotes e outras métricas relevantes, incluindo o desempenho da aplicação. Os resultados revelam que, embora a arquitetura monolítica apresente menor custo operacional inicial e pacotes menores, a abordagem de microfront-ends demonstra ser mais vantajosa para a manutenção a longo prazo, apresentando taxas de crescimento no tempo de execução das pipelines (4,08%) e no tamanho dos pacotes (14,68%) que representam aproximadamente metade do ritmo de crescimento observado no monólito (11,79% e 28,63%, respectivamente), com impactos no desempenho de carregamento praticamente imperceptíveis para o usuário final.

Palavras-chave: microfront-ends; aplicações monolíticas; arquitetura de software; análise comparativa; integração contínua; entrega contínua.

ABSTRACT

In recent years, the increasing complexity and maintenance challenges of traditional monolithic architectures have intensified the search for more scalable and modular solutions. Inspired by the principles of back-end microservices, the microfront-ends (MFE) architectural approach has emerged as a means of applying modularity and team autonomy to user interface development. This study investigates whether the benefits of microfront-end architectures effectively outweigh their technical and operational challenges when compared to a monolithic approach. The primary objective is to assess the technical and operational impacts of adopting microfront-ends, particularly their influence on Continuous Integration (CI) and Continuous Delivery (CD) pipeline efficiency and on application performance metrics based on Web Vitals indicators. The research adopts a comparative analyses methodology, in which a video streaming application was implemented in two distinct versions: a traditional monolithic architecture and a microfront-end-based architecture employing vertical route segmentation. Both implementations rely on automated CI/CD pipelines configured using GitHub Actions. The evaluation is conducted through the analysis of artifacts and metrics generated during the continuous integration process, including pipeline execution time, bundle size growth, and application performance indicators. The results indicate that while the monolithic architecture presents lower initial operational costs and smaller bundle sizes, the microfront-end approach demonstrates superior long-term maintainability, exhibiting lower growth rates in pipeline execution time (4.08%) and bundle size (14.68%), approximately half of those observed in the monolithic architecture (11.79% and 28.63%, respectively), with negligible impact on perceived loading performance for the end user.

Keywords: microfront-ends; monolithic applications; software architecture; comparative analysis; continuous integration; continuous delivery.

LISTA DE FIGURAS

Figura 1 – Princípios dos microsserviços 17
Figura 2 – Representação da divisão vertical 19
Figura 3 – Representação da divisão horizontal 20
Figura 4 – Representação da composição de microfront-ends 21
Figura 5 – Representação da integração e entrega contínua 23
Figura 6 – Etapas da metodologia 30
Figura 7 – Diagrama de casos de uso da aplicação 34
Figura 8 – Resultado tempo médio de execução do pipeline CI/CD — Arquitetura microfront-ends vs Arquitetura monolítica 44
Figura 9 – Resultado tempo médio de execução do pipeline CI/CD por versão — Arquitetura microfront-ends 44
Figura 10 – Resultado tempo médio de execução do pipeline CI/CD por versão — Arquitetura monolítica 45
Figura 11 – Resultado tamanho médio do pacote (KB) - Arquitetura microfront-ends vs Arquitetura monolítica 46
Figura 12 – Resultado tamanho médio do pacote (KB) por versão - Microfront-end . . . 47
Figura 13 – Resultado tamanho médio do pacote (KB) por versão - Arquitetura Monolítica 47
Figura 14 – Resultado tamanho médio do pacote (KB) por repositório 48
Figura 15 – Relatório de desempenho - *Microfront-end* (MFE) Apresentação 49
Figura 16 – Relatório de desempenho - MFE Autenticação 50
Figura 17 – Relatório de desempenho - MFE Catálogo 51
Figura 18 – Relatório de desempenho - MFE Player 52
Figura 19 – Relatório de desempenho arquitetura monolítica - Apresentação 53
Figura 20 – Relatório de desempenho arquitetura monolítica - Autenticação 54
Figura 21 – Relatório de desempenho arquitetura monolítica - Catálogo 55
Figura 22 – Relatório de desempenho arquitetura monolítica - Player 56

LISTA DE ABREVIATURAS E SIGLAS

CD	*Entrega contínua*
CDN	*Content Delivery Network*
CI	*Integração contínua*
CI/CD	*Integração contínua/Entrega contínua*
CLS	*Cumulative Layout Shift*
DOM	*Document Object Model*
FCP	*First Contentful Paint*
KB	*Kilobyte*
LCP	*Largest Contentful Paint*
MFE	*Microfront-end*
SPA	*Single-Page Application*
TBT	*Total Blocking Time*

SUMÁRIO

1 INTRODUÇÃO 11
1.1 Objetivo 12
1.2 Objetivos Específicos 12
2 FUNDAMENTAÇÃO TEÓRICA 14
2.1 Arquitetura de Software 14
2.1.1 Evolução das arquiteturas de software 15
2.2 Arquitetura em Microsserviços 16
2.3 Arquitetura de Microfront-ends 18
2.3.1 Divisão Vertical e Horizontal 19
2.4 Composição de Microfront-ends 20
2.4.1 Application Shell 22
2.5 Integração e Entrega Contínua 22
2.5.1 Single Page Application (Single-Page Application (SPA)) 23
2.5.2 Testes Automatizados 23
3 TRABALHOS RELACIONADOS 25
3.1 Micro-Frontends: A multivocal literature review (Motivations, benefits, and issues for adopting) 25
3.2 Microfront-ends: Based performance improvement and prediction for microservices using machine learning 26
3.3 Analysis and comparison of microfront-ends and monolithic architecture for web applications 26
3.4 Quadro comparativa de trabalhos relacionados 27
4 METODOLOGIA 29
4.1 Configuração da infraestrutura 30
4.1.1 Instalação das dependências 31
4.1.2 Empacotamento da aplicação 31
4.1.3 Execução dos testes unitários 31
4.1.4 Coleta de métricas 31
4.1.5 Deploy para a nuvem 32
4.1.6 Invalidação de cache 32

4.1.7 *Configuração do ambiente em nuvem* . . . 32
4.2 **Implementação da aplicação com microfront-ends** . . . 33
4.2.1 *Definição da Análise* . . . 33
4.2.2 *Acessar página de apresentação* . . . 33
4.2.3 *Identificar usuário* . . . 34
4.2.4 *Visualizar catálogo de vídeos* . . . 35
4.2.5 *Visualizar Detalhes de um Vídeo* . . . 35
4.2.6 *Reproduzir Vídeo* . . . 35
4.2.7 *Implementação das aplicações* . . . 36
4.3 **Implementação da Aplicação Monolítica** . . . 37
4.4 **Coleta e análise de dados** . . . 38
4.4.1 *Coleta de dados* . . . 38
4.4.2 *Análise de dados* . . . 41
5 **RESULTADOS** . . . 43
5.1 **Resultados das métricas de tempo de execução das pipelines** . . . 43
5.2 **Resultados das métricas de tamanho dos pacotes** . . . 46
5.3 **Resultados das métricas de desempenho dos indicadores Web Vitals** . . . 49
5.4 **Análise dos Resultados** . . . 57
5.5 **Discussão dos Resultados** . . . 60
6 **CONCLUSÕES** . . . 61
6.1 **Limitações do Trabalho** . . . 61
6.2 **Trabalhos futuros** . . . 62
REFERÊNCIAS . . . 63

1 INTRODUÇÃO

Nos últimos anos, diversas organizações têm adotado amplamente a arquitetura em microsserviços como uma alternativa às arquiteturas monolíticas tradicionais. Essa abordagem busca escalar sistemas, reduzir a complexidade e o acoplamento, e tornar os componentes mais independentes (IBM, 2021).

A arquitetura monolítica, originalmente concebida e adotada como solução centralizada para aplicações com demandas de escalabilidade e múltiplos requisitos, tornou-se insustentável diante do crescimento dessas necessidades. À medida que as bases de código se expandiam rapidamente, surgiram desafios críticos, como o aumento da complexidade na manutenção, riscos de efeitos colaterais, dificuldades na coordenação de grandes equipes e obstáculos à adoção de práticas modernas como integração contínua e entrega contínua. Especialistas da área consideram essa rigidez estrutural, observada tanto em back-end quanto em front-end, um limitador à capacidade de adaptação às mudanças do mercado (Hidayat *et al.*, 2024; Mezzalira, 2021; Kaushik *et al.*, 2024).

Com o surgimento das aplicações cliente-servidor, houve uma divisão de responsabilidades entre o back-end, responsável pelo fornecimento de dados e lógica de negócio, e o front-end, responsável pela interface e experiência do usuário (Mezzalira, 2021).

A complexidade e o alto acoplamento de sistemas monolíticos levaram à exploração de arquiteturas distribuídas, com o objetivo de isolar o escopo e aumentar a autonomia das aplicações. Essa necessidade impulsionou o surgimento da arquitetura de microsserviços para o back-end. De forma semelhante, no front-end, foi proposto o conceito de microfront-ends, no qual se aplicam os mesmos princípios de modularidade e independência ao desenvolvimento da interface do usuário (Newman, 2015; Mezzalira, 2021).

A implantação da arquitetura em microfront-ends implica em uma maior complexidade técnica e organizacional (Mezzalira, 2021). A modularidade das aplicações pode aumentar o desafio cognitivo para compreender os diversos contextos presentes em um sistema. Gerando dificuldades de coordenação entre equipes. Esse cenário exige um ambiente maduro em ferramentas de automação, como *Integração contínua* (CI) e *Entrega contínua* (CD).

A integração contínua e a entrega contínua (CI/CD) são pontos-chave dessa arquitetura, pois permitem escalabilidade aos sistemas e garantem que sua evolução ocorra de forma simples. Esse processo permite a execução de processos essenciais para o sistema de forma automatizada, como testes automatizados e a publicação de diversos módulos independentes,

assegurando que a aplicação esteja funcionando e sendo distribuída corretamente (Mezzalira, 2021).

A adoção da arquitetura de microfront-ends deve ser avaliada de acordo com o contexto do projeto e da organização, pois não se trata de uma solução universal para todos os casos.

Diante desse contexto, surge a seguinte questão: **os benefícios da implantação da arquitetura de microfront-ends justificam seus desafios operacionais e técnicos em comparação com a arquitetura monolítica?** Este trabalho tem como objetivo examinar os **impactos operacionais** e **técnicos** da adoção de microfront-ends em relação à abordagem monolítica tradicional no desenvolvimento front-end com aplicações SPA.

Este trabalho busca identificar benefícios e limitações da arquitetura de microfront-ends em dimensões como **processos de integração contínua** e **entrega contínua** e métricas dos indicadores ***Web Vitals***[1]. Com base em evidências obtidas em ambiente controlado, exploramos a viabilidade técnica e as consequências operacionais da adoção de microfront-ends.

Com esse propósito, este estudo adotará uma abordagem de análise comparativa, implementando o mesmo projeto em duas versões: uma monolítica e outra baseada em microfront-ends. Essa comparação permitirá analisar atributos técnicos, como tempo de execução, tamanho dos pacotes gerados, nas etapas de integração e entrega contínua e métricas de desempenho com base nos indicadores *Web Vitals*, oferecendo percepções concretas sobre os impactos de cada arquitetura.

1.1 Objetivo

Compreender os impactos técnicos e operacionais da arquitetura de microfront-ends em comparação com a abordagem monolítica tradicional em aplicações front-end SPAs. Com base em avaliação empírica comparativa, este trabalho visa analisar o comportamento da arquitetura em microfront-ends em cenários reais de desenvolvimento front-end.

1.2 Objetivos Específicos

– Analisar o impacto da arquitetura de microfront-ends na eficiência dos pipelines de integração e entrega contínua, em comparação com aplicações monolíticas.

[1] Disponível em: https://web.dev/articles/vitals?hl=pt-br

- Comparar o tempo de execução das pipelines automatizadas entre as arquiteturas monolítica e de microfront-ends.
- Avaliar o tamanho dos pacotes gerados para cada aplicação implementada na arquitetura monolítica e em microfront-ends.
- Analisar o desempenho entre as aplicações com base nas métricas dos indicadores *Web Vitals* entre arquiteturas de front-end monolítico e microfront-ends.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a fundamentação para a compreensão dos elementos teóricos utilizados nesta pesquisa. Na Seção 2.1 são abordados os fundamentos da **arquitetura de software e sua evolução**. Na Seção 2.2, são apresentados os princípios e fundamentos da **arquitetura em microsserviços**. Na Seção 2.3, é introduzido e aprofundado o conceito de **microfront-ends**, permitindo uma compreensão mais ampla sobre essa abordagem.

2.1 Arquitetura de Software

A arquitetura de software representa a base conceitual sobre a qual um sistema é construído. Diferente da implementação, que trata dos detalhes técnicos de como funcionalidades são executadas, a arquitetura descreve o que deve acontecer e como os componentes do sistema interagem entre si. O arquiteto, nesse contexto, é responsável por estabelecer uma visão técnica coesa e garantir sua consistência ao longo de todo o ciclo de vida do software (Brooks Jr, 1995). Essa visão, no entanto, não deve ser encarada como estática. A arquitetura de software é, por natureza, evolutiva e adaptável. Em um cenário de constante transformação tecnológica e mudança de requisitos, torna-se essencial que a arquitetura permita ajustes contínuos. Modelos tradicionais têm cedido espaço a abordagens iterativas, nas quais o aprendizado e a adaptação orientam a construção arquitetural ao longo do tempo (Valente, 2020).

Nesse processo dinâmico, o arquiteto frequentemente se depara com decisões críticas e compromissos técnicos. Toda decisão arquitetural implica em concessões, como equilibrar desempenho com facilidade de manutenção ou escalabilidade com simplicidade estrutural, que precisam ser avaliadas com base nos objetivos do sistema e nas particularidades do contexto organizacional (Mezzalira, 2021).

As decisões arquiteturais impactam diretamente na capacidade de entrega do time, na escalabilidade da aplicação, na autonomia das equipes de desenvolvimento e na qualidade geral do software. Uma arquitetura bem definida promove a adoção de novas tecnologias, facilita a manutenção do código e contribui para sistemas mais confiáveis e sustentáveis a longo prazo (Newman, 2015).

2.1.1 Evolução das arquiteturas de software

Historicamente, projetos de software eram construídos com base em arquiteturas monolíticas, nas quais todas as funcionalidades permaneciam integradas em um único artefato. Essa abordagem era adequada para equipes pequenas e favorecia o foco direto na lógica de negócio. No entanto, à medida que o sistema evoluía e novas equipes, muitas vezes distribuídas geograficamente, eram incorporadas ao desenvolvimento, surgiam desafios significativos relacionados à escalabilidade, manutenção e coordenação. Essa crescente complexidade é amplamente discutida por Mezzalira (2021) e Newman (2015), que destacam as limitações das arquiteturas monolíticas em cenários de expansão organizacional e técnica.

A complexidade de manter um grande sistema monolítico se torna um desafio em ambientes onde há mudanças constantes nas demandas de negócio, conforme visto em Mezzalira (2021). A adaptação ou implementação de novas funcionalidades pode ser muito custosa, pois cada alteração no código pode impactar domínios aparentemente desconectados (Newman, 2015). Isso é ainda mais crítico quando não há uma base sólida de testes automatizados para garantir que as mudanças não afetem outras partes do sistema, o que dificulta a escalabilidade e compromete o processo de entrega contínua (Valente, 2020).

Como observado em Conway (1968), a **Lei de Conway** estabelece uma relação crucial entre a arquitetura de software e a estrutura organizacional. Conway (1968) notou que a estrutura de comunicação de uma organização inevitavelmente se reflete na arquitetura do sistema desenvolvido. Essa visão é complementada por Raymond (1999), que enfatiza que múltiplas equipes tendem, naturalmente, a produzir arquiteturas modularizadas, refletindo a divisão de responsabilidades. Diante dessa constatação, surgiu a chamada *Manobra Inversa de Conway*, descrita por Mezzalira (2021), na qual as organizações estruturam proativamente suas equipes para espelhar a arquitetura desejada. Um exemplo notável dessa prática é apresentado por Newman (2015), ao descrever o caso da Netflix, que organizou pequenas equipes independentes para garantir a autonomia dos serviços e a agilidade nas mudanças.

Introduzido por Brooks Jr (1995), o conceito “Não existe bala de prata” enfatiza que nenhuma solução única resolve todos os desafios do desenvolvimento de software. Este princípio destaca a necessidade de abordagens específicas e contextualizadas, rejeitando expectativas sobre soluções milagrosas e universais (Valente, 2020). A busca por uma nova arquitetura não elimina a complexidade, mas sim a transforma, trocando um conjunto de problemas conhecidos (os da arquitetura monolítica) por um novo conjunto (os de sistemas distribuídos).

Os padrões arquiteturais de microsserviços e microfront-ends são manifestações concretas da aplicação desses princípios. Os microsserviços visam decompor grandes aplicações monolíticas em um conjunto de serviços menores, independentes e focados em um domínio de negócio específico. Cada serviço é de propriedade de uma equipe pequena e autônoma, que controla seu ciclo de vida completo, da concepção à implantação e manutenção. Esta abordagem é uma aplicação direta da *Manobra Inversa de Conway*. Contudo, alinhada ao princípio de Brooks Jr (1995), ela não é uma "bala de prata", pois introduz sua própria complexidade técnica e organizacional de arquiteturas distribuídas, conforme apontado por Mezzalira (2021), entre eles:

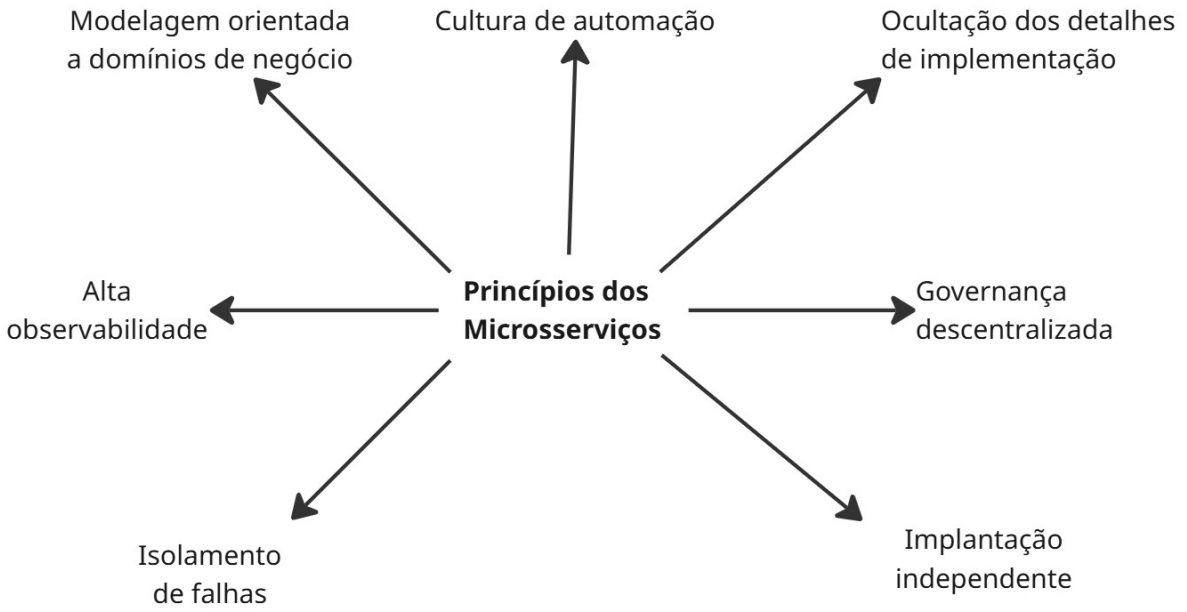
- **A necessidade de um pipeline de *Integração contínua/Entrega contínua* (CI/CD) robusto**, fundamental para o sucesso dessas arquiteturas. Gerenciar dezenas de serviços independentes exige automação eficiente para integração e entrega contínuas, tornando essa estratégia essencial.
- **A orquestração de serviços**, que diz respeito à forma como diferentes partes do sistema interagem para entregar funcionalidades completas de maneira coesa e eficiente.
- **O monitoramento distribuído**, que se torna mais complexo à medida que o sistema é dividido em múltiplos serviços. Apesar disso, é crucial manter o controle e garantir respostas rápidas a incidentes nas aplicações.
- **A resiliência**, entendida como a capacidade do sistema de lidar com falhas de módulos individuais sem que isso cause uma interrupção em cascata de todo o sistema (Newman, 2015).

2.2 Arquitetura em Microsserviços

Microsserviços constituem uma arquitetura utilizada no back-end de sistemas, serviços autônomos que colaboram entre si para formar uma arquitetura distribuída e eficiente. Essa abordagem permite decompor sistemas monolíticos em componentes menores e independentes, facilitando o desenvolvimento, implantação e gerenciamento por equipes autônomas (Newman, 2015). Tal divisão promove agilidade e eficiência, acelerando o ciclo de entrega de software ao mercado e permitindo adaptações rápidas às necessidades de negócios e tecnológicas.

A Figura 1 apresenta os princípios que sustentam a arquitetura de microsserviços. Uma implementação bem-sucedida depende da adesão a esses pilares, destacados por Newman (2015), que garantem que a autonomia e a distribuição não resultem em caos, mas sim em um

Figura 1 – Princípios dos microsserviços



Fonte: Adaptado de Newman (2015)

sistema coeso, resiliente e escalável.

- **Modelagem orientada a domínios de negócio:** estabelece que cada serviço deve refletir uma operação ou capacidade específica da organização. A arquitetura é, portanto, projetada com base em domínios de negócio bem compreendidos por toda a empresa.
- **Cultura de automação:** é outro pilar essencial dessa arquitetura, pois a automação de testes, integrações e entregas contínuas assegura qualidade, agilidade e confiabilidade no ciclo de desenvolvimento.
- **Ocultação dos detalhes de implementação:** garante que um serviço exponha claramente sua interface de comunicação, mas mantenha sua lógica interna encapsulada. Isso significa que ele deve resolver bem os problemas do seu domínio sem que outros serviços precisem saber como ele funciona internamente.
- **Governança descentralizada:** permite que as equipes sejam responsáveis pelos seus próprios domínios de negócio, tendo profundo conhecimento sobre eles. Ao contrário da governança centralizada, esse modelo promove maior autonomia nas decisões tecnológicas e possibilita a evolução dos serviços conforme as necessidades específicas de cada domínio.
- **Implantação independente:** preconiza que cada serviço deve ser projetado para ser implantado de forma autônoma, sem impactar outros serviços.
- **Isolamento de falhas:** determina que o sistema deve ser capaz de tolerar falhas em

um ou mais serviços, isolando os problemas e mantendo o restante da aplicação em funcionamento.

- **Alta observabilidade:** refere-se à capacidade de monitorar diversos serviços independentes. Esse é um dos maiores desafios dessa arquitetura, em comparação com aplicações monolíticas, onde os sistemas são centralizados. A observabilidade permite diagnosticar falhas e entender o comportamento de cada serviço de forma eficaz.

Esses são os pilares das arquiteturas distribuídas, aplicáveis tanto a microsserviços quanto a microfront-ends. Eles ajudam a orientar o planejamento e a execução dessas arquiteturas, com o objetivo de evitar problemas futuros e complexidade desnecessária.

2.3 Arquitetura de Microfront-ends

A arquitetura baseada em microfront-ends é uma **abordagem emergente** que representa uma estratégia para organizar entregas de software com base em domínios de negócio e áreas de responsabilidade, em contraste com as arquiteturas monolíticas tradicionalmente utilizadas na web. Essa abordagem surgiu como uma solução para escalar projetos desenvolvidos por múltiplas equipes, especialmente em cenários com grandes gargalos de autonomia e coordenação. Embora ainda seja considerada relativamente recente, já vem sendo adotada há alguns anos por organizações de médio e grande porte, demonstrando maturidade e viabilidade em ambientes corporativos complexos (Mezzalira, 2021).

Inspirada nos princípios dos microsserviços, a abordagem de microfront-ends adota a modelagem do sistema em torno de domínios de negócio, conforme apresentado por Newman (2015), Mezzalira (2021). Cada módulo da aplicação front-end é responsável por encapsular um domínio específico, de forma que as aplicações se mantenham independentes. Isso possibilita não apenas o isolamento de falhas, mas também a adoção dos demais princípios discutidos na Seção 2.2.

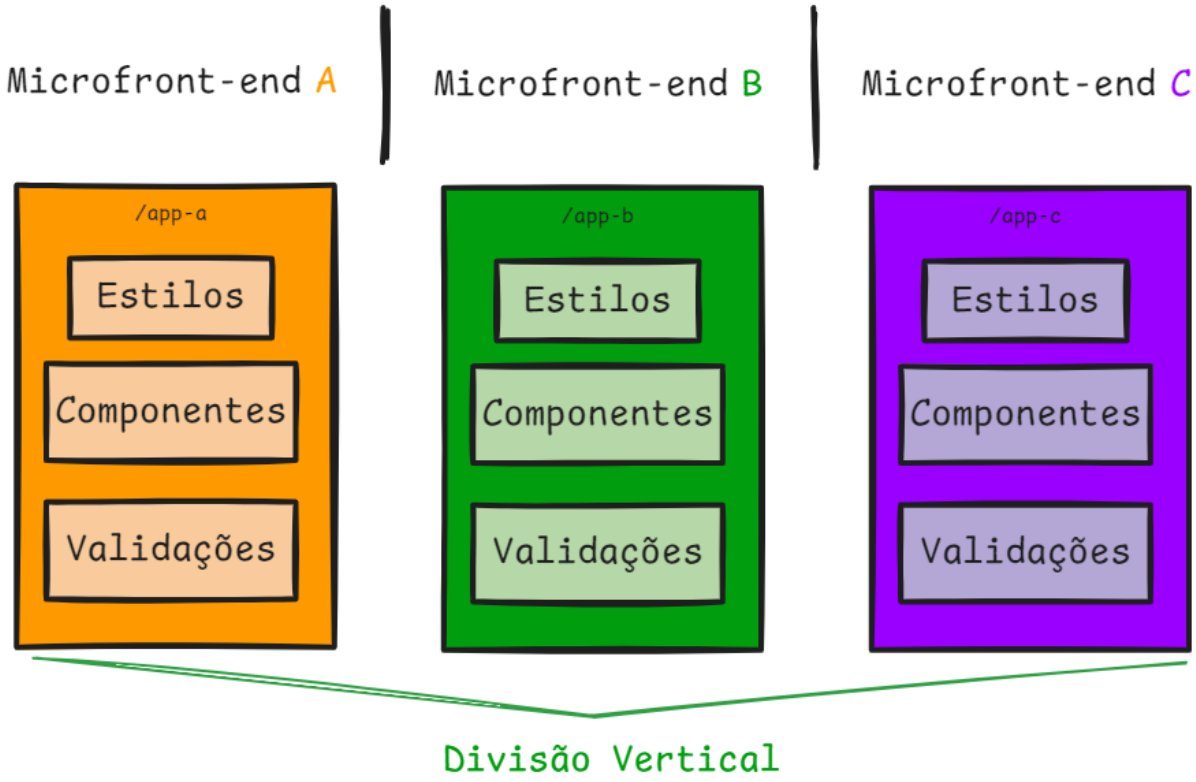
Para compreender mais a fundo o conceito de microfront-ends, é importante conhecer os tipos de arquitetura mais comuns, como a **divisão vertical** e a **divisão horizontal**, utilizadas nas aplicações. Também é essencial entender como é feita a **composição dos microfront-ends**, como os diferentes módulos se integram e qual o papel das ferramentas envolvidas para que tudo funcione como um sistema completo.

2.3.1 Divisão Vertical e Horizontal

As arquiteturas de microfront-ends podem ser organizadas de maneira **vertical** ou **horizontal**, dependendo de como os módulos, também chamados de MFE, são estruturados e integrados à interface do usuário.

A Figura 2 representa a **divisão vertical**, consiste em associar cada microfront-end a uma rota da aplicação, que determina qual parte do sistema deve ser acionada quando o usuário acessa um determinado endereço no navegador (Mezzalira, 2021). Por exemplo, ao acessar a rota "/produtos", a aplicação entende que deve responder com a listagem de produtos, relacionada ao microfront-end correspondente. A ferramenta responsável pela composição entende qual módulo deve ser carregado, garantindo que o MFE adequado seja exibido.

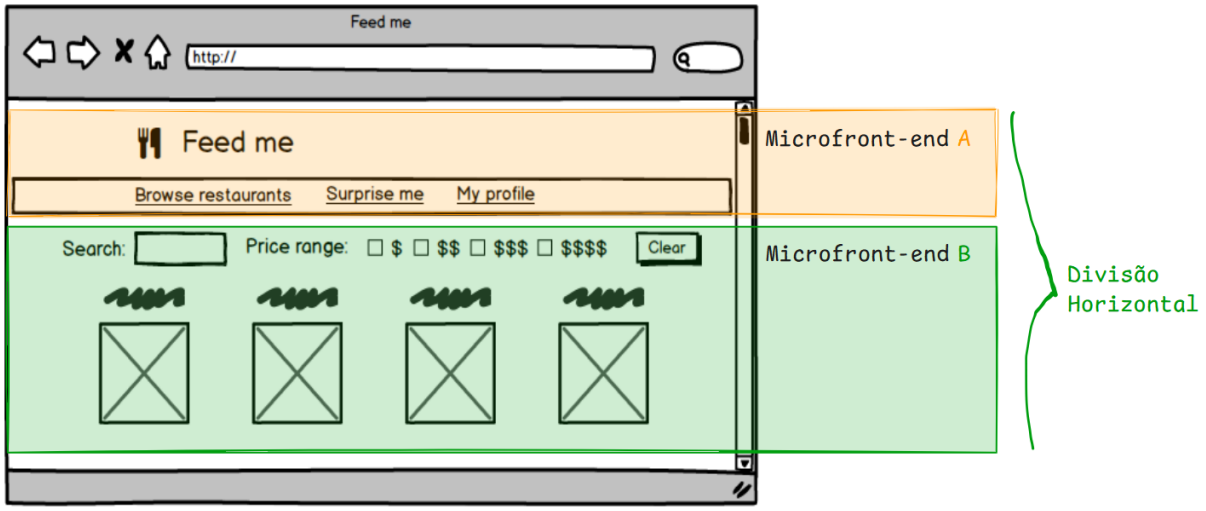
Figura 2 – Representação da divisão vertical



Fonte: Adaptado de Fowler (2019)

A Figura 3 representa a **divisão horizontal**, ocorre quando múltiplos microfront-ends são carregados simultaneamente em uma mesma rota (Mezzalira, 2021). Por exemplo, ao acessar a rota "/produtos", a página pode conter um cabeçalho e uma listagem de produtos, que, nessa abordagem, podem ser implementados como dois módulos de MFE distintos.

Figura 3 – Representação da divisão horizontal



Fonte: Adaptado de Fowler (2019)

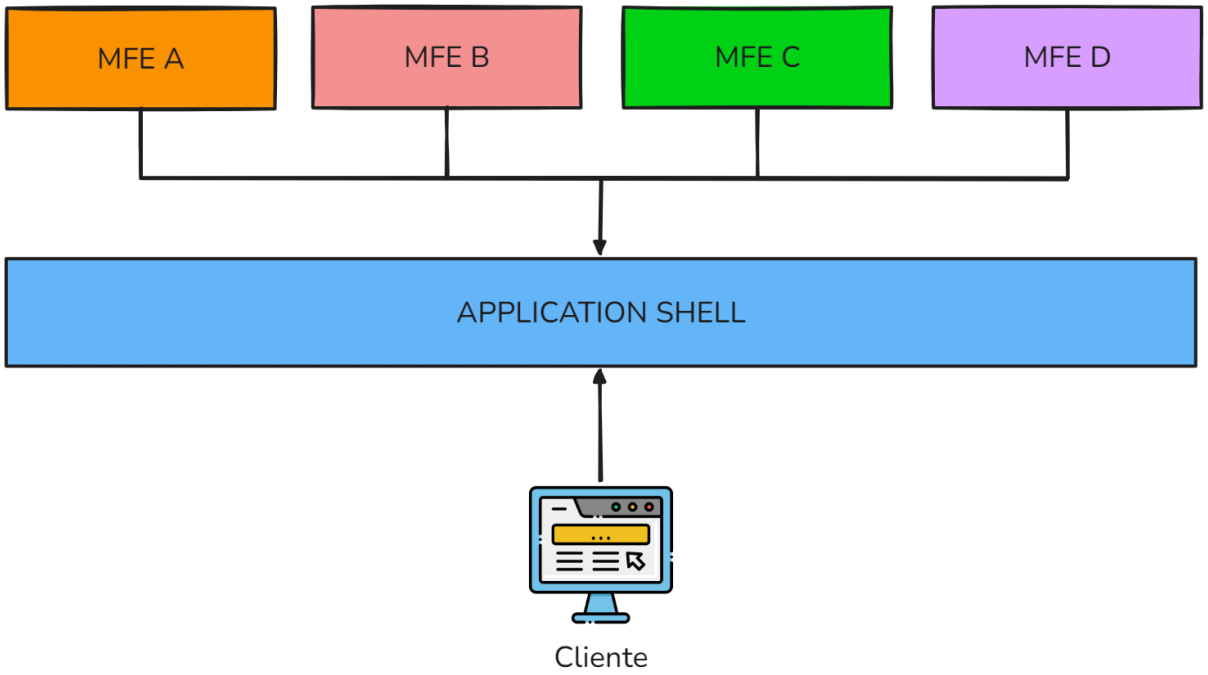
2.4 Composição de Microfront-ends

A **composição de microfront-ends** refere-se à forma como múltiplos microfront-ends independentes são reunidos para formar uma única interface coesa. Essa prática é uma das decisões centrais na arquitetura de microfront-ends, pois define como os fragmentos da aplicação serão orquestrados. Entre as diversas abordagens disponíveis, destaca-se a composição no lado do cliente, onde o navegador do usuário é responsável por carregar e renderizar os diferentes microfront-ends. Essa estratégia é geralmente estruturada por meio de um *application shell*, conforme ilustrado na Figura 4, que atua como ponto de entrada persistente da aplicação.

A composição no lado do cliente oferece benefícios relevantes, entre eles uma experiência de desenvolvimento semelhante à de aplicações front-end monolíticas convencionais. Essa abordagem facilita a adoção da arquitetura por equipes já acostumadas com esse modelo (Mezzalira, 2021). Além disso, essa abordagem permite um carregamento eficiente da aplicação, pois apenas os módulos necessários para a interação atual do usuário são baixados. Na divisão vertical, esse modelo reduz significativamente o risco de conflitos de dependência entre bibliotecas, já que apenas um microfront-end é carregado por vez. Por outro lado, a composição horizontal apresenta desafios importantes, como a comunicação entre microfront-ends e o isolamento da estilização das páginas. Este último refere-se à forma como o design das interfaces foi planejado e aos conflitos que podem surgir quando diferentes aplicações compartilham a mesma página (Mezzalira, 2021).

O *Module Federation* permite carregar módulos de forma assíncrona durante a

Figura 4 – Representação da composição de microfront-ends



Fonte: Adaptado de Mezzalira (2021)

execução da aplicação, viabilizando o compartilhamento de dependências entre microfront-ends sem necessidade de carregamento redundante. Essa tecnologia facilita a composição tanto vertical quanto horizontal, promovendo maior modularidade e independência entre os componentes. Outra solução, como os *Web Components*[1] contribui para a estruturação de aplicações distribuídas mais coesas e escaláveis (Mezzalira, 2021).

Nesse contexto, *Module Federation* opera a partir de dois conceitos fundamentais: ***host*** e ***remote***.

- **Host:** é a aplicação que conhece e consome os *remotes* da aplicação. Ele carrega módulos externos como se fossem bibliotecas locais, resolvendo dependências e executando códigos provenientes de outros projetos.
- **Remote:** é um módulo independente (microfront-end) que expõe partes de sua aplicação para serem consumidas pelo *host*, permitindo a composição distribuída e o compartilhamento de código entre diferentes microfront-ends.

[1] Web Components, facilita a construção de interfaces reutilizáveis e compatíveis com qualquer framework ou aplicação web.

2.4.1 Application Shell

O ***Application shell*** assume um papel central na composição da interface. Trata-se de uma camada persistente que é carregada inicialmente e permanece ativa durante toda a sessão do usuário, sendo responsável por orquestrar e gerenciar a aplicação como um todo. Sua principal função é montar e desmontar dinamicamente os microfront-ends conforme o usuário interage com domínios da aplicação. Para isso, o *application shell* expõe mecanismos de controle de ciclo de vida, permitindo que os microfront-ends reajam adequadamente ao serem carregados ou desmontados (Mezzalira, 2021).

2.5 Integração e Entrega Contínua

A CI é uma prática do desenvolvimento de software que consiste em integrar frequentemente o código em desenvolvimento à base principal, com o objetivo de evitar conflitos complexos e facilitar a evolução contínua do sistema. Essa prática é especialmente importante em contextos de metodologias ágeis, que adotam abordagens incrementais e envolvem múltiplas equipes trabalhando em paralelo, o que pode aumentar a complexidade de manutenção do projeto (Valente, 2020). Por meio da automação, a integração contínua assegura que o processo de compilação e os testes automatizados sejam executados de forma constante, garantindo que a aplicação continue funcionando corretamente ao longo de sua evolução.

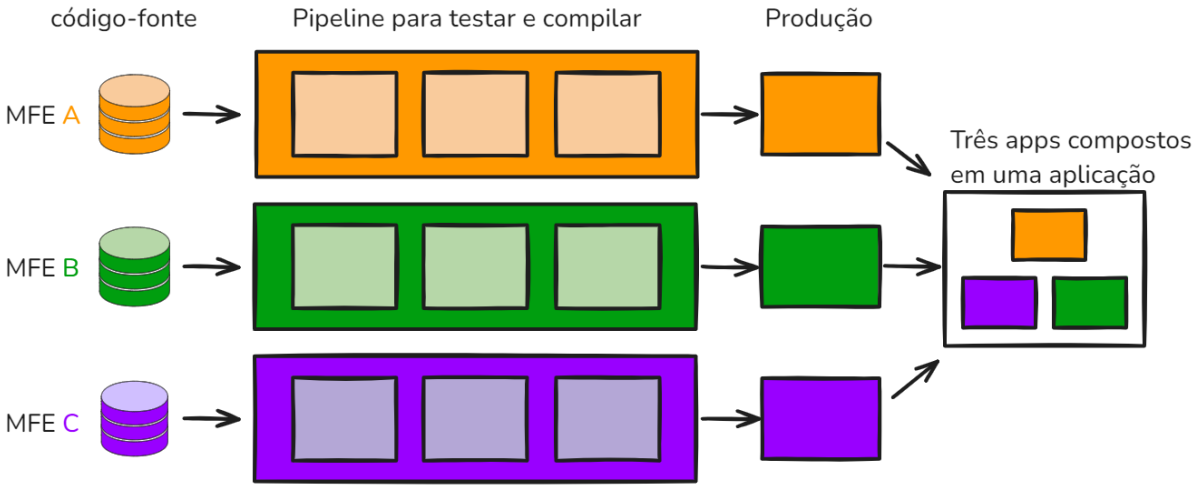
A CD é uma extensão do processo de integração contínua. Ela não apenas assegura que o código seja integrado e testado continuamente, mas também garante que esteja sempre em um estado pronto para ser liberado para produção. Dessa forma, novas funcionalidades e correções de erros podem ser disponibilizadas aos usuários quase que imediatamente (Valente, 2020).

As pipelines representam o processo automatizado completo que engloba a execução da Integração Contínua e da Entrega Contínua. Elas incluem etapas como testes unitários, testes de integração, compilação da aplicação e verificação da qualidade do código, garantindo que ele esteja em conformidade com os padrões estabelecidos. Além disso, contribuem para a redução da carga cognitiva dos desenvolvedores, promovendo maior agilidade no processo automatizado (Valente, 2020).

No contexto de microfront-ends, a integração e entrega contínua trazem benefícios, como a possibilidade de implantação independente, permitindo que cada módulo possua sua

própria pipeline. A Figura 5 representa essa abordagem, onde cada microfront-end possui sua pipeline. Essa abordagem favorece a agilidade, especialmente em ambientes com domínios de negócio fragmentados, além de proporcionar maior autonomia entre as equipes. No entanto, também impõe desafios, como o aumento da complexidade da infraestrutura e a dificuldade de monitoramento das aplicações distribuídas. Ainda assim, nesse cenário, a adoção da integração e entrega contínua é essencial para alcançar autonomia, agilidade e escalabilidade dentro dessa arquitetura (Mezzalira, 2021).

Figura 5 – Representação da integração e entrega contínua



Fonte: Adaptado de Fowler (2019)

2.5.1 Single Page Application (SPA)

A arquitetura Single Page Application (SPA) caracteriza-se por um modelo de aplicação web no qual toda a interação do usuário ocorre dentro de uma única página *HTML*, sendo a navegação e a atualização da interface realizadas dinamicamente no lado do cliente por meio de código *JavaScript*. Nesse modelo, o navegador assume a responsabilidade pela renderização da interface, pelo gerenciamento de estado e pelo controle das rotas da aplicação, enquanto o servidor atua predominantemente como provedor de dados e serviços, geralmente por meio de *APIs* (Mezzalira, 2021).

2.5.2 Testes Automatizados

Testes automatizados são práticas fundamentais na engenharia de software, com o objetivo principal de verificar automaticamente se o programa atende às expectativas especifi-

cadas, identificando problemas precocemente, ainda durante o desenvolvimento. O avanço dos métodos ágeis ampliou significativamente a relevância dos testes automatizados, destacando sua importância na prevenção de falhas, na proteção contra regressões e como meio eficaz de documentação técnica (Valente, 2020).

3 TRABALHOS RELACIONADOS

Os trabalhos relacionados presentes nesta pesquisa visam concretizar o entendimento sobre microfront-ends, além de servirem de referência para a definição da metodologia. A Seção 3.1 traz uma **revisão sistemática da literatura sobre microfront-ends**. A Seção 3.2 descreve uma aplicação prática da **arquitetura de microfront-ends e microsserviços, com foco nos impactos sobre o desempenho**. A Seção 3.3 analisa os efeitos da adoção de uma arquitetura monolítica em comparação aos microfront-ends, com ênfase nos impactos relacionados à **integração e entrega contínuas**.

3.1 Micro-Frontends: A multivocal literature review (Motivations, benefits, and issues for adopting)

Peltonen *et al.* (2021) apresentam uma análise sistemática da literatura sobre microfront-ends e evidenciam que a adoção dessa arquitetura é fortemente influenciada pelo aumento da complexidade no desenvolvimento de front-end, durante a evolução do sistema, onde grandes bases de código tornam-se inviáveis de escalar.

Nesse contexto, problemas organizacionais e a necessidade de escalar equipes impulsionam a busca por implantações independentes e entregas mais rápidas. Ao adotar uma arquitetura em microfront-ends, as equipes ganham liberdade para evoluir domínios de negócio, sem repercutir alterações em todo o projeto, preservando a estabilidade geral.

Entre os benefícios percebidos, como equipes mais autônomas que desenvolvem, testam, implantam e gerenciam seus domínios de forma isolada, estão também:

- Melhor testabilidade;
- Isolamento de falhas;
- Carregamento inicial mais rápido.

Contudo, a pesquisa também aponta que essa abordagem traz:

- Geração de duplicação de código;
- Dificulta o controle de dependências compartilhadas;
- Necessidade de padronização e controle mais rigorosos.

Esses estudos, além de ilustrar vantagens e desvantagens práticas, também sinalizam lacunas de pesquisa, como a identificação de padrões e anti-padrões, a avaliação de dívida técnica e a combinação de microfront-ends com outras soluções arquiteturais.

3.2 Microfront-ends: Based performance improvement and prediction for microservices using machine learning

Kaushik *et al.* (2024) realiza uma experimentação prática envolvendo a implementação da arquitetura de microfront-ends em conjunto com a de microsserviços, com o objetivo de avaliar o desempenho das aplicações por meio de testes de carga que simulam o tráfego real de usuários. O experimento foi controlado, com aumento gradual da carga de usuários, observando-se o comportamento das aplicações em relação ao tempo de resposta. Além disso, foram aplicadas técnicas de *Machine Learning* para auxiliar na previsão do tempo de resposta das aplicações.

Os autores também apontam os desafios de manutenção e escalabilidade próprios da arquitetura monolítica, como a dificuldade de coordenação entre grandes equipes e o risco de falhas no sistema como um todo, mesmo quando mudanças são feitas em apenas uma parte da aplicação. Além disso, ressaltam que, mesmo com a divisão do back-end em microsserviços, ainda persistem problemas de comunicação e escalabilidade no front-end monolítico, que continua centralizado e qualquer falha nele pode afetar toda a aplicação.

O artigo ressalta que a adoção de microfront-ends ajuda a evitar que as equipes fiquem presas a uma única tecnologia, além de promover mais estabilidade às aplicações e agilidade nas entregas. Suas conclusões propõem práticas voltadas a engenheiros de software que desejam otimizar aplicações seguindo essa abordagem.

3.3 Analysis and comparison of microfront-ends and monolithic architecture for web applications

Hidayat *et al.* (2024) apresenta um estudo de caso sobre a implementação da arquitetura de microfront-ends em uma aplicação originalmente monolítica, destacando os impactos dessa mudança arquitetural. Nesse contexto, a aplicação Smart BTW possuía pipelines com tempo de execução entre 30 e 50 minutos. Após a adoção da arquitetura em microfront-ends, esse tempo foi reduzido para apenas 1 a 2 minutos.

Os autores reforçam a ideia de que aplicações monolíticas altamente acopladas impõem gargalos no pipeline de entrega contínua e restringem a capacidade da organização de inovar com agilidade, uma vez que qualquer alteração exige a reconstrução e o reteste de todo o sistema. Em contraste, a arquitetura de microfront-ends fragmenta a aplicação em

subprojetos autônomos, cada um potencialmente escrito em versões distintas de ferramentas web, promovendo a modularidade e permitindo que múltiplas equipes trabalhem em paralelo, sem interferências diretas.

A configuração de pipelines automatizados com *GitHub Actions* para cada módulo, aliada ao uso do *Webpack Module Federation* durante a execução do sistema, reduz o tempo de compilação e montagem do projeto para 1 a 2 minutos e garante independência na publicação: alterações locais não exigem a recompilação de toda a aplicação.

O estudo também observou os impactos de desempenho relacionados ao tempo de interatividade das aplicações. Nas aplicações com arquitetura em microfront-ends, o tempo para que a aplicação se tornasse interativa foi de 12,77 segundos, em comparação aos 10,33 segundos nas aplicações monolíticas, segundo o estudo. No entanto, os ganhos em manutenibilidade, isolamento de falhas e escalabilidade compensam essa diferença.

Assim, segundo os autores, a adoção dessa arquitetura destaca-se como uma estratégia promissora para mitigar os gargalos observados em sistemas monolíticos e acelerar a entrega de valor ao usuário.

3.4 Quadro comparativa de trabalhos relacionados

O Quadro 1 apresenta a comparação entre os trabalhos relacionados com este trabalho.

Quadro 1 – Comparação dos trabalhos relacionados

Critérios	**Este trabalho**	**(Peltonen *et al.*, 2021)**	**(Kaushik *et al.*, 2024)**	**(Hidayat *et al.*, 2024)**
Taxa de Crescimento	X			
Diversidade Tecnológica	X		X	X
Tamanho do Pacote	X	X		
Métricas *Web Vitals*	X	X	X	X

Fonte: Elaborado pelo autor

O critério de **taxa de crescimento** avalia se o estudo analisa a evolução das métricas técnicas ao longo do incremento de novas funcionalidades e versões. Diferente de Peltonen *et al.* (2021), que oferece uma visão abrangente sobre as motivações para a adoção de microfront-ends através de revisões de literatura, este trabalho busca medir o ritmo de expansão de *pipelines* e

pacotes ao longo de quatro versões distintas.

A **diversidade tecnológica** identifica a capacidade e os impactos de integrar diferentes ferramentas em uma mesma aplicação. Diferente de Hidayat *et al.* (2024), que foca na redução de tempo de *pipeline* em um cenário de migração, este estudo demonstra a viabilidade prática da convivência entre tecnologias distintas ao implementar módulos em *Vue.js* e *React.js*.

O critério de **tamanho do pacote** analisa o peso final dos artefatos gerados para distribuição, métrica essencial para avaliar a eficiência do carregamento. Peltonen *et al.* (2021) discutem teoricamente o risco de duplicação de código e aumento de dependências na fragmentação de interfaces, mas não quantificam esse impacto em ambiente controlado. Neste trabalho, esse critério é explorado através de medições reais.

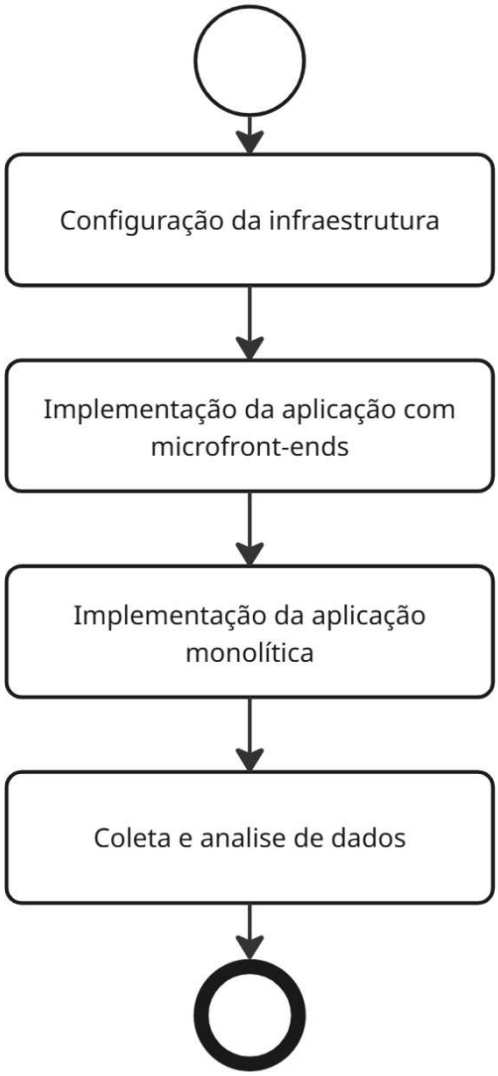
As **métricas *Web Vitals*** focam na experiência do usuário e na performance de renderização da interface através de seus indicadores apresentados posteriormente. O trabalho de Hidayat *et al.* (2024) utiliza técnicas de aprendizado de máquina para prever tempos de resposta em testes de carga; esta pesquisa utiliza auditorias do Lighthouse para medir o impacto direto na percepção de carregamento do usuário.

4 METODOLOGIA

A Figura 6 ilustra a metodologia dividida em quatro etapas principais: i) configuração da infraestrutura, ii) implementação da aplicação com microfront-ends, iii) implementação da aplicação monolítica, iv) coleta e análise de dados. O estudo propõe uma abordagem comparativa entre duas arquiteturas de front-end: uma monolítica e uma baseada em microfront-ends, aplicadas ao desenvolvimento de uma aplicação de *streaming* de vídeo, adaptada das funcionalidades da aplicação apresentada no estudo de caso teórico de Mezzalira (2021).

Este estudo adota como base metodológica o trabalho de Stoermer *et al.* (2003), que apresenta um método para apoiar a seleção de arquiteturas de software por meio da relação entre decisões técnicas e objetivos de negócio. Nesse contexto, é conduzida uma análise comparativa entre as arquiteturas monolítica e em microfront-ends.

Figura 6 – Etapas da metodologia



Fonte: Elaborada pelo autor.

4.1 Configuração da infraestrutura

A Seção 2.5 aborda a importância da integração e entrega contínua, especialmente no contexto de microfront-ends. Nesse sentido, a configuração de pipelines automatizadas para os projetos é uma das partes cruciais desta pesquisa, pois permite a independência entre os módulos e proporciona a geração de dados e embasamento teórico para compreender, de forma aprofundada, o comportamento tanto em arquiteturas monolíticas quanto em microfront-ends.

Cada projeto deste estudo possui um repositório próprio no GitHub[1], onde é realizado o controle de versão. A organização do código é feita por meio de *branches*, que permitem o desenvolvimento paralelo e incremental das funcionalidades. Neste trabalho, utilizam-se duas

[1] Disponível em: https://github.com/

branches principais:

- **develop**: ambiente de desenvolvimento contínuo.
- **main**: versão estável que representa o código de produção.

A automação de todo o fluxo é realizada com o GitHub Actions[2], que possibilita a execução de *pipelines* na nuvem a partir de eventos como *commits* ou sincronização entre *branches*. O *pipeline* de CI/CD desenvolvido para este estudo é composto pelas seguintes etapas:

4.1.1 Instalação das dependências

O *GitHub Actions* inicializa um ambiente de execução na nuvem e instala todas as dependências do projeto utilizando *Node.js*[3]. Essa etapa garante que o ambiente está padronizado e pronto para compilar ou testar o código.

4.1.2 Empacotamento da aplicação

A aplicação é compilada utilizando *Vite*[4], que fornece alto desempenho no processo de empacotamento. Em conjunto com *Module Federation*[5], o que permite que cada projeto seja construído de forma independente e compartilhado dinamicamente entre os microfront-ends. O resultado desta etapa é um conjunto de artefatos otimizados apresentados na Seção 2.4.

4.1.3 Execução dos testes unitários

Cada repositório possui uma configuração de testes unitários aplicado com *Vitest*[6], ferramenta que facilita a criação e execução de testes de forma rápida.

4.1.4 Coleta de métricas

Para cada novo *commit*, são coletadas métricas como:

- tempo total de execução da pipeline.
- tamanho do pacote gerado.
- contexto do *commit*.

[2] Disponível em: https://github.com/features/actions
[3] Disponível em: https://nodejs.org/pt
[4] Disponível em: https://vite.dev/
[5] Disponível em: https://module-federation.io/guide/basic/vite
[6] Disponível em: https://vitest.dev/

Essas métricas são enviadas automaticamente para uma planilha por meio de *scripts* dedicados, permitindo análise posterior e comparação entre as arquiteturas estudadas, discutidas na Seção 4.4.

4.1.5 Deploy para a nuvem

Quando a *branch* ***develop*** é sincronizada com a ***main***, o *pipeline* executa o processo de *deploy*. Os artefatos gerados são enviados para o *Amazon S3*[7], que armazena os arquivos estáticos. Em seguida, esses arquivos passam a ser distribuídos através do *Amazon CloudFront*[8], responsável por servir os conteúdos.

4.1.6 Invalidação de cache

Após o deploy, uma etapa de invalidação de *cache* é executada no *Amazon CloudFront*. Isso garante que os usuários recebam a versão mais recente dos arquivos, atualizando automaticamente as redes de distribuição que armazenam conteúdos em cache.

4.1.7 Configuração do ambiente em nuvem

A arquitetura em nuvem utilizada para armazenar e distribuir os artefatos do projeto é composta principalmente por dois serviços: ***Amazon S3***, responsável pelo armazenamento dos arquivos estáticos, e ***Amazon CloudFront***, encarregado de distribuí-los globalmente com segurança, desempenho e suporte adequado para aplicações do tipo SPA.

O ***Amazon S3*** é um serviço de armazenamento, ideal para hospedar arquivos estáticos como *HTML*, *CSS*, *JavaScript*, imagens e metadados. Ele organiza os arquivos em *buckets*, onde é possível configurar permissões granulares por meio de gerenciamento de identidade de acesso (IAM). Isso garante que apenas o *CloudFront* (e não o público geral) tenha acesso direto ao conteúdo, aumentando a segurança.

O ***Amazon CloudFront*** é uma *Content Delivery Network* (CDN) usada para entregar o conteúdo do *Amazon S3* otimizado com cache e segurança. Ele se conecta ao serviço de armazenamento e disponibiliza o conteúdo por meio de um domínio gerado pelo *CloudFront* (ou um domínio customizado).

[7] Disponível em: https://aws.amazon.com/pt/s3/
[8] Disponível em: https://aws.amazon.com/pt/cloudfront/

4.2 Implementação da aplicação com microfront-ends

Na Subseção 4.2.1, detalhamos a definição da aplicação desenvolvida e os casos de uso considerados na análise. Na Subseção 4.2.7, são apresentados os aspectos técnicos da implementação.

4.2.1 Definição da Análise

A aplicação idealizada para este trabalho é uma aplicação de *streaming* de vídeo, adaptada a partir do estudo de caso teórico apresentado no livro (Mezzalira, 2021). A aplicação contará com um catálogo de vídeos gerados por Inteligência Artificial no estilo de curtas-metragens, permitindo que o usuário visualize detalhes sobre os vídeos e os reproduza. O foco está em implementar funcionalidades que simulem o comportamento de uma aplicação SPA.

O fluxo se inicia em uma página de apresentação, que fornece uma visão geral da aplicação. Em seguida, o usuário realiza sua identificação, obtendo acesso ao catálogo de vídeos. A partir desse ponto, ele poderá visualizar os detalhes de cada vídeo e iniciar sua reprodução.

Os atores envolvidos nos casos de uso são **visitantes** e **usuários**.

O **visitante** é qualquer pessoa que acessa as partes públicas da aplicação, como páginas de apresentação.

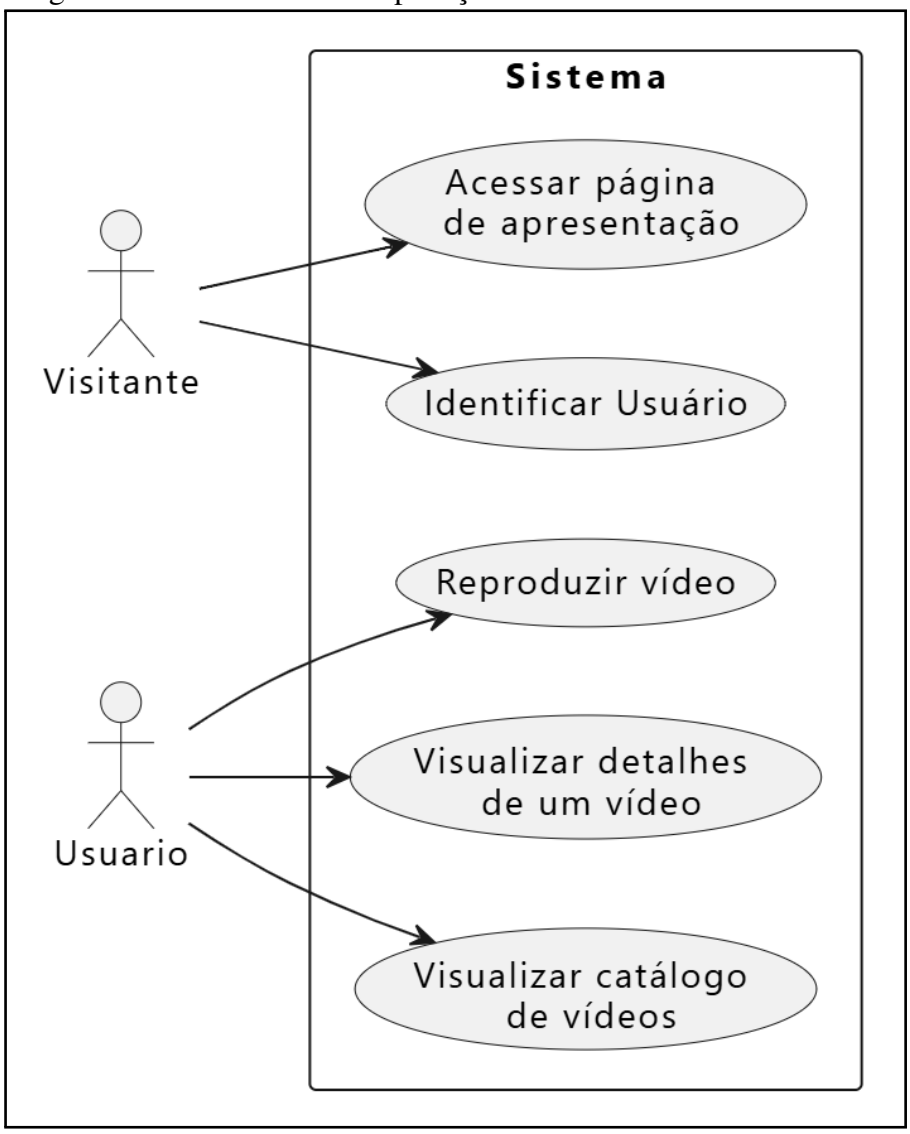
O **usuário** é aquele que possui credenciais de acesso e pode interagir com funcionalidades restritas da aplicação.

Esses fluxos estão representados na **??**:

4.2.2 Acessar página de apresentação

- **Ator:** Visitante
- **Fluxo Principal:**
    1. O visitante acessa o domínio principal da aplicação.
    2. O sistema exibe a página de apresentação com um elemento em destaque contendo uma chamada para a ação (ex: "Começar a assistir").
    3. O visitante pode rolar a página e visualizar seções explicativas sobre os recursos da plataforma.
    4. O visitante pode clicar no botão "Começar a assistir"
    5. O sistema redireciona para a página de Identificar usuário.

Figura 7 – Diagrama de casos de uso da aplicação



Fonte: Elaborada pelo autor.

4.2.3 Identificar usuário

- **Ator:** Usuário
- **Fluxo Principal:**
  1. O usuário acessa a página de identificar usuário.
  2. O sistema exibe o formulário de autenticação com campos para e-mail e senha. O usuário preenche os campos com suas credenciais.
  3. O usuário clica no botão “Entrar”.
  4. O sistema valida as credenciais fornecidas.
  5. Caso os dados estejam corretos, o sistema autentica o usuário e o redireciona para a página de catálogo.

- **Extensões:**
    1. **3a.** O usuário deixa algum campo vazio:
       O sistema exibe uma mensagem de erro solicitando o preenchimento obrigatório dos campos.
    2. **5a.** As credenciais estão incorretas:
       O sistema informa que e-mail ou senha são inválidos e permite nova tentativa.

4.2.4 Visualizar catálogo de vídeos

- **Ator:** Usuário
- **Fluxo Principal:**
    1. O usuário acessa a página de catálogo de vídeos após identificação do usuário.
    2. O sistema exibe o catálogo de vídeos disponíveis.
    3. O usuário pode rolar a página e navegar pelos diferentes vídeos.
    4. O usuário visualiza as miniaturas, títulos e descrições resumidas dos vídeos.
    5. O usuário pode clicar em um vídeo para acessar a página de detalhes ou iniciar a reprodução.

4.2.5 Visualizar Detalhes de um Vídeo

- **Ator:** Usuário
- **Fluxo Principal:**
    1. O usuário navega pelo catálogo e seleciona um vídeo.
    2. O sistema exibe a página de detalhes do vídeo com as seguintes informações:
       Título, sinopse e imagem de capa
       gênero e duração
       Botão para iniciar a reprodução
    3. O usuário pode clicar no botão “Assistir” para iniciar o vídeo.
    4. O usuário também pode voltar ao catálogo.

4.2.6 Reproduzir Vídeo

- **Ator:** Usuário
- **Fluxo Principal:**

1. O usuário acessa a página de detalhes de um vídeo.”.
2. O usuário clica no botão “Assistir”. O sistema carrega o vídeo e inicia a reprodução.
3. O usuário pode pausar, retomar, ajustar o volume ou alternar para tela cheia.
4. Ao final da reprodução, o sistema retorna ao catálogo.

4.2.7 Implementação das aplicações

Esta etapa foca no desenvolvimento dos casos de uso definidos na Subseção 4.2.1. A arquitetura de microfront-ends segue o modelo de **divisão vertical** descrito na Subseção 2.3.1. Cada módulo, denominado *MFE*, concentra um caso de uso específico. A organização das rotas reflete essa divisão, conforme descrito a seguir:

- **Nome:** MFE Apresentação
  Rota: `/`
  Caso de uso: Subseção 4.2.2
- **Nome:** MFE Autenticação
  Rota: `/auth`
  Caso de uso: Subseção 4.2.3
- **Nome:** MFE Catálogo
  Rota: `/catalog`
  Caso de uso: Subseção 4.2.4
- **Nome:** MFE Player
  Rota: `/player`
  Caso de uso: Subseção 4.2.6

A aplicação contará com o MFE denominado ***shell***, implementado em *Vue.js*[9], que desempenha o papel de **composição dos microfront-ends** (ver Seção 2.4). Esse módulo será responsável por gerenciar o carregamento dinâmico dos *remotes*, atuando como o *Application Shell* (ver Subseção 2.4.1) e integrando-se ao mecanismo do *Module Federation*. Nesse contexto, o *shell* assume o papel de *host*, isto é, o módulo que conhece os demais microfront-ends e permite que sejam carregados de forma dinâmica durante a execução da aplicação.

O controle de navegação será realizado com a biblioteca *Vue Router*[10], responsável pelo gerenciamento das rotas e pela definição dos fluxos de acesso à aplicação.

[9] Disponível em: https://vuejs.org/
[10] Disponível em: https://router.vuejs.org/

Todos os módulos MFE seguirão a mesma **tecnologia de implementação**, utilizando o framework *Vue.js*. Essa escolha baseia-se no conhecimento prévio sobre essa tecnologia.

A única exceção é o MFE responsável pela autenticação, que utiliza a biblioteca *React.js*[11]. Essa decisão evidencia uma das principais vantagens da arquitetura de microfront-ends, a possibilidade de integrar múltiplas ferramentas em um mesmo sistema.

A **estilização** da aplicação será realizada com *Tailwind CSS*[12], um *framework* utilitário baseado em *CSS* que oferece uma abordagem moderna para construção de interfaces. Diferente do *CSS* tradicional, o *Tailwind* fornece classes pré-definidas para cores, tamanhos, espaçamentos e demais propriedades visuais, permitindo que os estilos sejam aplicados diretamente no *HTML* ou nos componentes. Essa abordagem reduz a necessidade de criar arquivos de estilo separados, melhora a consistência visual e acelera o desenvolvimento.

Cada módulo implementado possui sua própria infraestrutura de pipelines automatizadas (ver Seção 4.1), elemento fundamental para o processo de integração e entrega contínua. Esse modelo traz benefícios importantes, como a possibilidade de implementação, evolução e implantação independentes.

4.3 Implementação da Aplicação Monolítica

Essa etapa consiste no desenvolvimento da aplicação utilizando uma arquitetura monolítica tradicional, contemplando os casos de uso apresentados na Subseção 4.2.1. Nesse modelo, os domínios de negócio e funcionalidades permanecem centralizados em um único projeto. A implementação será realizada com ***Vue.js***, utilizando o **Vite** como ferramenta de empacotamento.

A aplicação monolítica seguirá a **mesma estrutura de pipeline** adotada pelos módulos baseados em microfront-ends, incluindo todas as etapas descritas na Seção 4.1, de forma a garantir as mesmas condições de comparação.

O controle de rotas será realizado com a biblioteca **Vue Router**, responsável por gerenciar o carregamento das páginas de acordo com as rotas definidas.

A distribuição do projeto compilado com *Vite* reaproveitará os mesmos serviços de infraestrutura utilizados pelos microfront-ends, especificamente o **Amazon S3** e o **Amazon CloudFront**.

[11] Disponível em: https://react.dev/
[12] Disponível em: https://tailwindcss.com/

A aplicação também utilizará **Tailwind CSS** para estilização. O objetivo central é manter as implementações, tanto dos módulos MFE quanto da aplicação monolítica, o mais equivalentes possível, assegurando uma base comparativa consistente entre as duas arquiteturas.

4.4 Coleta e análise de dados

Na Subseção 4.4.1, detalhamos o funcionamento da coleta de dados e os atributos considerados na análise. Na Subseção 4.4.2, apresentamos os aspectos de análise de dados que podem ser extraídos, possibilitando a compreensão das perspectivas e diferenças entre as arquiteturas a partir de métricas extraídas e trabalhos relacionados.

4.4.1 Coleta de dados

A coleta de dados automatizada ocorre durante a execução das pipelines de CI no *GitHub Actions*. A cada execução, diversos atributos são gerados e utilizados para análise comparativa.

- **Tempo de execução da pipeline:** representa a duração completa dos processos configurados no CI. Esse parâmetro permite medir a agilidade do fluxo de entrega. Em sistemas monolíticos de grandes empresas, com bases de código extensas, esses processos podem levar de vários minutos a horas, tornando-se um fator crítico para disponibilizar novas funcionalidades ou correções importantes em produção.
- **Tamanho final do pacote:** utilizando a ferramenta *Vite*, o processo de compilação gera um diretório final, geralmente denominado de *dist*, que representa o pacote a ser distribuído na nuvem. A partir desse diretório, é possível calcular o tamanho final do pacote, métrica relevante para avaliar o tempo de carregamento da aplicação nos navegadores, já que arquivos maiores implicam maior tempo para que o usuário visualize e interaja com a interface.
- **Variáveis de ambiente e contexto de execução:** a execução em nuvem permite configuração semelhante ao ambiente local, possibilitando registrar informações como o **momento exato da execução**, o **contexto da aplicação** e demais valores que enriquecem a análise. Embora diversos outros dados possam ser capturados, este estudo se concentra nos seguintes atributos:

  Alguns desses dados não são explorados em profundidade neste estudo, mas podem

Quadro 2 – Descrição dos dados coletados das pipelines

Dado	**Descrição**
timestamp_utc	Momento exato em que a pipeline foi criada e executada.
repository	Repositório do GitHub onde o código está armazenado.
architecture_type	Tipo de arquitetura da aplicação: Microfront-end ou Monolítica.
context	Contexto de desenvolvimento relacionado à funcionalidade ou módulo.
commit_sha	Hash gerado pelo Git para identificar o commit na linha do tempo.
commit_message	Mensagem que descreve a alteração realizada no commit.
cc_type	Tipo de commit seguindo o padrão *Conventional Commits* (fix, feat, chore, ci, etc.).
author_name	Nome do autor responsável pelo commit.
version_raw	Versão do repositório e versão atual do projeto no momento da execução.
pipeline_url	URL de referência da pipeline executada na nuvem.
pipeline_duration	Tempo total de execução da pipeline.
build_size	Tamanho final do pacote gerado pela aplicação.
test_duration	Tempo de execução dos testes unitários.
coverage_line	Percentual de cobertura de testes por linha de código.

Fonte: Elaborado pelo autor.

ser valiosos para análises futuras, especialmente aqueles relacionados à execução de testes automatizados.

Após a coleta, todos os dados são integrados automaticamente a uma planilha do *Google Planilhas*[13] por meio de *scripts*, garantindo padronização e consistência, além de facilitar as etapas posteriores de análise quantitativa.

Para o controle de versão das funcionalidades do projeto o padrão ***Versionamento Semântico (SemVer)***[14] é aplicado, amplamente utilizado e aceito na comunidade de desenvolvimento de software. Esse padrão estabelece diretrizes bem definidas para a organização das versões, o que facilita a evolução incremental do projeto.

A versão *0.3.0* contempla os casos Subseção 4.2.4 e Subseção 4.2.5 por se tratarem

[13] Disponível em: https://workspace.google.com/intl/pt-BR/products/sheets/
[14] Disponível em: https://semver.org/lang/pt-BR/

Quadro 3 – Descrição das versões implementadas no projeto

Versão	Caso de Uso	Detalhe
0.1.0	Acessar página de apresentação	Permite ao usuário compreender o sistema e sua proposta, apresentando informações iniciais sobre a plataforma.
0.2.0	Identificar usuário	Realiza a autenticação e o controle de acesso à plataforma, garantindo que apenas usuários autorizados utilizem o sistema.
0.3.0	Visualizar catálogo e detalhes de um vídeo	Possibilita a exploração de conteúdos, a visualização de detalhes dos vídeos e a realização de interações pelo usuário.
0.4.0	Reproduzir vídeo	Permite a reprodução do conteúdo audiovisual com foco em desempenho e na experiência do usuário.

Fonte: Elaborado pelo autor.

de contextos muito semelhantes, justificando sua inclusão em um único pacote de evolução.

O ***Conventional Commits*** [15] é um padrão amplamente adotado pela comunidade de desenvolvimento, que define regras claras para padronizar mensagens de *commit*. Essa padronização facilita a compreensão da linha do tempo do projeto e melhora a rastreabilidade das alterações no versionamento de código.

Quadro 4 – Estrutura geral de mensagens seguindo o padrão *Conventional Commits*

O formato geral de uma mensagem seguindo esse padrão é:

```
<TIPO_COMMIT>(<CONTEXTO>):  <DESCRIÇÃO DO COMMIT>
```

Tipo do commit: indica a natureza da alteração. Exemplos comuns:
- **feat**: nova funcionalidade;
- **fix**: correção de defeitos;
- **styles**: ajustes de estilo sem alterar lógica;
- **ci**: alterações na configuração de integração/entrega contínua;
- **chore**: melhorias internas ou manutenção do código.

Contexto: refere-se à parte ou módulo da aplicação afetado pela mudança. Exemplo: implementação do player de vídeo ou do catálogo de exibição.

Descrição: detalha de forma objetiva o que foi modificado ou adicionado no commit.

Fonte: Elaborado pelo autor.

Para a análise das métricas de desempenho das aplicações, utilizamos o ***Lighthouse***[16], uma ferramenta de código aberto que permite auditar páginas web e gerar relatórios contendo métricas detalhadas sobre aspectos críticos de performance, acessibilidade e boas práticas.

[15] Disponível em: https://www.conventionalcommits.org/pt-br/v1.0.0/

[16] Disponível em: https://developer.chrome.com/docs/lighthouse/overview?hl=pt-br

Essa ferramenta possibilita compreender como uma aplicação web se comporta durante o carregamento, além de oferecer recomendações de melhorias com base nos pontos de atenção identificados.

No contexto deste estudo, serão gerados relatórios de desempenho para cada módulo implementado. Assim, será possível comparar os resultados entre as arquiteturas monolítica e microfront-ends a cada incremento funcional.

- ***First Contentful Paint* (FCP)** mede o tempo que o navegador leva para renderizar o primeiro conteúdo do *Document Object Model* (DOM) após a navegação do usuário até a página Developers (2025b).
- ***Largest Contentful Paint* (LCP):** mede o momento em que o maior elemento de conteúdo dentro da área visível da página é renderizado. Essa métrica indica quando o conteúdo principal se torna visível para o usuário Developers (2025c).
- ***Total Blocking Time* (TBT):** calcula o tempo total durante o qual a página permanece bloqueada para responder a interações do usuário, como cliques, toques ou teclas pressionadas Developers (2025d).
- ***Cumulative Layout Shift* (CLS):** mede a soma dos deslocamentos inesperados de layout ao longo do ciclo de vida da página, indicando o quanto os elementos visuais se movem durante o carregamento. Developers (2025a).
- **Speed Index:** avalia a rapidez com que o conteúdo visível é carregado durante o processo de renderização da página developers (2025).

4.4.2 Análise de dados

A partir da coleta dos dados, métricas como **tempo de execução das pipelines** e **tamanho dos pacotes** permitem compreender de forma aprofundada o comportamento das duas arquiteturas analisadas. Com essas informações, é possível aplicar modelos estatísticos para obter o tempo médio e o tamanho médio dos pacotes por arquitetura, além de gerar visualizações em gráficos, como gráficos de barras e colunas, que facilitam a interpretação dos resultados e o entendimento dos impactos de cada métrica nas arquiteturas.

Os dados coletados neste estudo são:

- **Tempo médio de execução das pipelines CI/CD**
- **Tempo de execução das pipelines por versão (segundos)**
- **Tamanho médio dos pacotes gerados (KB)**

- **Tamanho dos pacotes por versão (KB)**

A partir dos dados quantitativos coletados e dos objetivos da pesquisa, os critérios de avaliação que atuam como uma régua de comparação entre as arquiteturas monolítica e de microfront-ends são:

- **Escalabilidade das pipelines de CI/CD**: Avaliar a capacidade da arquitetura de sustentar processos eficientes de integração e entrega contínua à medida que o sistema evolui. Relacionado ao objetivo de tempo de execução das pipelines automatizadas.
- **Controle de tamanho do pacote**: Analisar a eficiência da arquitetura no controle do tamanho dos pacotes distribuídos ao cliente final, um fator que impacta no desempenho e acessibilidade, especialmente em redes com conexões lentas. Conectado ao objetivo de analisar como a arquitetura lida com o crescimento dos pacotes ao longo do tempo.
- **Experiência do Usuário (*Web Vitals*)**: Verificar se a adoção da arquitetura de microfront-ends impacta a percepção de desempenho da aplicação pelo usuário final. Relacionado ao objetivo de analisar o desempenho das aplicações por meio das métricas *Web Vitals*.
- **Diversidade Técnica**: Avaliar a flexibilidade da arquitetura em acomodar tecnologias diferentes dentro de uma mesma aplicação, permitindo a adoção de ferramentas mais adequadas aos seus domínios específicos. Analisando a capacidade da arquitetura de suportar múltiplos frameworks e abordagens de desenvolvimento sem comprometer a integração e a coesão do sistema.
- **Simplicidade Operacional**: Examinar o esforço necessário para configurar, manter e operar a arquitetura ao longo do tempo, considerando aspectos como complexidade de orquestração, carga cognitiva e custos iniciais de adoção. Relacionado ao objetivo de eficiência operacional.

5 RESULTADOS

Na Seção 5.1 apresentamos os resultados referentes ao **tempo de execução das pipelines**. Na Seção 5.2 apresentamos os resultados para o **tamanho dos pacotes**. Na Seção 5.3, são exibidas as métricas dos relatórios de **desempenho** gerados pela ferramenta *Lighthouse*. Já na Seção 5.4 é é realizada uma análise detalhada dos resultados, contemplando uma comparação entre as arquiteturas avaliadas e um panorama geral do estudo. Na Seção 5.5 discutimos os resultados obtidos com base nos trabalhos relacionados.

Os repositórios contendo as implementações estão disponíveis publicamente no GitHub da organização ***Moduflix***[1]. A planilha utilizada para a coleta dos dados de análise está publicada e acessível nas referências deste trabalho, conforme (Torquato, 2025).

5.1 Resultados das métricas de tempo de execução das pipelines

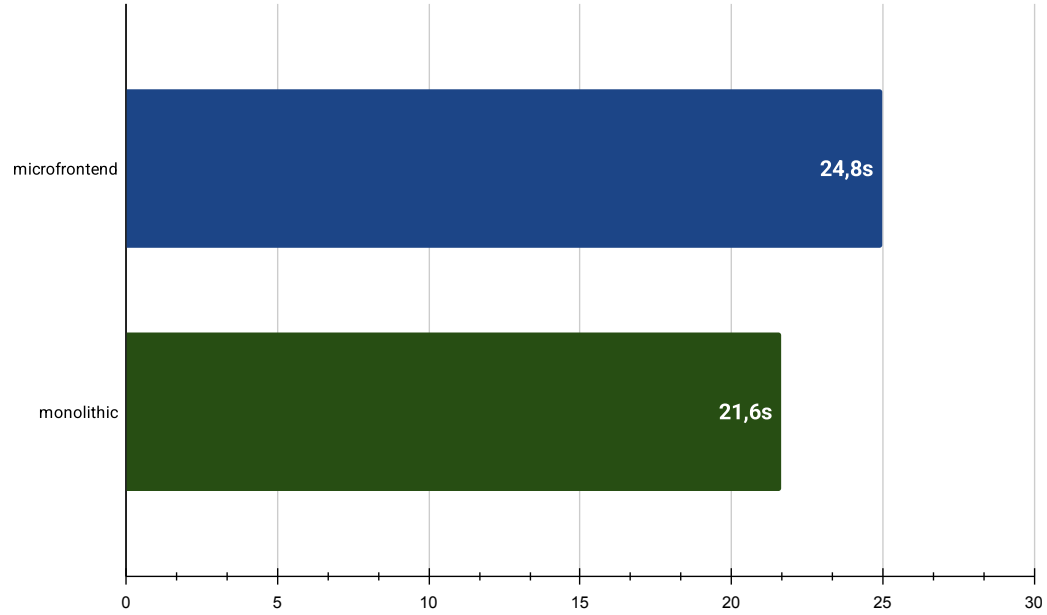
A Figura 8 ilustra o tempo médio, em segundos, de execução das *pipelines* de CI/CD para cada arquitetura avaliada. Nos resultados obtidos neste projeto, a pipeline baseada em microfront-ends apresentou um tempo de execução **14,81% maior** em relação à *pipeline* da arquitetura monolítica.

A Figura 9 ilustra o tempo médio de execução da *pipeline* da arquitetura em microfront-ends, em segundos, para cada versão implementada. Os resultados indicam uma taxa média de **crescimento por versão de 4,08%** no tempo de execução.

A Figura 10 ilustra o tempo médio de execução da *pipeline* da arquitetura monolítica, em segundos, para cada versão implementada. Os resultados mostram que o tempo de execução cresceu de forma progressiva entre as versões, resultando em uma taxa média de **crescimento de 11,79%** a cada nova versão.

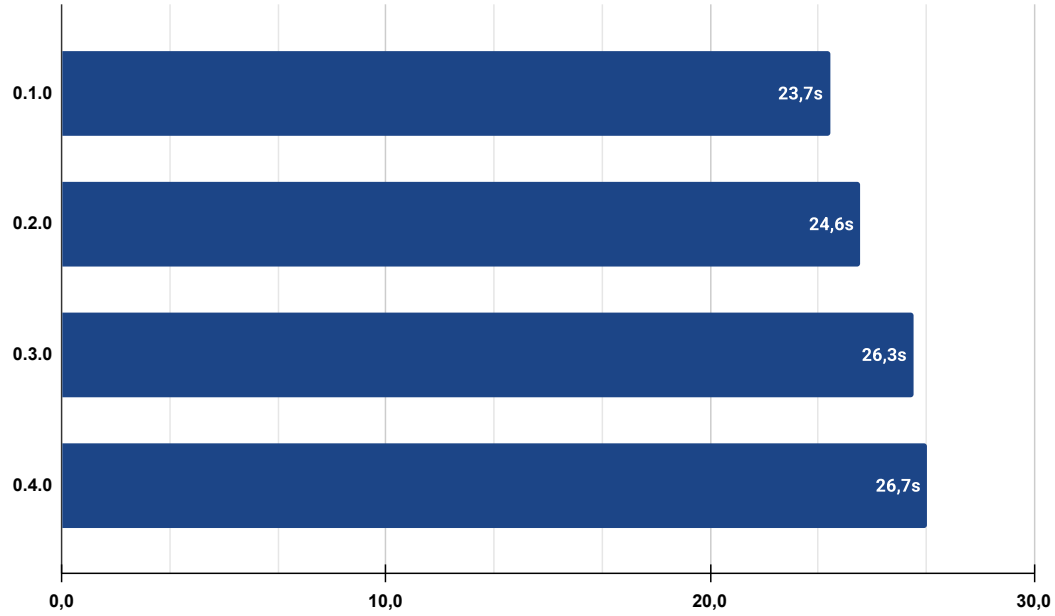
[1] Disponível em: https://github.com/moduflix

Figura 8 – Resultado tempo médio de execução do pipeline CI/CD — Arquitetura microfront-ends vs Arquitetura monolítica



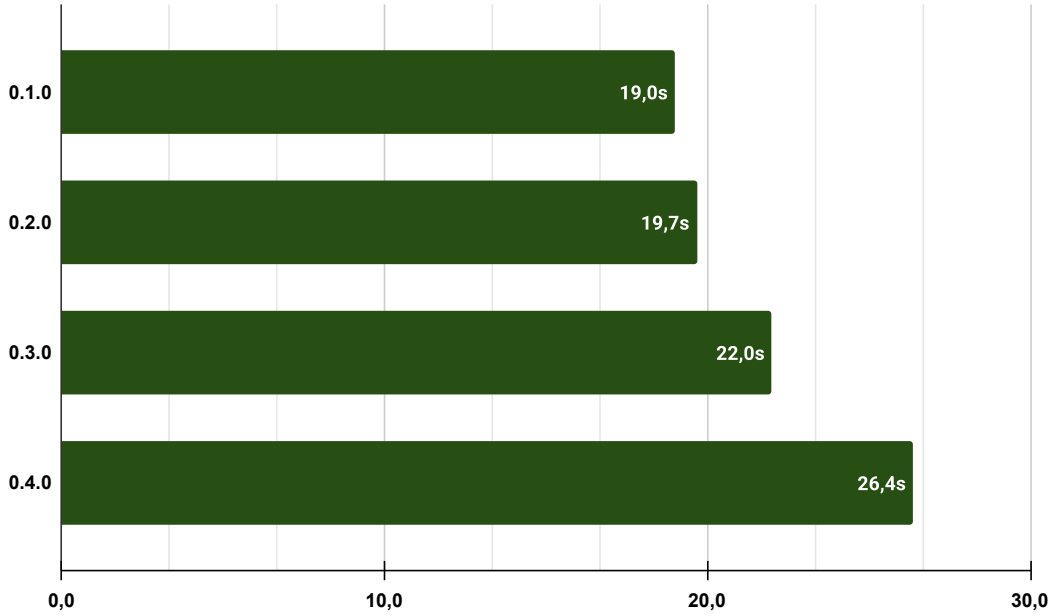
Fonte: Elaborada pelo autor.

Figura 9 – Resultado tempo médio de execução do pipeline CI/CD por versão — Arquitetura microfront-ends



Fonte: Elaborada pelo autor.

Figura 10 – Resultado tempo médio de execução do pipeline CI/CD por versão — Arquitetura monolítica

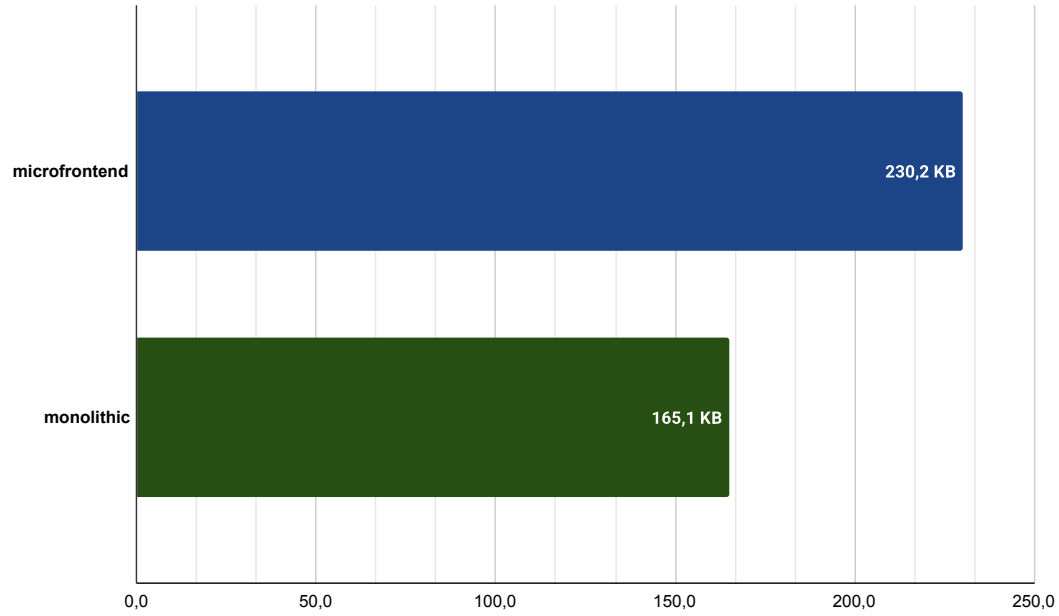


Fonte: Elaborada pelo autor.

5.2 Resultados das métricas de tamanho dos pacotes

A Figura 11 ilustra a média do tamanho dos pacotes, em *Kilobyte* (KB), para as arquiteturas monolítica e em microfront-ends. Os resultados mostram que o tamanho médio dos pacotes da arquitetura em microfront-ends é **39,43% maior** em comparação à arquitetura monolítica.

Figura 11 – Resultado tamanho médio do pacote (KB) - Arquitetura microfront-ends vs Arquitetura monolítica

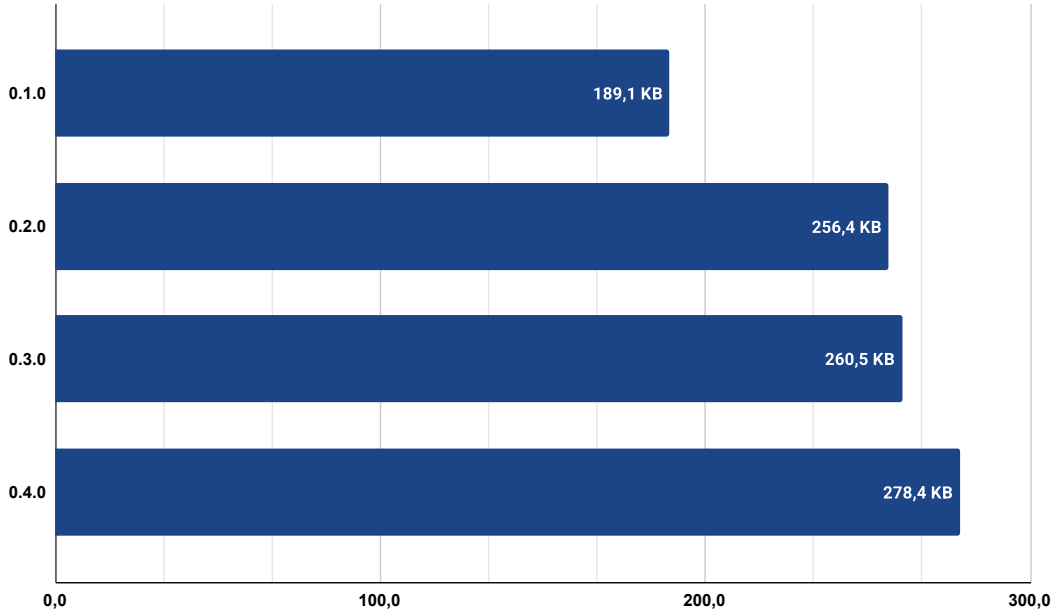


Fonte: Elaborada pelo autor.

A Figura 12 ilustra o tamanho médio dos pacotes das aplicações em microfront-ends, em KB, para cada nova versão implementada. Os resultados indicam que a taxa média de **crescimento por versão foi de 14,68%**.

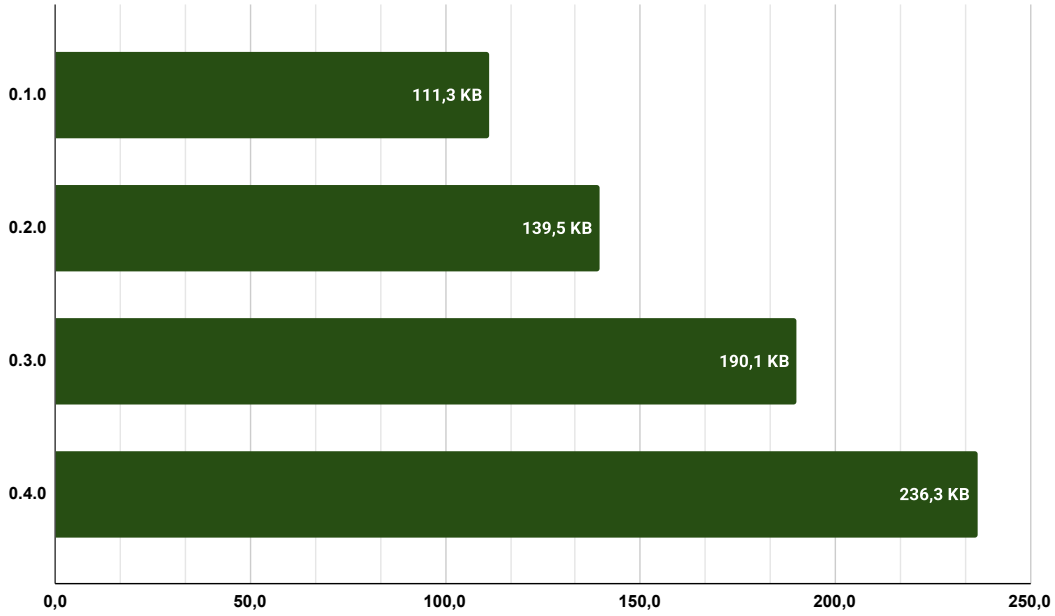
A Figura 13 ilustra o tamanho médio dos pacotes da aplicação com arquitetura monolítica, em KB, para cada versão implementada. Os resultados indicam que a taxa média de **crescimento por versão foi de 28,63%**.

Figura 12 – Resultado tamanho médio do pacote (KB) por versão - Microfront-end



Fonte: Elaborada pelo autor.

Figura 13 – Resultado tamanho médio do pacote (KB) por versão - Arquitetura Monolítica

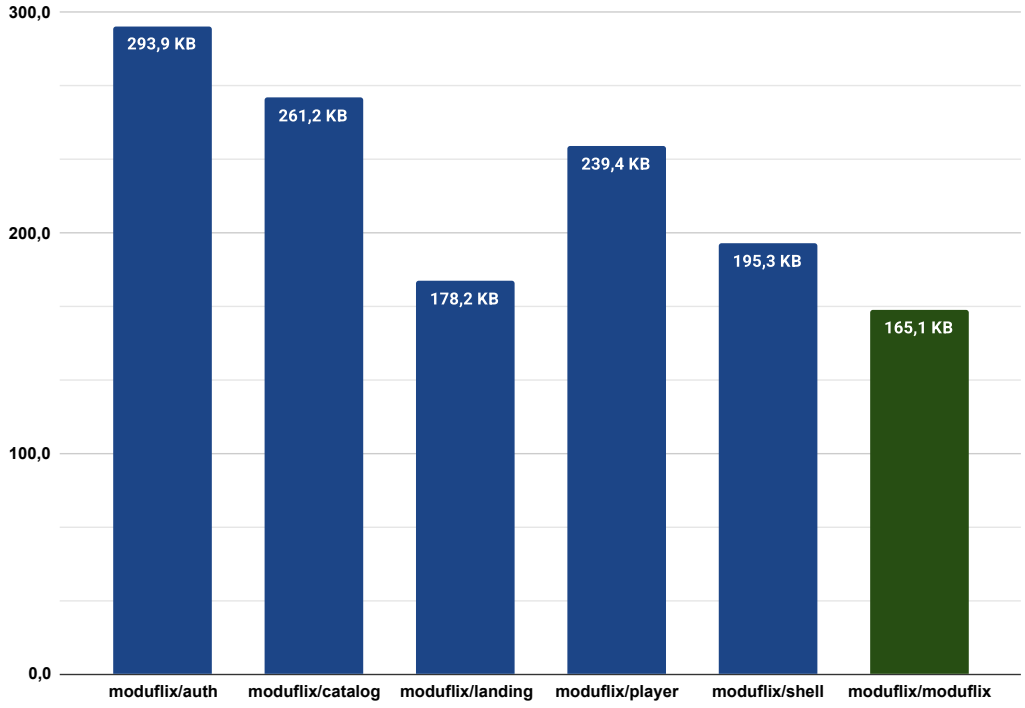


Fonte: Elaborada pelo autor.

A Figura 14 apresenta o tamanho médio dos pacotes, em KB, por repositório das duas arquiteturas implementadas. O repositório *moduflix/auth*, desenvolvido com *React.js*, apresenta o maior tamanho de pacote entre todas as aplicações, superando inclusive os repositórios implementados em *Vue.js*. Os resultados também mostram que a aplicação monolítica possui o menor tamanho médio do pacote em comparação aos demais repositórios. Para os repositórios em *Vue.js* com arquitetura em microfront-ends, a média foi de **218,5 KB**, valor aproximadamente

32,35% maior que o tamanho médio do pacote da arquitetura monolítica.

Figura 14 – Resultado tamanho médio do pacote (KB) por repositório

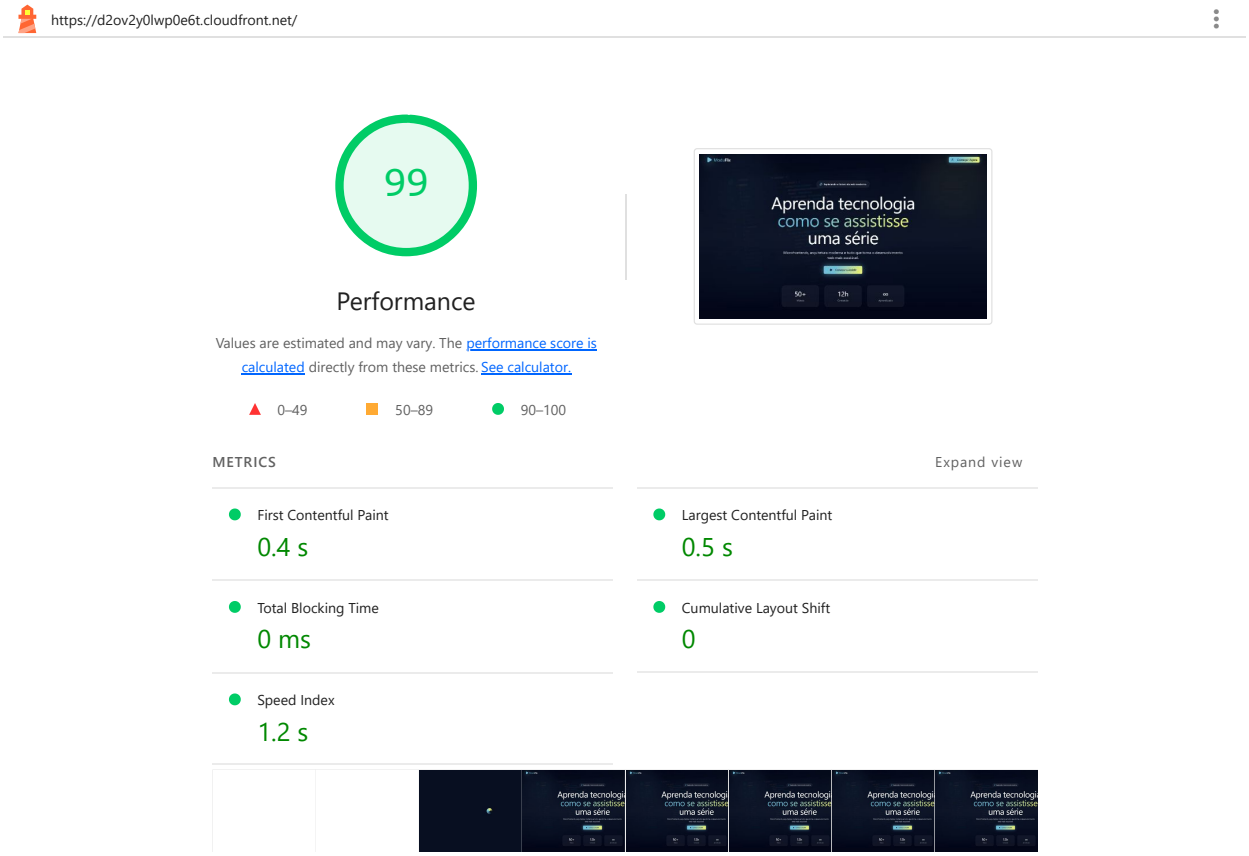


Fonte: Elaborada pelo autor.

5.3 Resultados das métricas de desempenho dos indicadores Web Vitals

A Figura 15 apresenta o módulo MFE de Apresentação avaliado pelas métricas do *Lighthouse*. Os resultados indicam um FCP de **0,4 segundos**, LCP de **0,5 segundos**, TBT de **0 milissegundos** e *Speed Index* de **1,2 segundos**, resultando em uma pontuação de **99/100** em desempenho.

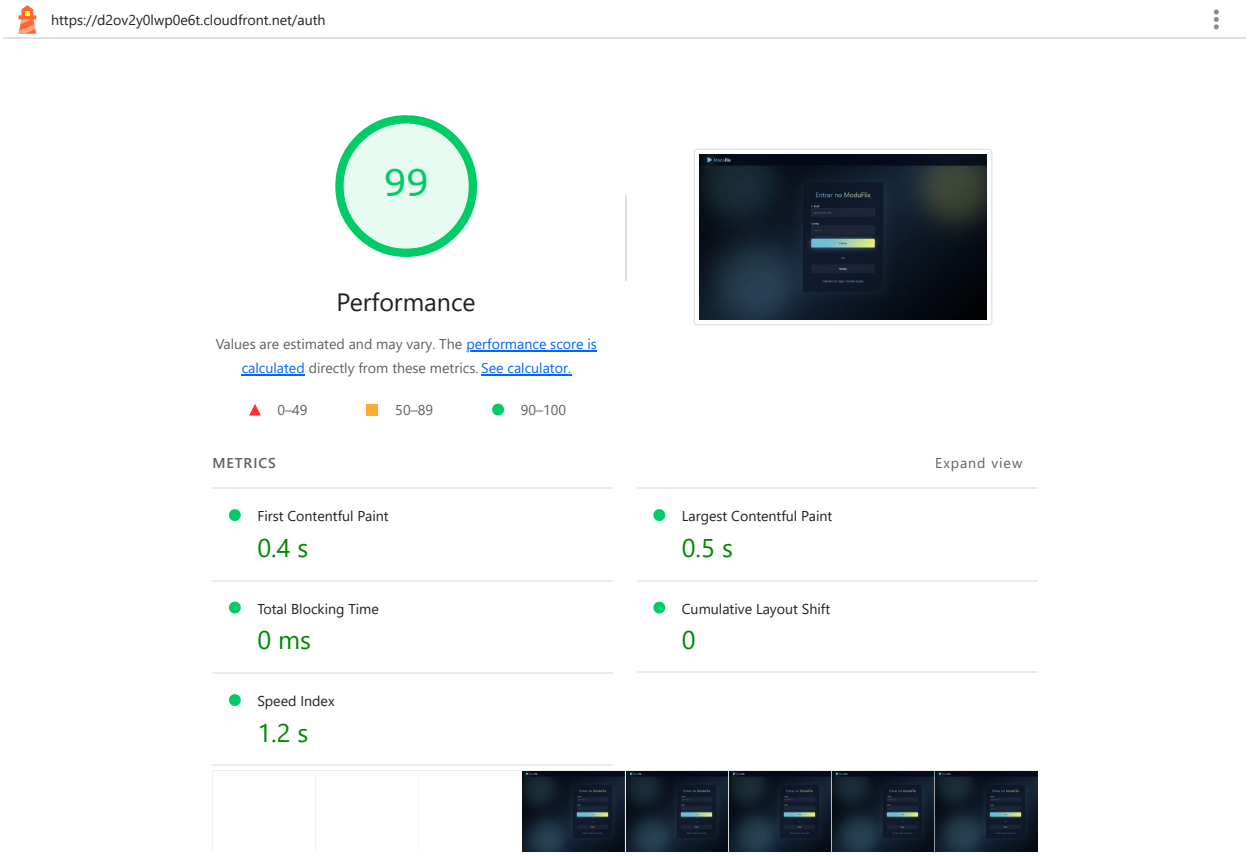
Figura 15 – Relatório de desempenho - MFE Apresentação



Fonte: Elaborada pelo autor.

A Figura 16 apresenta o módulo MFE de Autenticação avaliado pelas métricas do *Lighthouse*. Os resultados indicam um FCP de **0,4 segundos**, LCP de **0,5 segundos**, TBT de **0 milissegundos** e *Speed Index* de **1,2 segundos**, resultando em uma pontuação de **99/100** em desempenho.

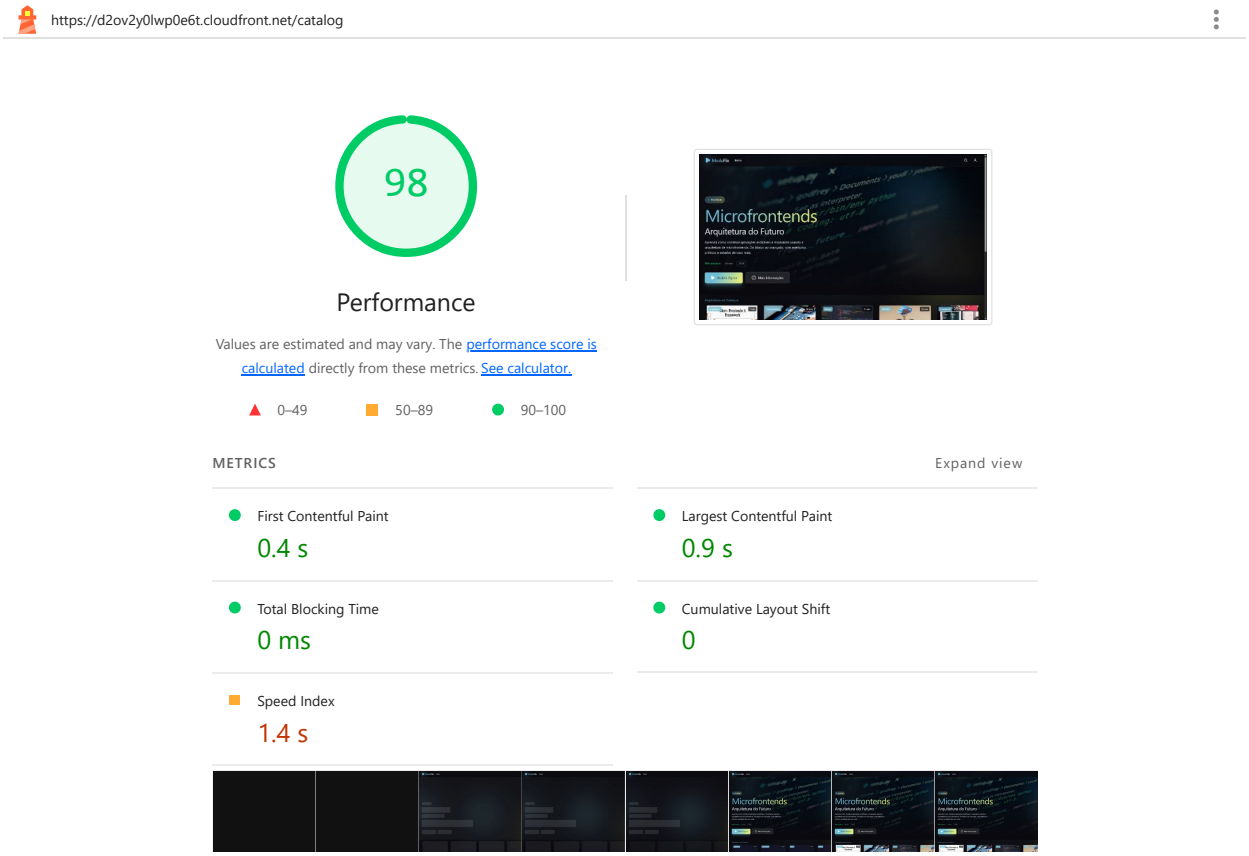
Figura 16 – Relatório de desempenho - MFE Autenticação



Fonte: Elaborada pelo autor.

A Figura 17 apresenta o módulo MFE de Catálogo avaliado pelas métricas do *Lighthouse*. Os resultados indicam um FCP de **0,4 segundos**, LCP de **0,9 segundos**, TBT de **0 milissegundos** e *Speed Index* de **1,4 segundos**, resultando em uma pontuação de **98/100** em desempenho.

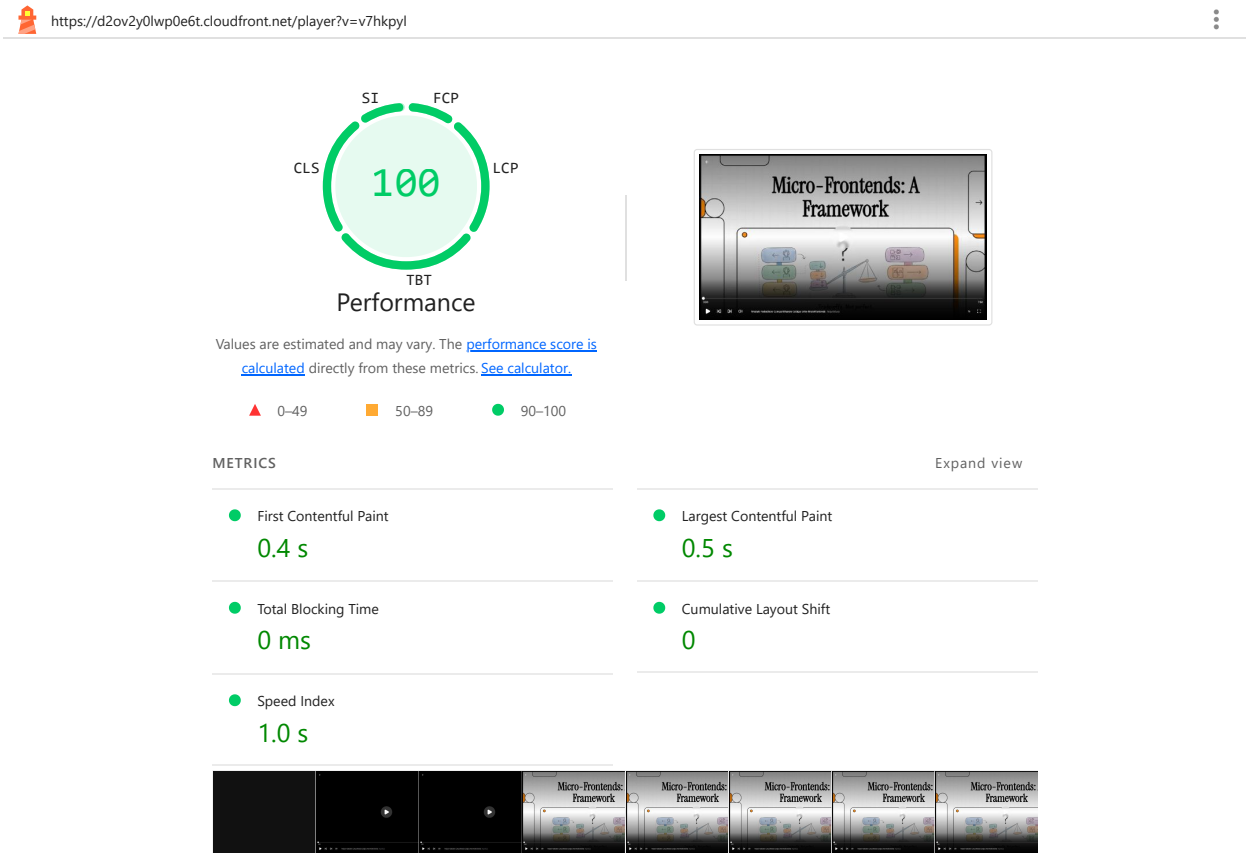
Figura 17 – Relatório de desempenho - MFE Catálogo



Fonte: Elaborada pelo autor.

A Figura 18 apresenta o módulo MFE de Player avaliado pelas métricas do *Lighthouse*. Os resultados indicam um FCP de **0,4 segundos**, LCP de **0,5 segundos**, TBT de **0 milissegundos** e *Speed Index* de **1,0 segundo**, resultando em uma pontuação de **100/100** em desempenho.

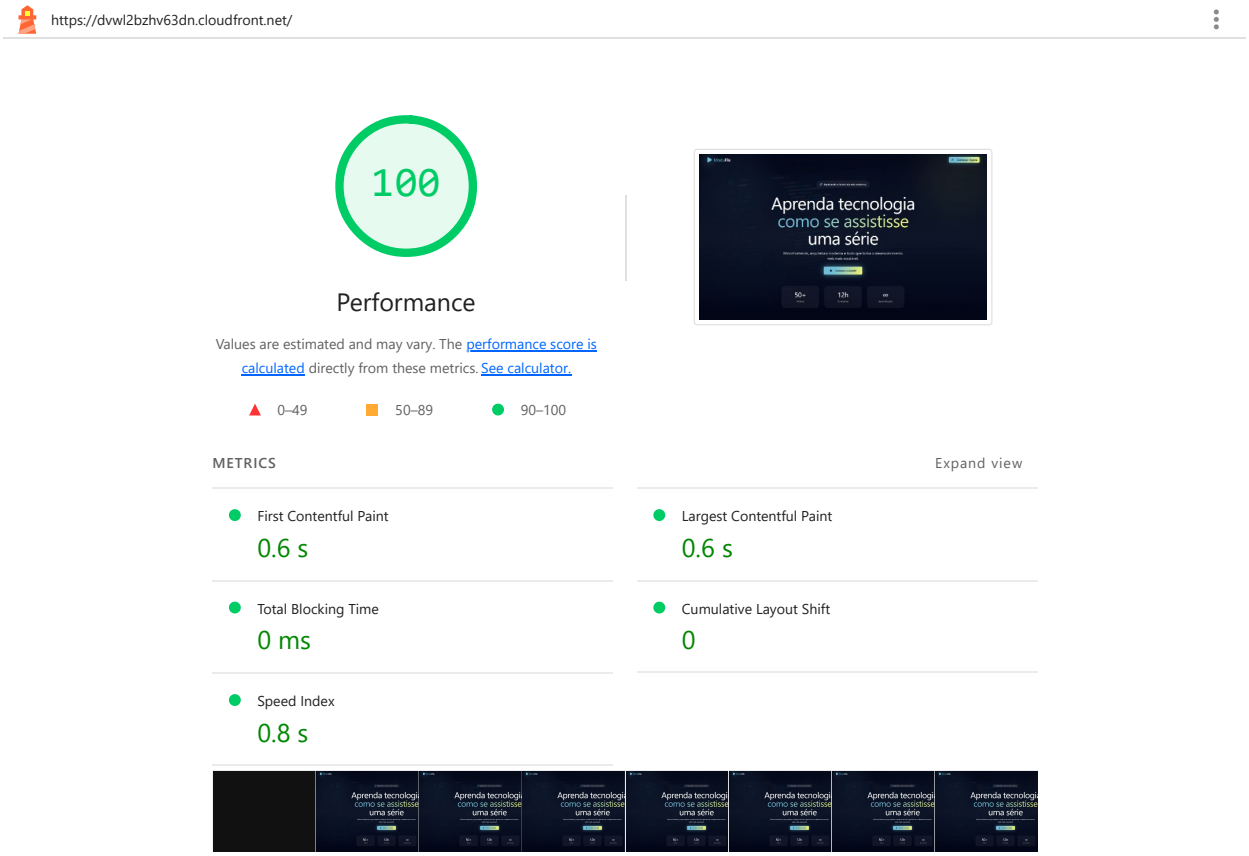
Figura 18 – Relatório de desempenho - MFE Player



Fonte: Elaborada pelo autor.

A Figura 19 apresenta a tela de apresentação da aplicação monolítica avaliada pelas métricas do *Lighthouse*. Os resultados indicam um FCP de **0,6 segundos**, LCP de **0,6 segundos**, TBT de **0 milissegundos** e *Speed Index* de **0,8 segundos**, resultando em uma pontuação de **100/100** em desempenho.

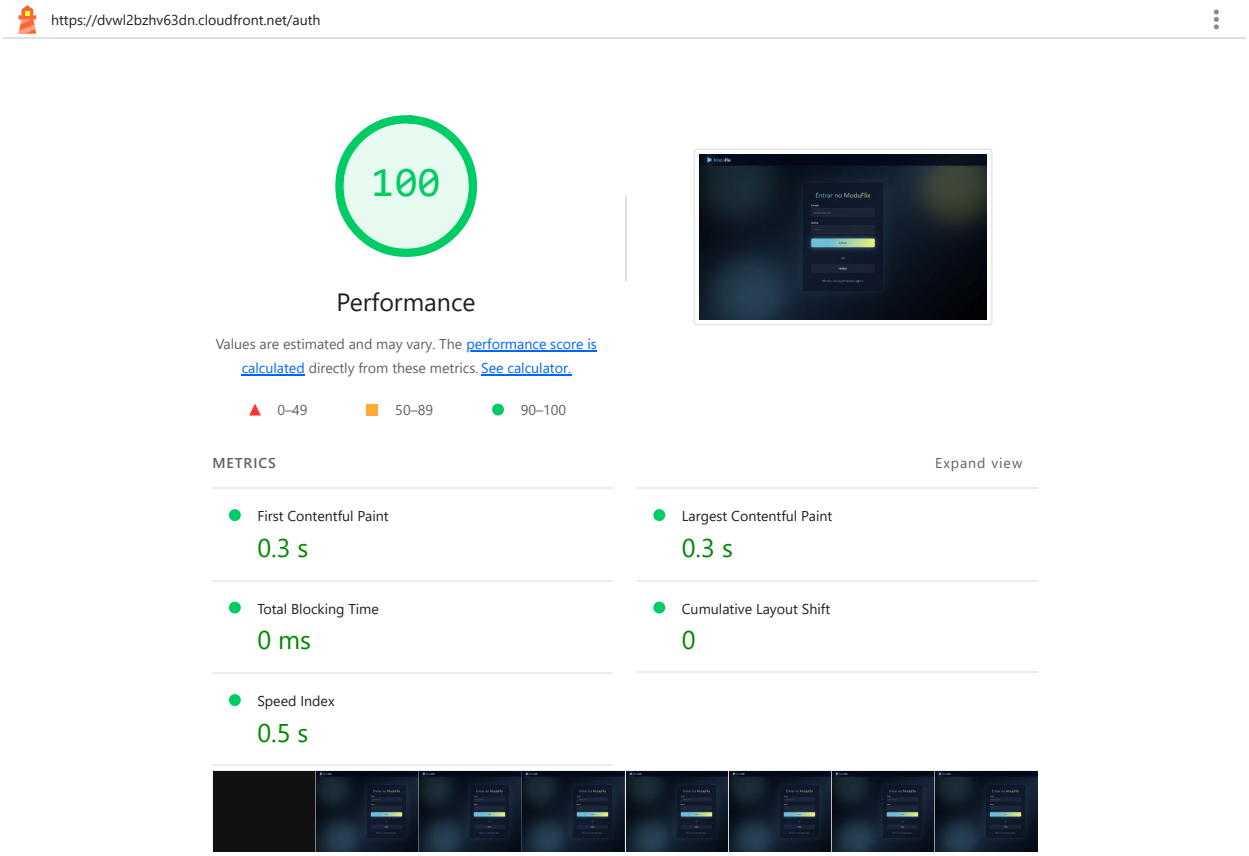
Figura 19 – Relatório de desempenho arquitetura monolítica - Apresentação



Fonte: Elaborada pelo autor.

A Figura 20 apresenta a tela de autenticação da aplicação monolítica avaliada pelas métricas do *Lighthouse*. Os resultados indicam um FCP de **0,3 segundos**, LCP de **0,3 segundos**, TBT de **0 milissegundos** e *Speed Index* de **0,5 segundos**, resultando em uma pontuação de **100/100** em desempenho.

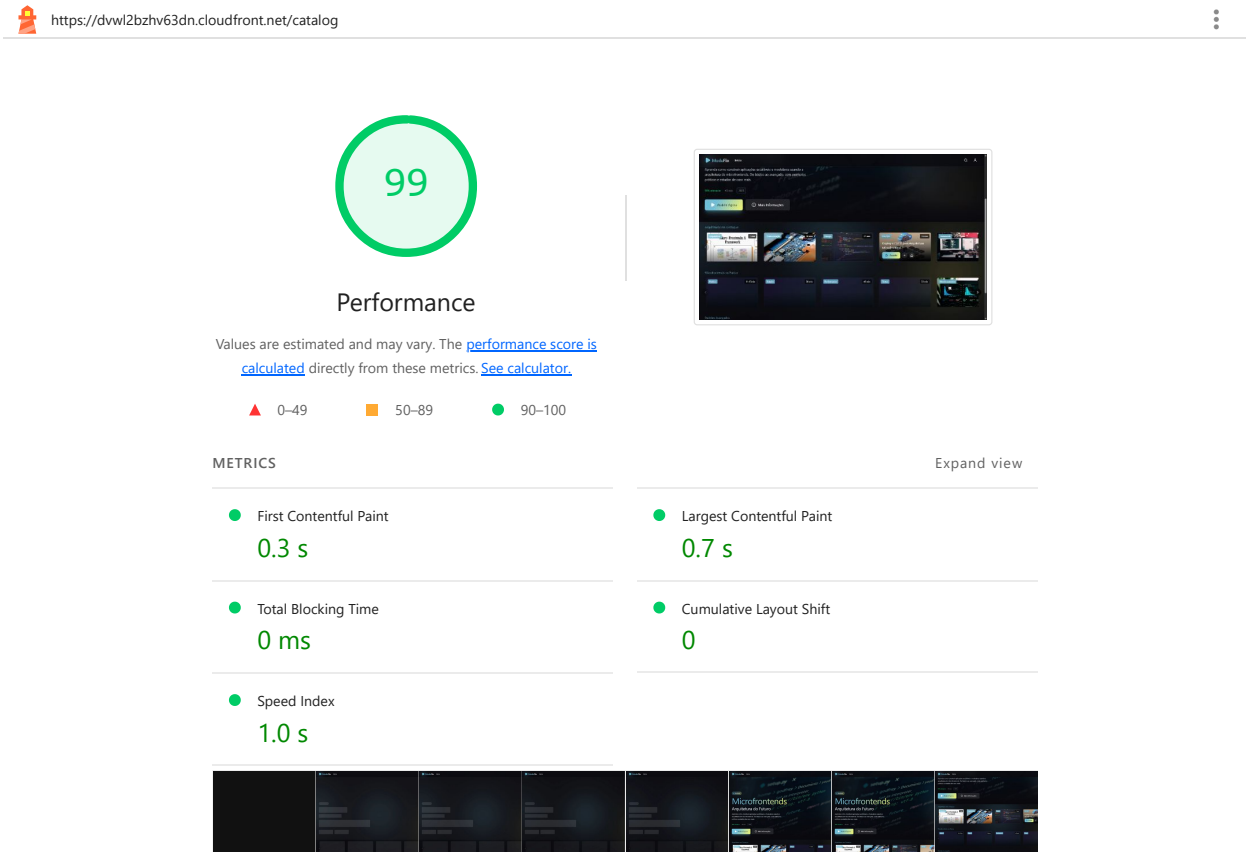
Figura 20 – Relatório de desempenho arquitetura monolítica - Autenticação



Fonte: Elaborada pelo autor.

A Figura 21 apresenta a tela de Catálogo da aplicação monolítica avaliada pelas métricas do *Lighthouse*. Os resultados indicam um FCP de **0,3 segundos**, LCP de **0,7 segundos**, TBT de **0 milissegundos** e *Speed Index* de **1,0 segundo**, resultando em uma pontuação de **99/100** em desempenho.

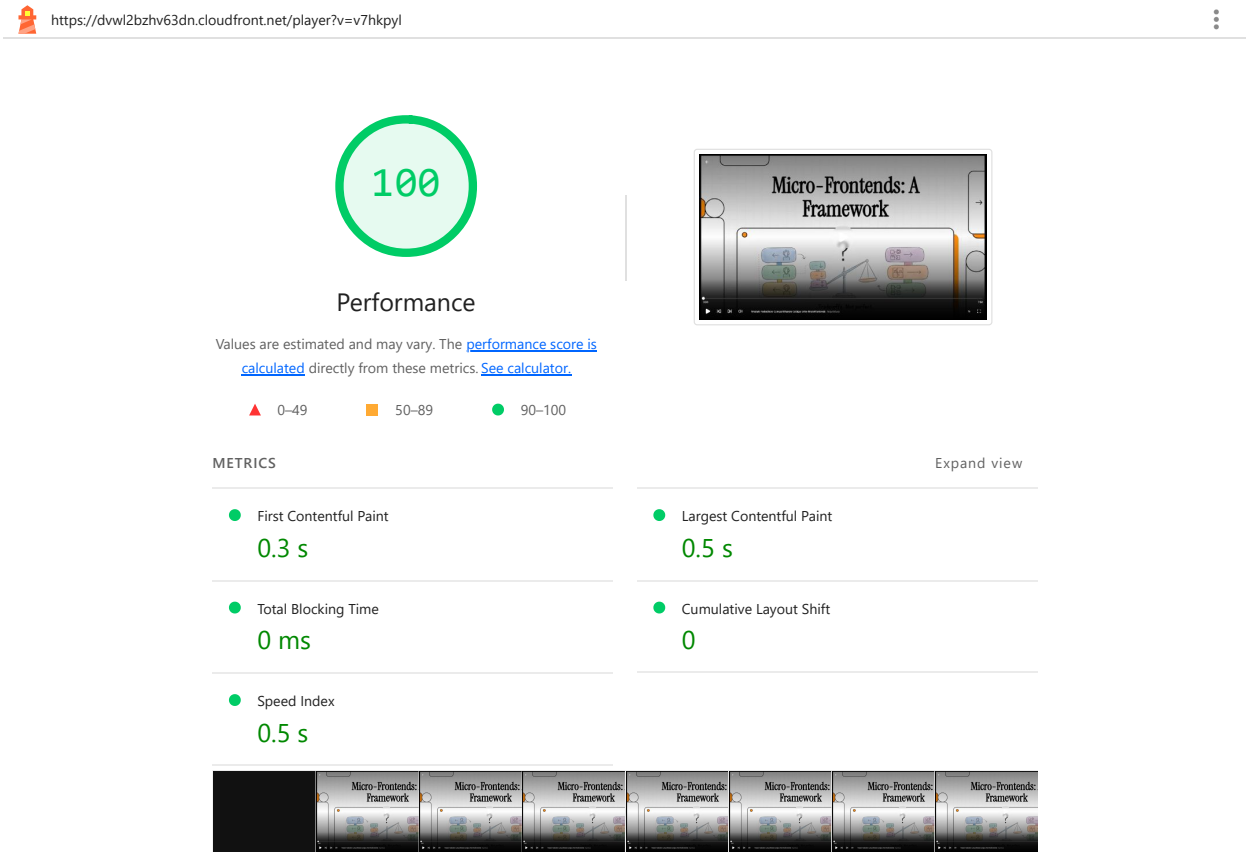
Figura 21 – Relatório de desempenho arquitetura monolítica - Catálogo



Fonte: Elaborada pelo autor.

A Figura 22 apresenta a tela de Player da aplicação monolítica avaliada pelas métricas do *Lighthouse*. Os resultados indicam um FCP de **0,3 segundos**, LCP de **0,5 segundos**, TBT de **0 milissegundos** e *Speed Index* de **0,5 segundos**, resultando em uma pontuação de **100/100** em desempenho.

Figura 22 – Relatório de desempenho arquitetura monolítica - Player



Fonte: Elaborada pelo autor.

A Tabela 1 apresenta uma visão geral de todos os relatórios capturados, bem como seus respectivos valores, organizados de forma comparativa.

Tabela 1 – Comparação das métricas Lighthouse entre arquitetura microfront-ends e arquitetura monolítica

Módulo	**Arquitetura**	**Pontos**	**FCP (s)**	**LCP (s)**	**TBT**	**CLS**	**Speed Index (s)**
Apresentação	MFE	99	0.4	0.5	0	0	1.2
	Monolítica	100	0.6	0.6	0	0	0.8
Autenticação	MFE	99	0.4	0.5	0	0	1.2
	Monolítica	100	0.3	0.3	0	0	0.5
Catálogo	MFE	98	0.4	0.9	0	0	1.4
	Monolítica	99	0.3	0.7	0	0	1.0
Player	MFE	100	0.4	0.5	0	0	1.0
	Monolítica	100	0.3	0.5	0	0	0.5

Fonte: Elaborado pelo autor.

5.4 Análise dos Resultados

Um aspecto importante a ser considerado é a necessidade de orquestração durante a implantação dos módulos em microfront-ends. Cada módulo exige configuração individual, o que aumenta a carga cognitiva e a complexidade do processo. Ou seja, para cada microfront-end, além da configuração de integração contínua, é necessário definir um fluxo de deploy independente e validar se o módulo está funcionando corretamente e integrado aos demais.

Esse cenário contrasta com uma aplicação monolítica, que possui apenas um único módulo, tornando o fluxo de integração e entrega contínua substancialmente mais simples.

Segundo os resultados obtidos no estudo comparativo entre o tempo de execução das *pipelines* e o tamanho dos pacotes, a arquitetura em microfront-ends apresentou desempenho inferior em comparação à arquitetura monolítica. A média do tempo de execução das *pipelines* foi ligeiramente maior, assim como o tamanho dos pacotes gerados.

No entanto, um ponto relevante emerge ao analisar a evolução das versões. **A taxa média de crescimento do tempo de execução das *pipelines* na aplicação em microfront-ends foi de 4,08%, enquanto na arquitetura monolítica alcançou 11,79%**. Para o tamanho dos pacotes, a arquitetura em microfront-ends apresentou uma variação média de **14,68%**, ao passo

que a aplicação monolítica apresentou **28,63%** de crescimento, conforme ilustrado nas Figuras 9, 10, 12 e 13.

A taxa média de crescimento do tempo de execução e do tamanho do pacote foi quase o dobro comparada à aplicação monolítica; ou seja, com o crescimento de uma base de código monolítica a longo prazo e na manutenção de uma grande aplicação, os atributos podem favorecer as aplicações em microfront-ends, já que o crescimento da aplicação é quase metade da velocidade de crescimento dos atributos analisados.

Esses resultados mostram que a taxa média de crescimento do tempo de execução e do tamanho do pacote foi quase o dobro na aplicação monolítica; ou seja, à medida que a base de código monolítica cresce a longo prazo, os custos operacionais tendem a aumentar mais rapidamente. Nesse cenário, a arquitetura em microfront-ends pode se tornar mais vantajosa, já que apresenta crescimento significativamente menor nesses atributos.

Com as métricas coletadas pelo *Lighthouse*, é possível observar que **o desempenho da aplicação monolítica foi levemente superior ao da arquitetura em microfront-ends, apresentando uma diferença de apenas 1 ponto**. Enquanto a arquitetura monolítica obteve 99/100, a arquitetura em microfront-ends alcançou 98/100, como no caso da implementação do catálogo. No entanto, em um contexto de uso real, essa diferença é praticamente imperceptível, não representando um impacto significativo na experiência do usuário ao comparar ambas as arquiteturas.

Para projetos de pequeno porte ou com equipes reduzidas, a implementação dessa arquitetura pode caracterizar um cenário de *overengineering*, projetando uma solução de forma mais complexa do que o necessário para o objetivo de seu uso, uma vez que os custos operacionais e a carga cognitiva necessários para gerenciar múltiplas pipelines e repositórios independentes superam os benefícios técnicos imediatos.

O Quadro 5 apresenta a comparação entre a Arquitetura Monolítica e a Arquitetura de Microfront-ends com base nos critérios técnicos, operacionais e de desempenho definidos neste estudo. Cada critério foi avaliado por meio de uma escala ordinal de **1 a 5**, na qual **1 representa desempenho ou aderência muito baixa ao critério avaliado e 5 representa desempenho ou aderência muito alta**, considerando os objetivos do trabalho. Essa escala permite sintetizar os resultados quantitativos e qualitativos observados durante a execução, fornecendo uma visão comparativa clara da adequação de cada arquitetura aos atributos de qualidade analisados.

Quadro 5 – Comparação entre arquitetura monolítica e microfront-ends

Critério de comparação	**Monolítica**	**MFE**	**Justificativa baseada nos dados**
Escalabilidade das pipelines de CI/CD	2	5	O crescimento do tempo de execução das pipelines no MFE foi significativamente menor (4,08%) em comparação à arquitetura monolítica (11,79%).
Controle de tamanho do pacote	2	5	A taxa de crescimento do pacote no MFE (14,68%) foi consideravelmente inferior à observada na arquitetura monolítica (28,63%), indicando melhor controle de evolução do pacote.
Experiência do usuário (*Web Vitals*)	5	5	Ambas as arquiteturas mantiveram pontuações elevadas e próximas nos indicadores de desempenho percebido, sem impacto relevante na experiência do usuário.
Diversidade técnica	1	5	O MFE permitiu a coexistência de diferentes tecnologias no mesmo sistema, enquanto a arquitetura monolítica impôs padronização tecnológica.
Simplicidade operacional	5	2	A arquitetura monolítica apresentou menor complexidade operacional, enquanto o MFE demandou maior esforço de orquestração e gestão de pipelines.
Síntese comparativa	**–**	**–**	**A arquitetura de microfront-ends apresentou melhor aderência aos critérios de escalabilidade e flexibilidade, enquanto a monolítica se destacou pela simplicidade operacional.**

Fonte: Elaborado pelo autor.

5.5 Discussão dos Resultados

Com base no estudo de Hidayat *et al.* (2024), a adoção da arquitetura em microfront-ends pode trazer ganhos de eficiência e redução no tempo de execução das *pipelines* de CI/CD. No entanto, no contexto deste trabalho, esse comportamento não se repetiu: o tempo médio de execução das *pipelines* na arquitetura em microfront-ends apresentou-se inferior a da implementação monolítica, como ilustrado na Figura 8.

No estudo de Peltonen *et al.* (2021), um dos problemas destacados na adoção de microfront-ends é a duplicação de código entre módulos. Esse fenômeno também foi identificado neste trabalho, já que o tamanho médio dos pacotes das aplicações em microfront-ends foi superior ao da aplicação monolítica. Entretanto, é possível que esse aumento também esteja relacionado ao uso da biblioteca *Module Federation*, que pode impactar o tamanho final dos pacotes gerados por módulo. Essa relação exige uma investigação mais aprofundada.

6 CONCLUSÕES

Este estudo buscou validar, por meio de um estudo análise comparativa de caráter empírico, se a adoção da arquitetura de microfront-ends pode ser considerada uma evolução arquitetural sustentável frente à arquitetura monolítica tradicional. Por meio do desenvolvimento da aplicação de *streaming Moduflix*, analisando as dimensões de **eficiência de CI/CD**, **tamanho de pacotes** e **desempenho a partir dos indicadores *Web Vitals***.

Os resultados confirmam o princípio de Brooks Jr (1995) de que "não existe bala de prata"no desenvolvimento de *software*. No curto prazo, a arquitetura monolítica demonstrou ser superior em agilidade operacional inicial, apresentando uma configuração de *pipeline* centralizada, pacotes 39,43% menores e processos de integração e entrega contínua mais simples. No entanto, esta simplicidade inicial é acompanhada ao longo da evolução do projeto, onde o crescimento da complexidade é acelerado a cada nova funcionalidade.

Ao analisar a evolução dos atributos técnicos das pipelines, foi possível identificar o ritmo de crescimento dessas métricas. Os dados quantitativos revelam que:

- **A taxa média de crescimento no tempo de execução das pipelines na arquitetura monolítica foi de 11,79%, enquanto no MFE foi de apenas 4,08%.**
- **O tamanho dos pacotes da arquitetura monolítica expandiu-se a um ritmo de 28,63% por versão, em contraste com os 14,68% observados no MFE.**

Quanto ao desempenho percebido a partir das métricas *Web Vitals*, mensurado via *Lighthouse*; enquanto a arquitetura monolítica obteve pontuações levemente superiores, tais variações **são praticamente imperceptíveis para o usuário final em cenários reais**, sugerindo que a escolha por microfront-ends não sacrifica a experiência de navegação.

Em suma, a adoção de microfront-ends é justificada quando o ciclo de vida previsto para a aplicação é extenso. O investimento inicial em complexidade estrutural e ferramentas de integração e entrega contínuas é compensado pela previsibilidade e sustentabilidade do crescimento do sistema, tornando-o resiliente à expansão organizacional e técnica que costuma tornar monólitos insustentáveis ao longo do tempo.

6.1 Limitações do Trabalho

Houve algumas limitações na execução deste estudo, principalmente relacionadas às restrições de tempo para o desenvolvimento do projeto, o que resultou em um cronograma de

execução bastante reduzido. Além disso, o escopo do projeto foi relativamente limitado, o que restringiu a quantidade de versões analisadas e, consequentemente, a profundidade da avaliação dos resultados obtidos. Uma investigação mais abrangente, com um número maior de iterações e versões, poderia produzir resultados distintos ou mais representativos. Em complemento, o fato de o trabalho ter sido desenvolvido de forma individual, limita a exploração de um dos principais benefícios destacados pela literatura sobre arquiteturas de *Microfront-ends*: a capacidade de distribuição do desenvolvimento entre múltiplas equipes de forma independente e paralela.

6.2 Trabalhos futuros

A partir dos resultados obtidos e dos aspectos analisados relacionados à adoção da arquitetura de microfront-ends, algumas sugestões de contribuições para trabalhos futuros podem ser destacadas:

- **Expansão da diversidade tecnológica**: Embora este estudo tenha demonstrado a aplicação de ferramentas como *Vue.js* e *React.js*, investigações futuras poderiam ampliar o escopo para incluir outras tecnologias ou versões distintas de uma mesma ferramenta. Permitindo, de forma mais abrangente, validar a proposta de independência tecnológica, bem como a viabilidade de migrações incrementais em cenários com maior diversidade tecnológica.
- **Análise aprofundada de testes unitários e de ponta a ponta (E2E)**: Apesar de este trabalho ter estabelecido uma base sólida ao integrar a execução de testes unitários automatizados nas pipelines de integração contínua, os dados brutos relacionados ao tempo de execução dos testes e à cobertura de código não foram explorados de maneira quantitativa. Estudos futuros poderiam investigar de forma mais precisa os impactos da arquitetura em termos de testabilidade, isolamento de domínios e complexidade na implementação de testes de ponta a ponta.
- **Avaliação dos custos financeiros e operacionais da arquitetura**: Seria pertinente quantificar o impacto financeiro da infraestrutura em nuvem necessária para sustentar múltiplos repositórios, pipelines e distribuições independentes. Além disso, estudos complementares poderiam mensurar os custos de manutenção humana, avaliando se compensa o aumento da carga cognitiva e do esforço de orquestração exigidos para configurar, manter e validar múltiplos fluxos de *deploy* e integração.

REFERÊNCIAS

BROOKS JR, F. P. **The Mythical Man-Month**: Essays on software engineering. 2. ed. Boston, MA: Addison Wesley, 1995.

CONWAY, M. E. How do committees invent? **Datamation**, v. 14, n. 4, p. 28–31, 1968.

DEVELOPERS, G. C. **Cumulative layout shift (CLS)**. 2025. Disponível em: https://web.dev/articles/cls?utm_source=lighthouse&utm_medium=devtools&hl=pt-br. Acesso em: 21 nov. 2025.

DEVELOPERS, G. C. **First contentful paint (FCP)**. 2025. Disponível em: https://developer.chrome.com/docs/lighthouse/performance/first-contentful-paint?utm_source=lighthouse&utm_medium=devtools&hl=pt-br. Acesso em: 21 nov. 2025.

DEVELOPERS, G. C. **Largest contentful paint (LCP)**. 2025. Disponível em: https://developer.chrome.com/docs/lighthouse/performance/lighthouse-largest-contentful-paint?utm_source=lighthouse&utm_medium=devtools&hl=pt-br. Acesso em: 21 nov. 2025.

DEVELOPERS, G. C. **Total blocking time (TBT)**. 2025. Disponível em: https://developer.chrome.com/docs/lighthouse/performance/lighthouse-total-blocking-time?utm_source=lighthouse&utm_medium=devtools&hl=pt-br. Acesso em: 21 nov. 2025.

DEVELOPERS, G. chrome. **Speed Index**. 2025. Disponível em: https://developer.chrome.com/docs/lighthouse/performance/speed-index?utm_source=lighthouse&utm_medium=devtools&hl=pt-br. Acesso em: 21 nov. 2025.

FOWLER, M. **Micro Frontends**. 2019. Disponível em: https://martinfowler.com/articles/micro-frontends.html. Acesso em: 21 jul. 2025.

HIDAYAT, D. C.; ATMAJA, I. K. J.; SARASVANANDA, I. B. G. Analysis and comparison of micro frontend and monolithic architecture for web applications. **Jurnal Galaksi**, v. 1, p. 92–100, 8 2024. ISSN 3048-2399. Disponível em: https://ejournal.pancawidya.or.id/index.php/galaksi/article/view/19.

IBM. **Microservices – An architectural approach**. 2021. Disponível em: https://www.ibm.com/think/topics/microservices. Acesso em: 21 ago. 2025.

KAUSHIK, N.; KUMAR, H.; RAJ, V. Micro frontend based performance improvement and prediction for microservices using machine learning. **Journal of Grid Computing**, Springer Science and Business Media B.V., v. 22, 6 2024. ISSN 15729184. Disponível em: https://link.springer.com/article/10.1007/s10723-024-09760-8https://www.researchgate.net/publication/379869365_Micro_Frontend_Based_Performance_Improvement_and_Prediction_for_Microservices_Using_Machine_Learning.

MEZZALIRA, L. **Building Micro-Frontends**. [*S. n.*], 2021. ISBN 9781492082996. Disponível em: https://www.oreilly.com/library/view/building-micro-frontends/9781492082989/.

NEWMAN, S. **Building microservices**. O'Reilly, 2015. ISBN 9781491950357. Disponível em: https://www.oreilly.com/library/view/building-microservices/9781491950340/.

PELTONEN, S.; MEZZALIRA, L.; TAIBI, D. Motivations, benefits, and issues for adopting Micro-Frontends: A multivocal literature review. **Inf. Softw. Technol.**, Elsevier BV, v. 136, n. 106571, p. 106571, ago. 2021.

RAYMOND, E. S. **The Cathedral and the Bazaar**: Musings on linux and open source by an accidental revolutionary. Sebastopol, CA: O'Reilly Media, 1999. ISBN 978-1-56592-724-7.

STOERMER, C.; BACHMANN, F.; VERHOEF, C. **SACAM**: The software architecture comparison analysis method. [*S. l.*]: Carnegie Mellon University, 2003.

TORQUATO, A. **Planilha de dados da pesquisa ModuFlix**. 2025. Disponível em: https://docs.google.com/spreadsheets/d/15xATfFcgRzCyHdctxgVOo2FfIgdGjRgA9cdp54D0eLQ/. Acesso em: 06 dez. 2025.

VALENTE, M. T. de O. **Engenharia de software moderna**. Brasil: Independente, 2020. ISBN 978-65-00-01950-6.