

Hiago de Oliveira Mendes e Lucas Sales Salvo Petrucci

Uma Análise Comparativa entre *Client-Side Rendering* e *Server-Side Rendering* em Aplicações Web

Campos dos Goytacazes-RJ

Outubro de 2025

Biblioteca Anton Dakitsch
CIP - Catalogação na Publicação

M538a

Mendes, Hiago de Oliveira
Uma Análise Comparativa entre Client-Side Rendering e Server-Side
Rendering em Aplicações Web
/ Hiago de Oliveira Mendes, Lucas Sales Salvo Petrucci - 2025.
117 f.

Orientador: Ronaldo Amaral Santos

Trabalho de conclusão de curso (graduação) -- Instituto Federal de
Educação, Ciência e Tecnologia Fluminense, Campus Campos Centro,
Curso de Bacharelado em Sistemas de Informação, Campos dos Goytacazes,
RJ, 2025.

Referências: f. 111 a 117.

1. Renderização no Lado do Cliente (CSR). 2. Renderização no Lado do
Servidor (SSR). 3. Core Web Vitals. 4. Desempenho Web. 5. Experiência
do Usuário (UX). I. Petrucci, Lucas Sales Salvo. II. Santos, Ronaldo
Amaral, orient. III. Título.

Hiago de Oliveira Mendes e Lucas Sales Salvo Petrucci

Uma Análise Comparativa entre *Client-Side Rendering* e *Server-Side Rendering* em Aplicações Web

Trabalho de Conclusão apresentado ao curso Bacharelado em Sistemas de Informação do Instituto Federal de Educação, Ciência e Tecnologia Fluminense, como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Instituto Federal de Educação, Ciência e Tecnologia Fluminense

Orientador: Prof. M.Sc. Ronaldo Amaral Santos

Campos dos Goytacazes-RJ

Outubro de 2025

Hiago de Oliveira Mendes e Lucas Sales Salvo Petruci

Uma Análise Comparativa entre *Client-Side Rendering* e *Server-Side Rendering* em Aplicações Web

Trabalho de Conclusão apresentado ao curso Bacharelado em Sistemas de Informação do Instituto Federal de Educação, Ciência e Tecnologia Fluminense, como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Campos dos Goytacazes-RJ, 07 de Outubro de 2025.

Prof. M.Sc. Ronaldo Amaral Santos (orientador)
Instituto Federal Fluminense (IFF)

Prof. D.Sc. Fernando Luiz de Carvalho e Silva
Instituto Federal Fluminense (IFF)

Prof. M.Sc. Ana Silvia Ribeiro Escocard Santiago
Instituto Federal Fluminense (IFF)

Campos dos Goytacazes-RJ
Outubro de 2025

Agradecimentos

Em primeiro lugar, agradecemos a Deus, por nos dar força, fé e sabedoria para superar todos os desafios e dificuldades ao longo de nossa jornada acadêmica.

Agradecemos de maneira profunda e especial às nossas famílias, nossos alicerces. Em especial às nossas mães, que sempre nos apoiaram com amor incondicional em cada passo desta jornada. Somos imensamente gratos por todo o sacrifício, carinho e dedicação. Em particular, um de nós presta uma homenagem carinhosa à sua bisavó, cuja memória e legado serviram como uma silenciosa e constante fonte de inspiração. Se hoje celebramos esta conquista, é porque tivemos o privilégio de contar com o suporte de vocês.

Ao nosso orientador, Prof. Me. Ronaldo Amaral Santos, expressamos nossa mais profunda gratidão. Seu apoio, orientação precisa e ensinamentos foram fundamentais não apenas para a condução deste trabalho, mas para nosso crescimento como pesquisadores. Agradecemos por sua paciência, dedicação e pelo conhecimento compartilhado.

Gostaríamos de registrar um agradecimento especial ao Prof. Dr. Rogerio Atem. A oportunidade de participar dos projetos de bolsa sob sua mentoria no polo de inovação nos proporcionou nossas primeiras experiências práticas, sendo um passo fundamental para a aplicação dos conhecimentos teóricos e para o despertar de nossa paixão pela tecnologia. Sua confiança foi essencial em nosso início de carreira.

Agradecemos a todos os professores do Instituto Federal Fluminense que, ao longo do curso, compartilharam seus conhecimentos e nos inspiraram, bem como aos nossos colegas e amigos, que tornaram a jornada mais leve com momentos de companheirismo e apoio mútuo.

Por fim, esta conquista reforça nossa crença de que a educação é a chave não apenas para atingir objetivos pessoais, mas para transformar a sociedade em um lugar mais próspero, solidário e justo.

Resumo

Com a crescente complexidade das aplicações web e a busca por experiências de usuário ricas e performáticas, a escolha da estratégia de renderização de conteúdo tornou-se uma decisão arquitetural fundamental. Nesse cenário, o *Client-Side Rendering* (CSR) e o *Server-Side Rendering* (SSR) emergem como as principais abordagens, cada uma com *trade-offs* significativos: o CSR favorece a interatividade contínua, enquanto o SSR otimiza o carregamento inicial e a indexação por motores de busca (SEO). A literatura, no entanto, carece de estudos de caso práticos que comparem seus impactos de forma direta e controlada.

Com a intenção de abordar essa lacuna, este trabalho apresenta uma análise comparativa detalhada, fundamentada em um estudo de caso realista: o desenvolvimento de uma plataforma de notícias em duas versões funcionalmente idênticas, uma com React (CSR) e outra com Next.js (SSR). Por meio da coleta de métricas de *Core Web Vitals* em um ambiente controlado com contêineres Docker, a pesquisa avalia empiricamente os efeitos de cada arquitetura no desempenho e na experiência do usuário.

Os resultados empíricos revelaram um claro trade-off: a arquitetura SSR obteve superioridade no carregamento inicial (TTFB e FCP) e na otimização para SEO, enquanto a abordagem CSR se destacou pela interatividade fluida (INP) e pelo custo reduzido de servidor, embora tenha apresentado maior instabilidade visual (CLS). Conclui-se que a escolha é estratégica e depende dos objetivos do projeto: SSR é recomendado para aplicações focadas em conteúdo e descoberta, como portais e e-commerce, enquanto CSR se adequa a sistemas ricos em interatividade, como dashboards e plataformas logadas. Como limitação, o estudo foi conduzido em ambiente local, não abrangendo a variabilidade de redes e dispositivos do mundo real.

Palavras-chave: Renderização do lado do cliente, CSR, Renderização do lado do servidor, SSR, Renderização Web, Desempenho, SEO, UX.

Abstract

With the growing complexity of web applications and the demand for rich, high-performance user experiences, the choice of a content rendering strategy has become a fundamental architectural decision. In this context, *Client-Side Rendering* (CSR) and *Server-Side Rendering* (SSR) emerge as the main approaches, each with significant *trade-offs*: CSR favors continuous interactivity, while SSR optimizes initial load times and search engine optimization (SEO). The literature, however, lacks practical case studies that directly and controllably compare their impacts.

To address this gap, this work presents a detailed comparative analysis based on a realistic case study: the development of a news platform in two functionally identical versions, one with React (CSR) and the other with Next.js (SSR). Through the collection of *Core Web Vitals* metrics in a controlled environment using Docker containers, the research empirically evaluates the effects of each architecture on performance and user experience.

The empirical results revealed a clear trade-off: the SSR architecture demonstrated superiority in initial loading (TTFB and FCP) and SEO, while the CSR approach excelled in fluid interactivity (INP) and reduced server cost, although it exhibited greater visual instability (CLS). It is concluded that the choice is strategic and depends on the project's objectives: SSR is recommended for applications focused on content and discovery, such as portals and e-commerce, while CSR is suited for interaction-rich systems, such as dashboards and logged-in platforms. As a limitation, the study was conducted in a local environment, not encompassing the variability of real-world networks and devices.

Keywords: Client-side rendering, CSR, Server-side rendering, SSR, Web Rendering, Performance, SEO, UX.

Listas de Figuras

Figura 1 – Comunicação entre cliente e servidor.	20
Figura 2 – Analogia entre HTML, CSS e JavaScript e os componentes de um corpo humano.	25
Figura 3 – Comparativo entre estratégias de renderização	28
Figura 4 – Etapas do método de renderização no lado do cliente	30
Figura 5 – Etapas do método de renderização no lado do servidor	32
Figura 6 – Etapas do método de geração estática de páginas (SSG)	35
Figura 7 – Funcionamento do modelo Incremental Static Regeneration (ISR)	37
Figura 8 – Funcionamento da estratégia Deferred Static Generation (DSG)	38
Figura 9 – Tempo de Rastreamento e Posicionamento da Página	45
Figura 10 – Arquitetura simplificada do Docker (cliente, daemon, registries e objetos).	52
Figura 11 – Número de publicações ao longo dos anos	59
Figura 12 – Distribuição das publicações por país	59
Figura 13 – Wireframe da plataforma WallTech	70
Figura 14 – Diagrama de caso de uso da plataforma WallTech	71
Figura 15 – Exemplo de quadro Kanban no <i>GitHub Projects</i>	72
Figura 16 – Exemplo de cartão <i>issue</i> no <i>GitHub Projects</i>	73
Figura 17 – Diagrama de sequência - SSR/MPA	80
Figura 18 – Diagrama de Componentes - MPA (SSR em Next.js)	81
Figura 19 – Diagrama de sequência - CSR/SPA	82
Figura 20 – Diagrama de Componentes - SPA (React)	82
Figura 21 – Aplicação SPA com React (CSR)	84
Figura 22 – Aplicação MPA com Next.js (SSR)	85
Figura 23 – Média de uso de CPU por tempo (SSR)	98
Figura 24 – Média de uso de memória por tempo (SSR)	99
Figura 25 – Classificação das métricas de desempenho (SSR) com Web Vitals	100
Figura 26 – Média de uso de CPU por tempo (CSR)	101
Figura 27 – Média de uso de memória por tempo (CSR)	102
Figura 28 – Classificação das métricas de desempenho (CSR) com Web Vitals	103

Lista de quadros

Quadro 1 – Estrutura PICOC aplicada à pesquisa	57
Quadro 2 – Expressão de busca utilizada	58
Quadro 3 – Artigos selecionados sobre estratégias de renderização e desempenho em aplicações web	60

Lista de códigos

2.1	Exemplo de HTML mínimo em aplicação Angular com CSR	31
2.2	Exemplo de HTML mínimo em aplicação Next.js com SSR	34
2.3	Exemplo de HTML estático gerado com SSG	35
2.4	Exemplo de revalidação de conteúdo com ISR no Next.js	37
2.5	Exemplo de definição de página DSG no Gatsby	39
2.6	Exemplo de multi-stage build para aplicação SSR com Node/Next.js	53
2.7	Servidor estático com Nginx para aplicação CSR/SSG	54

Siglas

CLS	Cumulative Layout Shift
CSR	Client-Side Rendering
CSS	Cascading Style Sheets
DOM	Document Object Model
DSG	Deferred Static Generation
FCP	First Contentful Paint
FID	First Input Delay
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
INP	Interaction to Next Paint
ISR	Incremental Static Regeneration
LCP	Largest Contentful Paint
MPA	Multi Page Application
QUIC	Quick UDP Internet Connections
RUM	Real User Monitoring
SEO	Search Engine Optimization
SPA	Single Page Application
SSG	Static Site Generation
SSL	Secure Sockets Layer
SSR	Server-Side Rendering
TBT	Total Blocking Time
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTFB	Time to First Byte

TTI Time to Interactive

UDP User Datagram Protocol

UX User Experience

Sumário

1	INTRODUÇÃO	15
1.1	Problema e contexto	15
1.2	Justificativa	16
1.3	Objetivos	17
1.3.1	Objetivos Específicos	17
2	FUNDAMENTAÇÃO TEÓRICA	19
2.1	Fundamentos de Desenvolvimento Web	19
2.1.1	Arquitetura Cliente-Servidor	19
2.1.2	Protocolo HTTP	20
2.1.3	HTML, CSS e JavaScript	23
2.2	Renderização na Web	25
2.2.1	Single Page Application (SPA) e Multi Page Application (MPA)	25
2.2.2	Estratégias e Terminologias de Renderização	26
2.2.2.1	Desempenho e Compensações	27
2.2.2.2	Impacto em SEO e Indexação	27
2.2.3	<i>Client-Side Rendering</i> (CSR)	29
2.2.4	<i>Server-Side Rendering</i> (SSR)	31
2.2.5	<i>Static Site Generation</i> (SSG)	34
2.2.6	<i>Incremental Static Regeneration</i> (ISR)	36
2.2.7	<i>Deferred Static Generation</i> (DSG)	37
2.3	Métricas e Ferramentas de Análise de Desempenho	39
2.3.1	Ferramentas de Análise de Mercado	39
2.3.2	Google Lighthouse	40
2.3.3	Core Web Vitals	40
2.3.3.1	Principais Métricas do Web Vitals	40
2.3.3.2	Métricas Complementares	41
2.4	Frameworks Web	41
2.4.1	Bibliotecas JavaScript	41
2.4.2	Frameworks para CSR	42
2.4.3	Meta-frameworks para SSR	42
2.4.4	Comparativo entre Frameworks CSR e SSR	43

2.5	Experiência do Usuário	44
2.5.1	<i>Search Engine Optimization (SEO)</i>	44
2.5.2	<i>Velocidade de carregamento</i>	45
2.5.3	Interatividade	46
2.5.4	Acessibilidade	46
2.6	Ferramentas Modernas para Prototipação e Interfaces	47
2.6.1	v0	47
2.6.2	shadcn/ui	48
2.6.3	Integração entre v0 e shadcn/ui	48
2.7	Processo de Desenvolvimento e Controle de Gestão de Software	48
2.7.1	Git	48
2.7.2	GitHub	49
2.7.3	GitHub Projects	49
2.8	News API	50
2.8.1	Escopo e cobertura	50
2.8.2	Autenticação e formato de resposta	50
2.8.3	Funcionalidade e integração	50
2.9	Infraestrutura de Contêineres e Docker	51
2.9.1	Fundamentos da Plataforma Docker	51
2.9.2	Construção de Imagens com Dockerfile	52
2.9.3	Boas Práticas e Otimização de Builds	52
2.9.4	Padrões de Infraestrutura para Aplicações CSR e SSR	53
3	TRABALHOS RELACIONADOS	55
3.1	Questões de pesquisa	55
3.2	Estratégia de busca	55
3.3	Quadro PICOC	56
3.3.1	Expressão de busca	57
3.4	Estratégia de seleção	58
3.5	Caracterização de pesquisa	58
3.6	Artigos selecionados	60
3.7	Resultados e Discussão	61
4	METODOLOGIA	63
4.1	Levantamento Teórico	63
4.2	Mapeamento Sistemático da Literatura	64
4.3	Estudo de Caso Prático	65
4.4	Coleta de Dados e Testes	65
4.4.1	Definição das Métricas	65
4.4.2	Ferramentas de Teste	66

4.4.3	Ambiente de Testes	67
4.4.4	Execução dos Testes	68
4.4.5	Tratamento e Análise dos Dados	68
5	ESTUDO DE CASO	69
5.1	Contexto	69
5.2	Processo de Desenvolvimento	71
5.2.1	Planejamento Funcional: Épico e Histórias de Usuário	73
5.3	Requisitos	77
5.4	Design do Sistema	78
5.5	Abordagens de Renderização e Navegação	79
5.5.1	SSR/MPA - Fluxo e Arquitetura	79
5.5.2	CSR/SPA - Fluxo e Arquitetura	81
5.6	Implementação	83
5.6.1	Implementação da Aplicação SPA	83
5.6.2	Implementação da Aplicação MPA	84
6	COLETA DE DADOS	86
6.1	Configuração do Ambiente Experimental	86
6.1.1	Limitações de Execução e Decisão Metodológica	86
6.1.2	Instrumentação de Web Vitals	86
6.1.3	Empacotamento Docker: SSR/MPA (Next.js) e CSR/SPA (React+Vite)	89
6.1.4	Paridade de Recursos, Observabilidade e Procedimento de Coleta	91
6.1.5	Repositórios e Reprodutibilidade	92
6.1.6	Coleta de CPU/RAM do contêiner e exportação em CSV	92
6.1.7	Uso complementar do Lighthouse (Chrome DevTools)	93
7	RESULTADOS E DISCUSSÕES	96
7.1	Resultados	96
7.1.1	Análise e Visualização dos Dados	96
7.1.2	Tratamento dos Dados	96
7.1.3	Metas de referência	97
7.1.4	Resultados Aplicação SSR	98
7.1.5	Resultados Aplicação CSR	101
7.1.6	Comparação entre SSR e CSR	104
7.1.7	Discussão	106
7.1.8	Implicações práticas	106
7.1.9	Ajustes recomendados	107
7.1.10	Síntese	107
8	CONCLUSÃO	108

1 Introdução

Este capítulo apresenta o tema do trabalho, que visa analisar a complexidade crescente das aplicações web e a importância da escolha de uma estratégia de renderização. O estudo se aprofunda na avaliação das abordagens de [Client-Side Rendering \(CSR\)](#) e [Server-Side Rendering \(SSR\)](#), comparando sua eficácia para otimização de desempenho e experiência do usuário.

1.1 Problema e contexto

O crescimento acelerado da web e o aumento da complexidade das aplicações modernas impuseram novos desafios ao desenvolvimento e à entrega de conteúdos na internet. Com o crescimento exponencial da web, estima-se que aproximadamente 252 mil novos sites sejam desenvolvidos diariamente, demonstrando não apenas a rapidez com que aplicações são criadas, mas também a necessidade crescente de estratégias eficientes para otimização de desempenho e escalabilidade ([SITEEFY, 2021](#)). A escolha da abordagem de renderização tornou-se um fator determinante para a experiência do usuário e a escalabilidade dos sistemas. Inicialmente, os sites eram compostos por páginas estáticas, cujo conteúdo era carregado diretamente do servidor. Com a evolução das tecnologias frontend, novas abordagens surgiram, destacando-se [Client-Side Rendering \(CSR\)](#) e [Server-Side Rendering \(SSR\)](#). Cada uma dessas técnicas possui características específicas que influenciam diretamente o desempenho e a experiência do usuário.

A performance em websites é um fator determinante para o sucesso de qualquer aplicação web. O desempenho, frequentemente medido pelo tempo de carregamento das páginas, desempenha um papel fundamental na experiência do usuário e na taxa de conversão de visitantes ([WAGNER, 2016](#)). Uma página que carrega rapidamente proporciona uma navegação mais fluida, reduzindo a taxa de rejeição e aumentando a retenção de usuários. Além disso, o desempenho da página não se limita a impactar a experiência do usuário, mas também interfere diretamente no [Search Engine Optimization \(SEO\)](#), tornando-se um critério essencial de indexação e ranqueamento em plataformas como o Google ([GOOGLE, 2010](#)).

Um exemplo notável de desafios enfrentados na escolha da estratégia de renderização ocorreu no *Twitter*. Em 2010, a empresa lançou uma nova versão de sua plataforma, conhecida como New Twitter, que utilizava extensivamente a renderização no lado do cliente ([CSR](#)) para aprimorar a interatividade e a experiência do usuário. No entanto, essa abordagem resultou em problemas significativos de desempenho, especialmente para

usuários com conexões de internet mais lentas ou dispositivos menos potentes. Além disso, a dependência intensa de JavaScript dificultou a indexação de conteúdo pelos mecanismos de busca, impactando negativamente a otimização para motores de busca ([SEO](#)). Reconhecendo essas limitações, o Twitter decidiu retornar à renderização no lado do servidor ([SSR](#)) em 2012, visando melhorar o desempenho e a acessibilidade de sua plataforma ([TWITTER, 2015](#)).

A arquitetura de frontend desempenha papel fundamental ao definir o fluxo de desenvolvimento e a escolha entre [CSR](#) e [SSR](#), sendo indispensável a adoção de um sistema modular e eficiente, capaz de ser mantido e escalado de forma sustentável ([GODBOLT, 2016](#)). Na abordagem [CSR](#), a renderização ocorre diretamente no navegador do usuário, reduzindo a carga no servidor, mas exigindo mais processamento no cliente; já na [SSR](#), o conteúdo é gerado no servidor antes de ser enviado ao cliente, o que proporciona carregamento mais rápido e melhor desempenho em dispositivos menos potentes. A decisão entre essas estratégias está diretamente ligada à performance da aplicação e deve considerar fatores como tempo de carregamento, complexidade da página e número de requisições HTTP , já que diferentes abordagens afetam não apenas a experiência do usuário, mas também os custos operacionais e a infraestrutura necessária para suportar a aplicação ([WAGNER, 2016](#)).

1.2 Justificativa

Nos últimos anos, observou-se um crescimento expressivo na adoção de abordagens de renderização tanto no lado do cliente ([CSR](#)) quanto no lado do servidor ([SSR](#)) em aplicações web, sobretudo quando comparadas a modelos tradicionais que utilizam apenas páginas estáticas ou *templates* processados integralmente no servidor. Esse avanço deve-se, em grande parte, à busca contínua por melhor desempenho, experiências de usuário mais rápidas e dinâmicas, além da popularização de *frameworks* e bibliotecas que simplificam a implementação dessas abordagens ([EMADAMERHO-ATORI, 2024](#)).

Contudo, essas técnicas são frequentemente empregadas de maneira inadequada em muitos projetos, seja pela falta de entendimento de suas vantagens e limitações, seja por uma análise superficial das necessidades do produto. Um exemplo ilustrativo dessa realidade pode ser visto na experiência do *Airbnb*, que optou por uma abordagem de [SSR](#) com o intuito de melhorar o desempenho em dispositivos com recursos limitados e, sobretudo, otimizar a indexação de seu vasto catálogo de acomodações em mecanismos de busca ([NEARY, 2017](#)). Por outro lado, a equipe do *Instagram* enfrentou desafios ao equilibrar o carregamento dinâmico de conteúdo no cliente com a necessidade de garantir uma experiência fluida aos usuários, levando-os a adotar soluções híbridas que envolvem tanto [CSR](#) quanto [SSR](#) em diferentes partes da aplicação ([CONNER, 2019](#)).

Paralelamente a esses casos, identifica-se uma carência de estudos de caso reais que analisem de forma aprofundada o impacto da adoção de **CSR** e **SSR**, principalmente no contexto nacional. Enquanto algumas publicações se concentram em apenas uma dessas abordagens, outras fornecem exemplos excessivamente simplificados, limitando a compreensão dos desafios técnicos e de negócios ao combinar essas estratégias em sistemas complexos.

Diante desse cenário, o presente trabalho se propõe a preencher essa lacuna por meio de uma análise detalhada e prática. Para isso, foi desenvolvido um estudo de caso focado em uma aplicação de notícias, implementada em duas versões funcionalmente idênticas: uma com **CSR** e outra com **SSR**. Ao comparar diretamente os efeitos de cada abordagem no desempenho, na experiência do usuário, na otimização para **SEO** e na escalabilidade, espera-se fornecer subsídios concretos que possam orientar equipes de desenvolvimento e gestores na seleção da arquitetura de renderização mais adequada, auxiliando na construção de sistemas mais robustos e eficientes.

1.3 Objetivos

O objetivo geral deste trabalho de conclusão de curso é realizar uma análise comparativa detalhada entre **Client-Side Rendering (CSR)**, aplicado em uma arquitetura **Single Page Application (SPA)**, e **Server-Side Rendering (SSR)**, associado a uma **Multi Page Application (MPA)**, materializada através do desenvolvimento e teste de um estudo de caso prático: uma aplicação de notícias implementada em ambas as arquiteturas. A análise busca avaliar, com base em dados empíricos, os efeitos de cada abordagem no desempenho, na experiência do usuário, na otimização do **Search Engine Optimization (SEO)** e na escalabilidade de aplicações web modernas.

1.3.1 Objetivos Específicos

- Realizar um levantamento do estado da arte sobre as arquiteturas **CSR** e **SSR**, identificando os conceitos fundamentais, métricas de desempenho e os *trade-offs* já consolidados na literatura.
- Projetar e implementar um estudo de caso prático, consistindo em duas aplicações de notícias funcionalmente idênticas, que isolem as arquiteturas **CSR** (**SPA** com React) e **SSR** (**MPA** com Next.js).
- Estruturar um ambiente de testes controlado e reproduzível, utilizando contêineres Docker, para executar as aplicações e coletar dados empíricos de desempenho, focados nas métricas de *Core Web Vitals* e no consumo de recursos do servidor (CPU e memória).

- Analisar comparativamente os dados coletados para validar o impacto de cada abordagem na experiência do usuário, identificar suas limitações práticas, e verificar seus efeitos na otimização para mecanismos de busca ([SEO](#)).
- Apresentar recomendações práticas, baseadas na análise empírica, para auxiliar equipes de desenvolvimento a decidir qual estratégia de renderização se alinha melhor aos requisitos de um projeto web.

2 Fundamentação Teórica

Este capítulo apresenta os conceitos de *Client-Side Rendering (CSR)* e *Server-Side Rendering (SSR)*, abordando os princípios fundamentais do desenvolvimento web relacionados à renderização de conteúdo. Também são discutidos aspectos como SEO, desempenho, infraestrutura de serviços e impacto na experiência do usuário, estabelecendo a base teórica para o estudo de caso desenvolvido neste trabalho.

2.1 Fundamentos de Desenvolvimento Web

Para entender como as abordagens SSR e CSR se inserem no cenário de desenvolvimento web, é fundamental revisar protocolos, modelos de arquitetura e ferramentas. Os fundamentos de desenvolvimento web englobam os princípios, tecnologias e práticas essenciais para a criação e manutenção de aplicações acessíveis via internet.

O desenvolvimento web baseia-se na arquitetura cliente-servidor, onde o cliente (geralmente um navegador) solicita recursos ao servidor, que processa essas requisições e retorna os dados necessários. Essa interação é mediada por protocolos como o *Hypertext Transfer Protocol (HTTP)*, que define as regras de comunicação entre cliente e servidor.

As tecnologias fundamentais incluem *HyperText Markup Language (HTML)* para estruturação do conteúdo, *Cascading Style Sheets (CSS)* para estilização e JavaScript para interatividade. Essas linguagens permitem a construção de interfaces dinâmicas e responsivas. Além disso, o desenvolvimento web envolve práticas como controle de versão, testes automatizados e integração contínua, que garantem a qualidade e a escalabilidade das aplicações (VIANA, 2024).

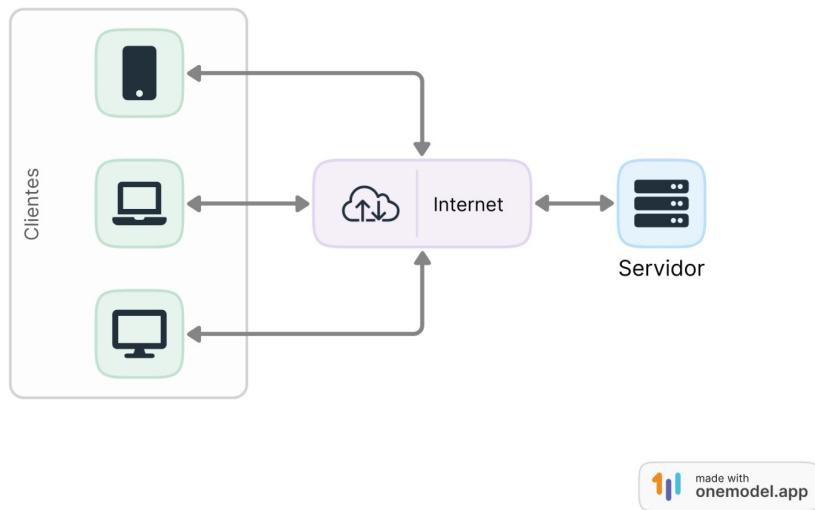
2.1.1 Arquitetura Cliente-Servidor

A arquitetura cliente-servidor é um modelo amplamente adotado no desenvolvimento de aplicações web, caracterizado pela separação entre dois componentes principais: cliente, responsável pela interface com o usuário, e o servidor, que processa solicitações e fornece os recursos necessários (ControleNet, 2025).

Nesse modelo, os clientes como navegadores em diferentes dispositivos enviam requisições através da internet, enquanto os servidores respondem disponibilizando dados, arquivos e serviços. Essa divisão de responsabilidades favorece a escalabilidade, facilita a manutenção e permite que cliente e servidor operem em plataformas distintas (VIANA, 2024).

A Figura 1 ilustra, de forma simplificada, esse fluxo de comunicação entre cliente e servidor.

Figura 1 – Comunicação entre cliente e servidor.



Fonte: (VIANA, 2024)

A arquitetura cliente-servidor apresenta características que contribuem para sua ampla adoção em aplicações web. Entre elas, destaca-se a distribuição de responsabilidades, onde o servidor gerencia dados e processos mais complexos, enquanto o cliente lida com a interface e a interação com o usuário.

Outro ponto relevante é a independência entre plataformas, possibilitada pelo uso de protocolos padronizados, o que permite a comunicação entre diferentes dispositivos e sistemas operacionais. Além disso, esse modelo favorece a facilidade de manutenção, já que atualizações podem ser feitas no servidor sem necessidade de intervenção nos dispositivos dos usuários.

Na web, essa arquitetura é implementada por padrão: navegadores atuam como clientes, enviando requisições **HTTP** que são processadas por servidores, os quais respondem com páginas e recursos solicitados (VIANA, 2024).

2.1.2 Protocolo **HTTP**

O Protocolo de Transferência de Hipertexto (**HTTP**) é a base da comunicação na World Wide Web, definindo como clientes (navegadores) e servidores trocam informações.

Ele especifica a estrutura das requisições e respostas, permitindo a recuperação de recursos como documentos html), imagens e vídeos ([MDN Web Docs, 2024](#)).

O Funcionamento do **HTTP** opera no modelo cliente-servidor, onde o cliente inicia uma requisição e o servidor responde com os recursos solicitados ou mensagens de erro, se aplicável. Cada interação consiste em uma mensagem de requisição do cliente e uma mensagem de resposta do servidor. As mensagens **HTTP** são compostas por:

- Linha de início: Indica o método **HTTP** (como **GET** ou **POST**) e o caminho do recurso.
- Cabeçalhos: Fornecem informações adicionais sobre a requisição ou resposta, como tipo de conteúdo e codificação.
- Corpo: Contém os dados enviados ou recebidos, sendo opcional dependendo do método utilizado.

([MDN Web Docs, 2024](#))

Métodos **HTTP** são operações definidas pelo protocolo que especificam a ação a ser realizada em um recurso. Os métodos mais comuns incluem:

- **GET**: Solicita a representação de um recurso específico. Requisições **GET** devem ser utilizadas apenas para recuperar dados.
- **POST**: Envia dados ao servidor para processamento, como o envio de formulários.
- **PUT**: Atualiza um recurso existente ou cria um novo se não existir.
- **DELETE**: Remove um recurso específico.
- **HEAD**: Similar ao **GET**, mas solicita apenas os cabeçalhos da resposta, sem o corpo.

Cada método possui uma finalidade específica e deve ser utilizado conforme a necessidade da aplicação ([Wikipedia, 2024](#)).

Códigos de Status **HTTP** são códigos de três dígitos que indicam o resultado de uma requisição feita pelo cliente ao servidor. Eles são agrupados em cinco classes principais:

- **1xx** (Informativo): Indica que a requisição foi recebida e o processo continua.
- **2xx** (Sucesso): Indica que a requisição foi bem-sucedida. Exemplo: **200 OK**.

- 3xx (Redirecionamento): Indica que é necessário tomar medidas adicionais para completar a requisição. Exemplo: 301 Moved Permanently.
- 4xx (Erro do Cliente): Indica que houve um erro na requisição do cliente. Exemplo: 404 Not Found.
- 5xx (Erro do Servidor): Indica que o servidor falhou ao processar uma requisição válida. Exemplo: 500 Internal Server Error.

Esses códigos auxiliam na identificação e resolução de problemas durante a comunicação [HTTP \(MDN Web Docs, 2024\)](#).

Evolução do [HTTP](#) refere-se às revisões progressivas do protocolo com o objetivo de aprimorar sua eficiência e desempenho ao longo do tempo. As principais versões são:

- HTTP/1.0: Primeira versão oficial do protocolo, em que cada requisição exigia uma nova conexão com o servidor.
- HTTP/1.1: Introduziu conexões persistentes, permitindo múltiplas requisições por conexão. Trouxe também melhorias no controle de cache e suporte a novos métodos.
- HTTP/2: Implementou multiplexação, compressão de cabeçalhos e priorização de fluxos, resultando em uma transferência de dados mais rápida e eficiente.
- HTTP/3: Baseado no protocolo [QUIC](#), substitui o [TCP](#) pelo [UDP](#), oferecendo conexões mais rápidas e seguras, com menor latência e melhor desempenho em redes instáveis.

Essas atualizações refletem a evolução das necessidades da web e a busca por protocolos mais robustos e otimizados ([Cloudflare, 2025](#)).

[HTTP](#) e [HTTPS](#) representam protocolos utilizados para comunicação na web, com a principal distinção centrada na segurança da transmissão dos dados.

O [*Hypertext Transfer Protocol Secure \(HTTPS\)*](#) é uma extensão do [HTTP](#) que adiciona uma camada de proteção por meio do protocolo [*Transport Layer Security \(TLS\)*](#) ou, anteriormente, [*Secure Sockets Layer \(SSL\)*](#). Essa camada de segurança garante a confidencialidade, integridade e autenticidade dos dados trafegados entre cliente e servidor.

A criptografia utilizada impede que terceiros acessem ou modifiquem as informações transmitidas, o que é fundamental em transações sensíveis, como cadastros, pagamentos e autenticações. Além disso, o uso de *certificados digitais* garante que o site visitado é

realmente aquele que afirma ser, protegendo os usuários contra ataques como o *man-in-the-middle*¹.

Enquanto o **HTTP** tradicional opera normalmente na porta **TCP 80**, o **HTTPS** utiliza, por convenção, a porta 443. Atualmente, o uso do **HTTPS** é fortemente recomendado e até exigido por navegadores modernos como padrão de segurança para qualquer aplicação web, contribuindo para a privacidade e confiança dos usuários ([Wikipedia, 2024](#)).

2.1.3 HTML, CSS e JavaScript

O desenvolvimento frontend, conforme definido por [Amazon Web Services \(2024\)](#), refere-se à camada de apresentação de uma aplicação web a interface gráfica com a qual os usuários interagem diretamente, composta por menus, botões, formulários e outros elementos visuais. Essa camada baseia-se em um conjunto de tecnologias fundamentais que operam em conjunto para fornecer estrutura, estilo e interatividade às páginas: **HTML**, **CSS** e **JavaScript**. Cada uma dessas linguagens desempenha um papel específico e complementar, sendo essenciais tanto em abordagens tradicionais quanto em técnicas modernas como o **CSR**.

HyperText Markup Language (HTML) é a linguagem de marcação padrão para a criação da estrutura de páginas web. Através de um conjunto de elementos (ou *tags*), o **HTML** organiza e define o conteúdo exibido ao usuário, como textos, imagens, links, formulários e tabelas. Além de estruturar visualmente o documento, o **HTML** também confere semântica aos elementos, facilitando a indexação por motores de busca e promovendo acessibilidade para leitores de tela. Elementos como `<header>`, `<main>`, `<article>` e `<footer>` exemplificam essa função semântica ([BALLERINI, 2023](#)).

Cascading Style Sheets (CSS) é a linguagem responsável pela estilização das páginas web. Com o **CSS**, define-se a aparência dos elementos estruturados no **HTML**, controlando propriedades visuais como cores, fontes, espaçamentos, tamanhos e posicionamentos. O **CSS** permite ainda a construção de layouts complexos e responsivos, adaptando o conteúdo para diferentes tamanhos de tela e dispositivos. A separação entre estrutura (**HTML**) e estilo (**CSS**) é um dos pilares das boas práticas em desenvolvimento web, promovendo manutenibilidade, reutilização e modularidade do código.

Entre os recursos modernos do **CSS**, destacam-se os seletores avançados, variáveis **CSS**, pseudo-classes, animações e as funcionalidades de *Flexbox* e *Grid*, que facilitam a criação de interfaces ricas e adaptáveis ([KATTAH, 2023](#)).

¹ Um ataque *man-in-the-middle* ocorre quando um invasor intercepta e possivelmente altera a comunicação entre duas partes que acreditam estar se comunicando diretamente. Isso pode permitir que o invasor capture informações sensíveis ou injete dados maliciosos na comunicação. ([Wikipedia, 2025](#))

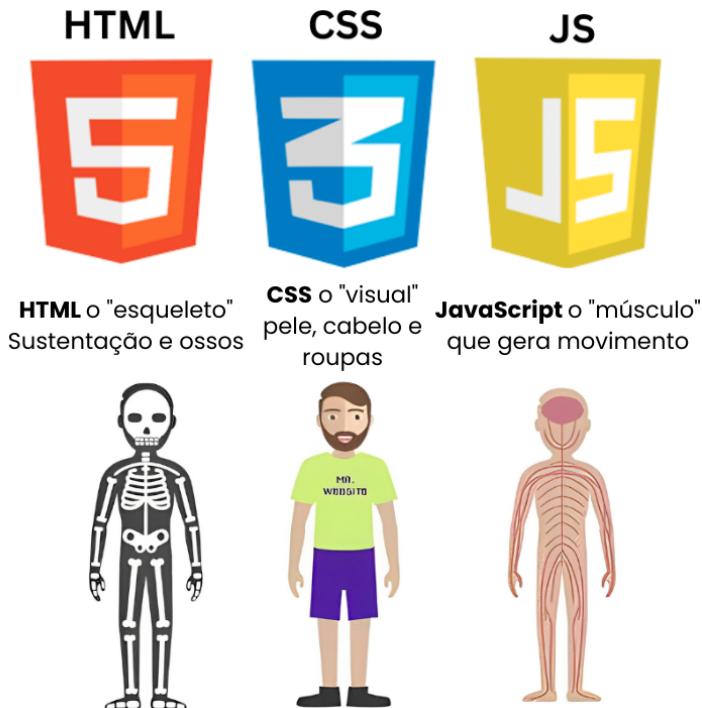
JavaScript é uma linguagem de programação interpretada, orientada a objetos e baseada em eventos, amplamente utilizada para adicionar interatividade e dinamismo às páginas web. Por meio da manipulação da *Document Object Model (DOM)*, permite implementar funcionalidades como respostas a cliques, envio de formulários, movimentações do mouse, digitação, animações, validações e atualizações em tempo real, enriquecendo significativamente a experiência do usuário (BALLERINI, 2023). Além disso, possibilita o carregamento assíncrono de dados com a técnica *AJAX (Asynchronous JavaScript and XML)*, evitando recarregamentos completos da página.

JavaScript é uma das três principais tecnologias da World Wide Web, juntamente com **HTML** e **CSS**, sendo essencial tanto em abordagens tradicionais quanto modernas. Nas aplicações baseadas em **CSR**, essa linguagem tem papel central, pois a renderização das páginas ocorre diretamente no navegador do usuário. Com a evolução do ecossistema JavaScript, surgiram bibliotecas e frameworks robustos como a biblioteca React e os frameworks Vue.js e Angular que facilitam o desenvolvimento de aplicações complexas com componentes reutilizáveis e gerenciamento eficiente de estado.

Adicionalmente, o JavaScript também pode ser executado no lado do servidor (**SSR**) por meio de ambientes como o Node.js uma plataforma de código aberto e multiplataforma baseada em eventos e não bloqueante, ideal para aplicações escaláveis e em tempo real (NODE, 2025; JAVASCRIPT, 2025). Isso permite o desenvolvimento de aplicações completas utilizando uma única linguagem em ambas as camadas, cliente e servidor.

A interação entre essas três tecnologias pode ser compreendida por meio de uma analogia: o **HTML** representa a estrutura de um corpo (esqueleto), o **CSS** corresponde à sua aparência externa (pele, roupas, estilo), enquanto o JavaScript age como os músculos e o sistema nervoso, controlando os movimentos e respostas interativas da aplicação. A Figura 2 ilustra essa relação.

Figura 2 – Analogia entre HTML, CSS e JavaScript e os componentes de um corpo humano.



Fonte: ([KATTAH, 2023](#)).

Portanto, o domínio dessas três tecnologias é indispensável para qualquer desenvolvedor web. Elas formam o alicerce sobre o qual se constroem interfaces acessíveis, performáticas e envolventes, sendo empregadas tanto em aplicações renderizadas no servidor ([SSR](#)) quanto no cliente ([CSR](#)), com adaptações específicas conforme a abordagem escolhida.

2.2 Renderização na Web

A renderização na Web diz respeito ao processo de transformar dados em conteúdo visual interpretável pelo navegador. A escolha sobre onde e como essa renderização será realizada (seja no cliente, no servidor ou em tempo de build), isso impacta diretamente métricas como tempo de carregamento, interatividade e indexação por mecanismos de busca ([OSMANI; MILLER, 2025](#)).

2.2.1 Single Page Application (SPA) e Multi Page Application (MPA)

As arquiteturas [Single Page Application \(SPA\)](#) e [Multi Page Application \(MPA\)](#) representam duas abordagens distintas para a estrutura de navegação e carregamento de conteúdo em aplicações web.

As **SPAs** são aplicações em que a navegação entre páginas ocorre sem recarregamentos completos do navegador. Após o carregamento inicial, todo o conteúdo adicional é gerenciado dinamicamente com JavaScript, o que proporciona uma experiência mais fluida e interativa. Essa abordagem é comumente utilizada em conjunto com a renderização no lado do cliente (**CSR**) e frameworks como React, Angular ou Vue.js ([EMADAMERHO-ATORI, 2024](#)).

Já as **MPAs** seguem o modelo tradicional de navegação, em que cada clique em um link leva a uma nova requisição **HTTP** e recarregamento completo da página. Essa arquitetura é naturalmente mais compatível com a renderização no lado do servidor (**SSR**) e favorece aspectos como **SEO**, acessibilidade e previsibilidade de comportamento ([OSMANI; MILLER, 2025](#)).

A escolha entre **SPA** e **MPA** está diretamente ligada à estratégia de renderização adotada. As **SPAs** tendem a oferecer experiências mais ricas e responsivas, mas exigem cuidados extras com desempenho e indexação. Por outro lado, as **MPAs** são mais robustas em cenários com grande volume de tráfego e requisitos de otimização para mecanismos de busca.

2.2.2 Estratégias e Terminologias de Renderização

Segundo [Osmani e Miller \(2025\)](#), é importante distinguir os principais modelos de renderização:

Client-Side Rendering (CSR) O conteúdo da aplicação é gerado dinamicamente no navegador, utilizando JavaScript. A página HTML inicial contém apenas uma estrutura básica com os scripts necessários para montar a interface após o carregamento. É comum em aplicações do tipo SPA (Single Page Application).

Server-Side Rendering (SSR) O servidor monta todo o conteúdo da página em HTML antes de enviá-lo ao cliente. Isso permite uma exibição mais rápida do conteúdo, mesmo em conexões lentas, e melhora a indexação por mecanismos de busca.

Static Site Generation (SSG) As páginas são geradas de forma estática em tempo de build, com base em dados disponíveis no momento da compilação. O conteúdo é entregue diretamente por uma CDN, garantindo alto desempenho.

Incremental Static Regeneration (ISR) Introduzido pelo Next.js, o ISR permite que páginas geradas estaticamente possam ser atualizadas de forma incremental, após um período de tempo definido. Isso é feito em segundo plano, sem bloquear o carregamento da página atual. Ideal para sites com atualizações frequentes, mas não críticas em tempo real.

Deferred Static Generation (DSG) Proposto pelo Gatsby, o DSG difere do ISR por não gerar certas páginas no momento do build. Em vez disso, elas são geradas apenas na primeira requisição (on-demand). Após isso, são armazenadas em cache e servidas como estáticas nas requisições seguintes. É útil em projetos com milhares de páginas de baixo acesso, reduzindo significativamente o tempo de build.

Além dessas abordagens, destaca-se o conceito de reidratação, que consiste em ativar a interatividade de páginas SSR ou SSG no cliente. Esse processo utiliza JavaScript para associar os eventos dinâmicos à estrutura HTML previamente renderizada, sendo essencial para tornar a página interativa após a exibição inicial (OSMANI; MILLER, 2025).

2.2.2.1 Desempenho e Compensações

A renderização do lado do servidor tende a exibir conteúdo mais rapidamente (menor FCP), favorecendo acessibilidade e SEO. No entanto, pode aumentar o TTFB, já que a página precisa ser processada antes de ser enviada (OSMANI; MILLER, 2025). Já a renderização no cliente pode reduzir o tempo de resposta inicial do servidor, mas exige mais do navegador e aumenta o tempo até a página estar interativa Time to Interactive (TTI), especialmente em dispositivos móveis.

Modelos híbridos como SSR com *hydration* tentam unir os benefícios de ambas as abordagens, mas podem causar atrasos na interatividade. Técnicas como *hydration progressiva* ou *streaming* reduzem esses impactos ao ativar partes da interface conforme necessário (OSMANI; MILLER, 2025). Estratégias mais recentes, como a renderização trimórfica, permitem que a renderização ocorra em três camadas: servidor, cliente e *service worker*. Isso possibilita desempenho superior em acessos repetidos e melhor controle sobre o cache e atualização de conteúdo dinâmico (OSMANI; MILLER, 2025).

2.2.2.2 Impacto em SEO e Indexação

Segundo Osmani e Miller (2025), abordagens que entregam HTML completo como SSR e SSG são mais eficazes para indexação por mecanismos de busca. Já modelos que dependem fortemente de JavaScript (CSR) exigem testes adicionais e podem comprometer a visibilidade em sistemas como o Googlebot, especialmente quando há falhas na execução dos scripts.

Figura 3 – Comparativo entre estratégias de renderização

	Server					Browser
Overview:	An application where input is navigation requests and the output is HTML in response to them.	Built as a Single Page App, but all pages prerendered to static HTML as a build step, and the JS is removed .	Built as a Single Page App. The server prerenders pages, but the full app is also booted on the client.	A Single Page App, where the initial shell/skeleton is prerendered to static HTML at build time.	A Single Page App. All logic, rendering and booting is done on the client. HTML is essentially just script & style tags.	
Authoring:	Entirely server-side (request-response, HTML)	Built as if client-side (components, DOM*, fetch)	Built as client-side	Client-side	Client-side	→
Rendering:	Dynamic HTML	Static HTML	Dynamic HTML and JS/DOM	Partial static HTML, then JS/DOM	Entirely JS/DOM	
Server role:	Controls all aspects. (thin client)	Delivers static HTML	Renders pages (navigation requests)	Delivers static HTML	Delivers static HTML	
Pros:	👍 TTI = FCP 👍 Fully streaming	👍 Fast TTFB 👍 TTI = FCP 👍 Fully streaming	👍 Flexible	👍 Flexible 👍 Fast TTFB	👍 Flexible 👍 Fast TTFB	
Cons:	👎 Slow TTFB 👎 Inflexible	👎 Inflexible 👎 Leads to hydration	👎 Slow TTFB 👎 TTI >> FCP 👎 Usually buffered	👎 TTI > FCP 👎 Limited streaming	👎 TTI >> FCP 👎 No streaming	
Scales via:	Infra size / cost	build/deploy size	Infra size & JS size	JS size	JS size	
Examples:	Gmail Basic HTML view, Hacker News	Docusaurus, Netflix*	Next.js , Razzle , etc	Gatsby, Vuepress, etc	Most apps	

Fonte: Adaptado de ([OSMANI; MILLER, 2025](#))

A Figura 3 oferece uma visão consolidada e detalhada de cinco estratégias de renderização web, do modelo puramente do servidor ao puramente do cliente. A tabela ilustra os *trade-offs* de cada abordagem em relação a aspectos como o papel do servidor, o tipo de HTML gerado e os impactos em desempenho e escala.

- **SSR com (Re)hydration:** O servidor pré-renderiza a página, mas a aplicação completa também é carregada no cliente. Embora o HTML inicial seja dinâmico, o JavaScript é responsável por tornar a página interativa, o que pode resultar em um atraso considerável entre a exibição do conteúdo e a interatividade completa (TTI » FCP). Este modelo é mais flexível, mas pode levar a um TTFB mais lento e a uma experiência "embaçada" (*buffered*) para o usuário.
- **SSR (Server Rendering):** Neste modelo, a aplicação é inteiramente renderizada no servidor, que responde a cada requisição de navegação com um HTML dinâmico completo. A principal vantagem é que o tempo de interatividade (TTI) é igual ao tempo até o primeiro conteúdo (FCP), e o conteúdo é totalmente transmitido em *streaming*. Contudo, ele pode ter um TTFB mais lento e é considerado inflexível.

- "Static SSR"(SSG): As páginas são pré-renderizadas para HTML estático em tempo de *build* e o JavaScript é removido. Essa abordagem é construída como um SPA, mas entrega um HTML estático que é servido diretamente ao cliente. Os benefícios incluem um rápido TTFB e o TTI ser igual ao FCP, com a desvantagem de ser inflexível e ter que passar pela *hydration* para interatividade.
- CSR com Prerendering: A página é um SPA onde um "esqueleto" ou *shell* inicial é pré-renderizado para HTML estático durante o *build*. O servidor entrega apenas esse HTML parcial e o JavaScript/DOM são injetados posteriormente para renderizar o conteúdo. É uma abordagem flexível e com um TTFB rápido, mas o TTI ainda é maior que o FCP.
- Full CSR: Toda a lógica, renderização e inicialização da aplicação são feitas no cliente. O HTML enviado pelo servidor é basicamente um conjunto de *tags* de *script* e estilo. É uma abordagem flexível com um TTFB rápido, mas sofre com um grande atraso entre a renderização inicial e a interatividade (TTI »> FCP).

Em resumo, a figura demonstra que a escolha da renderização é um compromisso estratégico entre a velocidade de exibição inicial do conteúdo e a interatividade, com cada abordagem influenciando a carga computacional no servidor ou no navegador.

2.2.3 Client-Side Rendering (CSR)

Client-Side Rendering (CSR) é uma técnica em que a geração da interface e do conteúdo final ocorre diretamente no navegador do usuário, utilizando JavaScript. Nessa abordagem, o servidor envia um arquivo *HyperText Markup Language (HTML)* mínimo, contendo apenas a estrutura básica da página e referências a arquivos de estilo e scripts.(EMADAMERHO-ATORI, 2024)

Segundo Emadamerho-Atori (2024), o processo de renderização no cliente segue as seguintes etapas:

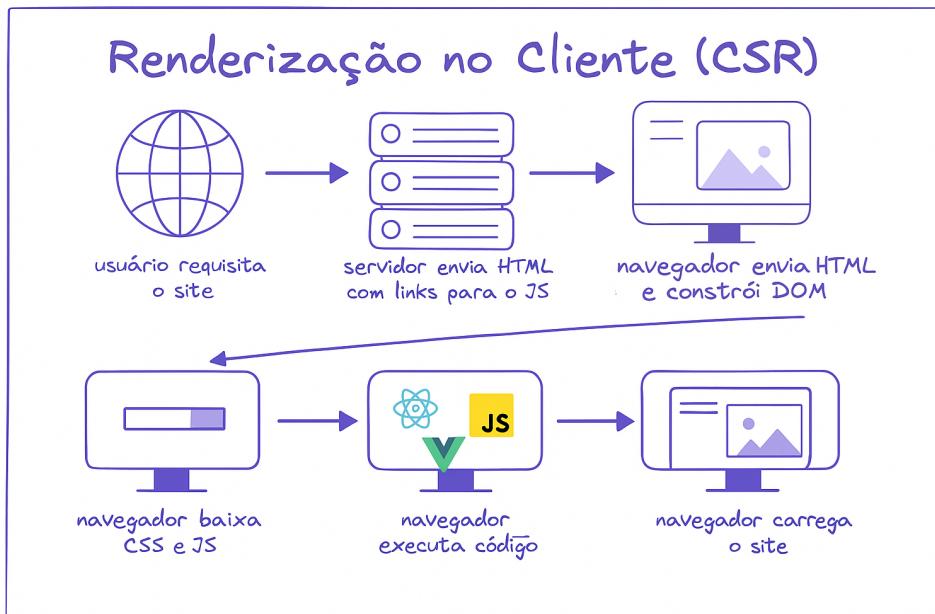
1. O servidor envia uma página *HTML* em branco contendo apenas links para os arquivos *Cascading Style Sheets (CSS)* e JavaScript.
2. O navegador interpreta o *HTML* e constrói a árvore do *Document Object Model (DOM)*
3. Os arquivos de estilo (*CSS*) e script (JavaScript) são baixados pelo navegador.
4. A aplicação é renderizada dinamicamente pelo JavaScript, incluindo elementos visuais como texto, imagens e botões.

- O conteúdo da página é atualizado de forma interativa conforme o usuário interage com a aplicação.

Esse modelo é comumente utilizado em aplicações *Single Page Application (SPA)*, nas quais o carregamento inicial é seguido por atualizações dinâmicas sem recarregamento da página. Ferramentas como a biblioteca React, e frameworks como Vue.js, Angular e Svelte são amplamente utilizadas para implementar **CSR**, permitindo o desenvolvimento de interfaces dinâmicas, interativas e responsivas.

A renderização no lado do cliente (**CSR**) é especialmente vantajosa em aplicações que exigem alta interatividade e atualizações frequentes de conteúdo, como redes sociais, plataformas de streaming e jogos online. No entanto, essa abordagem pode apresentar desvantagens em termos de desempenho inicial e **SEO**, uma vez que o conteúdo só é exibido após a execução do JavaScript, o que pode impactar negativamente a indexação por motores de busca e a experiência do usuário em conexões lentas (**EMADAMERHO-ATORI, 2024**).

Figura 4 – Etapas do método de renderização no lado do cliente



Fonte: (**EMADAMERHO-ATORI, 2024**) (adaptado)

A **Figura 4** ilustra visualmente o fluxo completo da renderização no lado do cliente (**CSR**). O processo é iniciado quando o usuário acessa o site em questão. Em resposta, o servidor envia o arquivo **HTML** básico, contendo apenas links para os arquivos de estilo **CSS** e scripts JavaScript responsáveis por carregar e renderizar o conteúdo da aplicação.

Na sequência, o navegador interpreta esse **HTML** e constrói a estrutura da página por meio da árvore **DOM**. No entanto, o conteúdo principal ainda não está visível.

O navegador então precisa baixar os arquivos de estilo (CSS) e os scripts JavaScript referenciados no documento inicial.

Com os scripts carregados, o navegador executa o código JavaScript, que normalmente utiliza bibliotecas ou frameworks como React ou Vue para gerar dinamicamente o conteúdo da aplicação. Somente após essa etapa o conteúdo completo do site é finalmente exibido ao usuário, quando o navegador conclui o processo de renderização e o site é carregado completamente.

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8">
5      <title>CryptoWebsite</title>
6      <base href="/">
7      <meta name="viewport" content="width=device-width, initial-scale=1">
8      <link rel="icon" type="image/x-icon" href="favicon.ico">
9      <style>*,*:before,*:after{margin:0;padding:0;box-sizing:border-box;
10         font-family:Inter,sans-serif}html{font-size:62.5%}</style>
11      <link rel="stylesheet" href="styles.9d4c7581c7242.css">
12  </head>
13  <body>
14      <app-root></app-root>
15      <script src="runtime.6170988ad52a05db.js" type="module"></script>
16      <script src="polyfills.574970d5ec4bdb97.js" type="module"></script>
17      <script src="main.202d37bb6740400e.js" type="module"></script>
18  </body>
19  </html>

```

Código 2.1 – Exemplo de HTML mínimo em aplicação Angular com CSR

Esse padrão é típico de aplicações SPA, onde todo o conteúdo é inserido dinamicamente a partir da execução dos arquivos JavaScript. O elemento `<app-root>` funciona como ponto de entrada da aplicação, sendo substituído no navegador pelos componentes definidos no framework Angular. (EMADAMERHO-ATORI, 2024)

2.2.4 Server-Side Rendering (SSR)

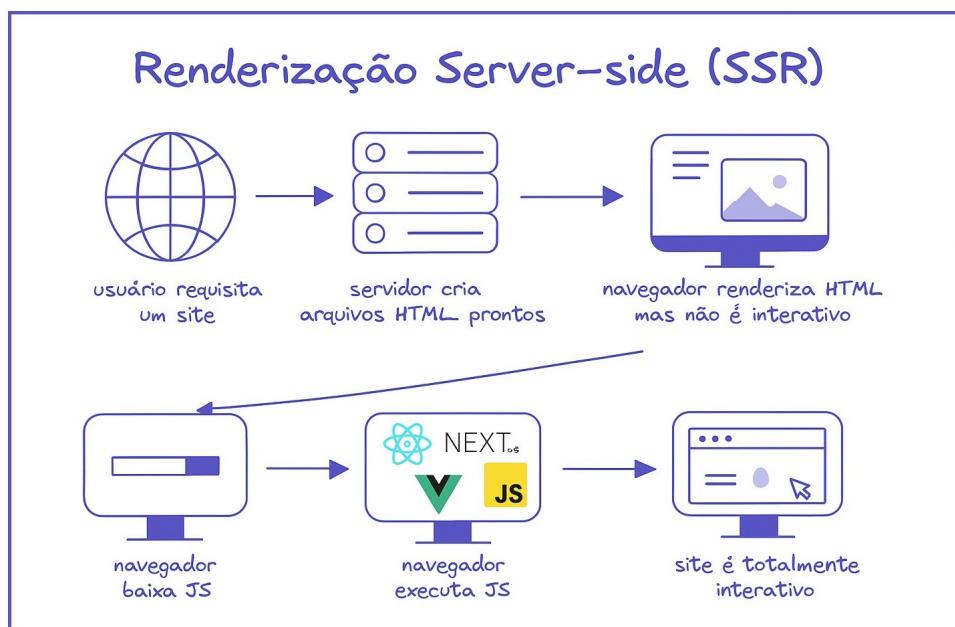
Server-Side Rendering (SSR) é uma abordagem em que a geração do conteúdo e da interface ocorre integralmente no servidor antes de ser enviada ao navegador do cliente. Ou seja, o servidor processa a lógica da aplicação, obtém dados necessários (por exemplo, em bancos de dados ou APIs) e retorna ao cliente um arquivo *HTML* já renderizado. Dessa forma, o navegador exibe imediatamente a página completa, sem precisar executar scripts para montar o conteúdo inicial (EMADAMERHO-ATORI, 2024).

Segundo Emadamerho-Atori (2024), o processo típico de renderização no lado do servidor pode ser descrito em quatro etapas principais:

1. O servidor recebe uma requisição para uma página e recupera os dados necessários para compor seu conteúdo (por exemplo, produtos de uma base de dados ou artigos de um blog).
2. O servidor insere esses dados em um *template HTML*, gerando a estrutura final da página.
3. Em seguida, o servidor aplica estilos e finaliza a renderização, resultando em um documento **HTML** completamente montado.
4. Por fim, esse documento **HTML** é enviado ao navegador do usuário, exibindo a página prontamente, sem a necessidade de executar *JavaScript* durante o carregamento inicial.

Nesse modelo, a fase de hydration² ocorre após o carregamento inicial da página. costuma ocorrer após a entrega do conteúdo estático. Significa que, assim que o arquivo **HTML** é carregado e mostrado ao usuário, o *JavaScript* do lado do cliente assume o controle para tratar as interações e atualizações dinâmicas subsequentes. Dessa forma, o **SSR** beneficia tanto o primeiro acesso (tornando o conteúdo visível rapidamente) quanto o **SEO**, por exibir ao rastreador dos mecanismos de busca um código **HTML** completo. ([EMADAMERHO-ATORI, 2024](#)).

Figura 5 – Etapas do método de renderização no lado do servidor



Fonte: ([EMADAMERHO-ATORI, 2024](#)) (adaptado)

² Hydration é uma etapa essencial no **SSR**, em que o *JavaScript* torna interativo o conteúdo **HTML** previamente renderizado no servidor.

A Figura 5 ilustra o fluxo de uma aplicação SSR. Ao receber a requisição, o servidor gera a página completa em HTML e a envia ao cliente. Essa estratégia costuma ser vantajosa em cenários onde o carregamento inicial rápido e a indexação por motores de busca são prioridades, como em sites de e-commerce e páginas de *landing*, permitindo que o usuário visualize o conteúdo de forma imediata.

Meta-frameworks como Next.js, Nuxt.js, SvelteKit, Angular Universal, Remix, Astro e Qwik são amplamente utilizados para construir aplicações com suporte a SSR. Esses frameworks operam em um nível superior aos tradicionais (como React, Vue ou Svelte), agregando funcionalidades comuns ao desenvolvimento web, como roteamento, pré-renderização, recuperação de dados e *hydration* podendo oferecer uma estrutura mais completa, opinativa e voltada à escalabilidade.

O SSR é especialmente útil em aplicações que exigem um carregamento inicial rápido e uma boa indexação por motores de busca, como sites de e-commerce, blogs e páginas de *landing*. Essa abordagem permite que o usuário visualize o conteúdo imediatamente, sem esperar pela execução do JavaScript. Além disso, o SSR melhora a SEO, pois os mecanismos de busca conseguem indexar o conteúdo completo da página desde o início.

No Código 2.2, pode-se observar que o arquivo HTML já contém todo o *markup* necessário para exibir o conteúdo da página. Assim que o navegador recebe esse arquivo, o usuário já visualiza o cabeçalho, o texto e o layout definidos. Posteriormente, o JavaScript baixado (por exemplo, `main.js`) pode entrar em ação para lidar com eventos, rotas adicionais e atualizações dinâmicas, caso o desenvolvedor deseje funcionalidades mais interativas.

Por fim, aplicações SSR tendem a apresentar melhor performance em termos de *time-to-first-byte*³ e de *indexabilidade*⁴ por motores de busca, ao mesmo tempo em que podem demandar maior carga de processamento no servidor. A escolha por SSR ou não, portanto, depende do perfil da aplicação e das prioridades do projeto, considerando fatores como volume de tráfego, necessidade de interatividade e requisitos de otimização de conteúdo.

³ O *time-to-first-byte* (TTFB) é uma métrica que mede o tempo decorrido entre o envio de uma solicitação HTTP pelo cliente e o recebimento do primeiro byte da resposta do servidor. Um TTFB menor indica maior rapidez na resposta do servidor, impactando diretamente na velocidade de carregamento da página e na experiência do usuário. (SMITH, 2025)

⁴ A *indexabilidade* refere-se à capacidade dos motores de busca de rastrear e indexar o conteúdo de uma página web. Aplicações SSR, ao fornecerem conteúdo totalmente renderizado no servidor, facilitam a indexação eficiente pelos motores de busca, melhorando a visibilidade nos resultados de pesquisa. (GÜNAÇAR, 2025)

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8">
5      <title>My SSR App</title>
6      <meta name="viewport" content="width=device-width, initial-scale=1">
7      <style>
8          /* Exemplo simples de estilo inline */
9          body {
10              margin: 0;
11              font-family: Arial, sans-serif;
12              background: #f6f6f6;
13          }
14          h1 { color: #333; }
15      </style>
16  </head>
17  <body>
18  <!-- Conteúdo já processado e inserido no servidor -->
19  <div id="__next">
20      <header>
21          <h1>Olá, mundo!</h1>
22      </header>
23      <main>
24          <p>Este conteúdo foi renderizado no servidor usando Next.js.</p>
25      </main>
26  </div>
27  <!-- Scripts do Next.js para interação no cliente -->
28  <script src="/_next/static/chunks/main.js" defer></script>
29 </body>
30 </html>

```

Código 2.2 – Exemplo de HTML mínimo em aplicação Next.js com SSR

2.2.5 Static Site Generation (SSG)

Static Site Generation (SSG) é uma técnica de pré-renderização na qual as páginas da aplicação são geradas estaticamente em tempo de *build* (compilação) e armazenadas como arquivos **HTML**. Ao contrário de abordagens como **CSR** e **SSR**, onde a renderização ocorre no navegador ou sob demanda no servidor, o **SSG** permite que o conteúdo já esteja pronto e otimizado para ser entregue diretamente ao navegador, reduzindo a carga do servidor e otimizando o desempenho de carregamento ([PERERA, 2022](#)).

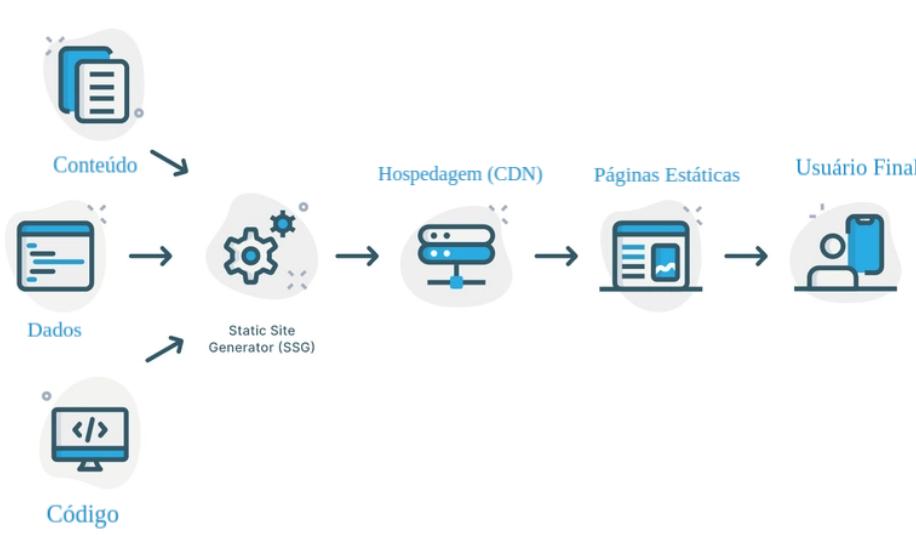
Segundo [Bose \(2022\)](#), a renderização no modelo **SSG** segue estas etapas principais:

1. Durante o processo de construção (build), o gerador de sites estáticos coleta dados de fontes como arquivos locais, *APIs* ou bancos de dados.
2. Esses dados são utilizados para gerar arquivos **HTML** completos para cada rota da aplicação.
3. Os arquivos gerados são armazenados e podem ser servidos diretamente por uma **CDN** (Content Delivery Network).

4. Quando o usuário acessa a aplicação, os arquivos estáticos são entregues instantaneamente, sem necessidade de renderização adicional.

Essa abordagem é ideal para páginas cujo conteúdo não muda com frequência, como blogs, documentações, portfólios e sites institucionais. Como os arquivos são pré-gerados, o tempo de resposta é extremamente rápido, e o **SEO** é favorecido, já que os mecanismos de busca encontram o conteúdo pronto para indexação.

Figura 6 – Etapas do método de geração estática de páginas (SSG)



Fonte: ([BOSE, 2022](#)) (adaptado)

Frameworks como Next.js, Gatsby, Hugo e Jekyll oferecem suporte completo ao **SSG**, integrando funcionalidades como roteamento dinâmico, *markdown*, e integração com CMSs. No exemplo a seguir, observa-se um documento **HTML** gerado estaticamente por meio de um processo de *build*:

```

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6     <title>Post: SSG Example</title>
7   </head>
8   <body>
9     <article>
10       <h1>Exemplo de página gerada com SSG</h1>
11       <p>Esse conteúdo foi gerado em tempo de build.</p>
12     </article>
13   </body>
14 </html>
  
```

Código 2.3 – Exemplo de HTML estático gerado com SSG

A principal limitação do **SSG** é a dificuldade em lidar com conteúdos altamente dinâmicos. Alterações nos dados requerem um novo processo de build para que as páginas sejam atualizadas, o que pode ser custoso em grandes aplicações ou com frequência de atualização elevada.

2.2.6 *Incremental Static Regeneration (ISR)*

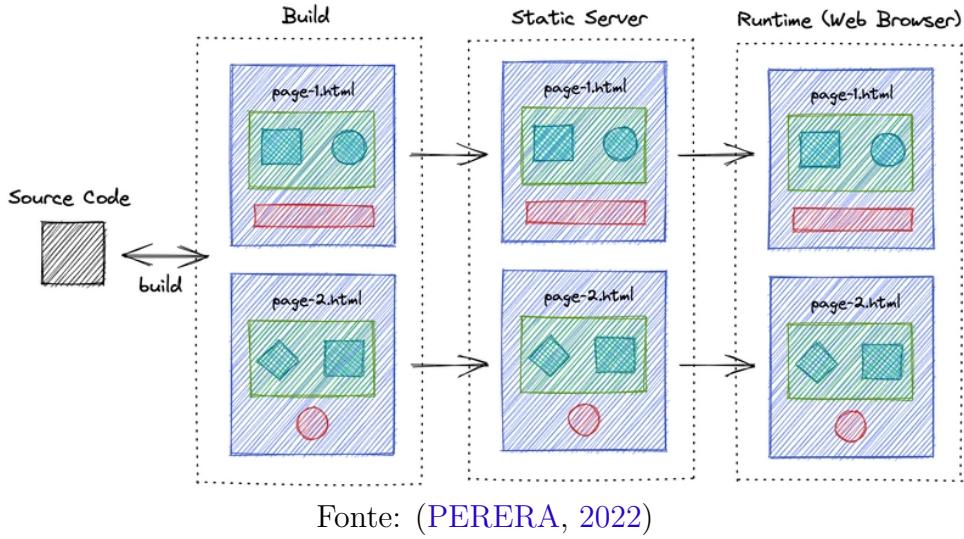
Incremental Static Regeneration (ISR) é uma estratégia híbrida introduzida por frameworks como o Next.js, que combina os benefícios da geração estática (**SSG**) com a flexibilidade de atualização dinâmica. Com **ISR**, as páginas são inicialmente geradas estaticamente em tempo de build, mas podem ser revalidadas e regeneradas no servidor de forma incremental e automática, com base em uma estratégia de tempo (ex: a cada 10 segundos) ou conforme novas requisições são feitas (PERERA, 2022).

De acordo com [Bose \(2022\)](#), o fluxo típico do **ISR** inclui as seguintes etapas:

1. No momento do build inicial, as páginas são geradas e armazenadas como arquivos estáticos.
2. Ao ser requisitada por um usuário, a página é entregue imediatamente, com o conteúdo pré-renderizado.
3. Se o tempo de revalidação definido (ex: `revalidate: 60`) tiver expirado, uma nova requisição ao backend é feita em segundo plano.
4. Essa nova versão da página é armazenada e substitui a anterior, sendo usada em acessos futuros.

Essa abordagem permite obter performance e **SEO** semelhantes ao **SSG**, mas com a vantagem de manter o conteúdo atualizado sem precisar de reconstruções manuais. Por isso, o **ISR** é ideal para sites que possuem atualizações regulares, porém não críticas em tempo real, como catálogos de produtos, blogs com comentários ou páginas de notícias.

Figura 7 – Funcionamento do modelo Incremental Static Regeneration (ISR)



Abaixo, um exemplo típico em Next.js:

```

1 // Função usada em getStaticProps
2 export async function getStaticProps() {
3     const res = await fetch('https://api.exemplo.com/posts')
4     const posts = await res.json()
5
6     return {
7         props: { posts },
8         revalidate: 60, // Página será regenerada a cada 60 segundos
9     }
10 }
```

Código 2.4 – Exemplo de revalidação de conteúdo com ISR no Next.js

O **ISR** representa um meio-termo eficiente entre a performance do **SSG** e a flexibilidade do **SSR**, oferecendo escalabilidade, **SEO** eficiente e atualização contínua do conteúdo, sem prejudicar a experiência do usuário.

2.2.7 Deferred Static Generation (DSG)

Deferred Static Generation (DSG) é uma extensão da estratégia de geração estática proposta pelo framework Gatsby. Essa abordagem permite que certas páginas do site sejam geradas sob demanda (ou seja, somente no momento em que forem requisitadas pela primeira vez) em vez de serem construídas durante o processo inicial de build, como ocorre no **SSG** tradicional ([Gatsby, 2023](#)).

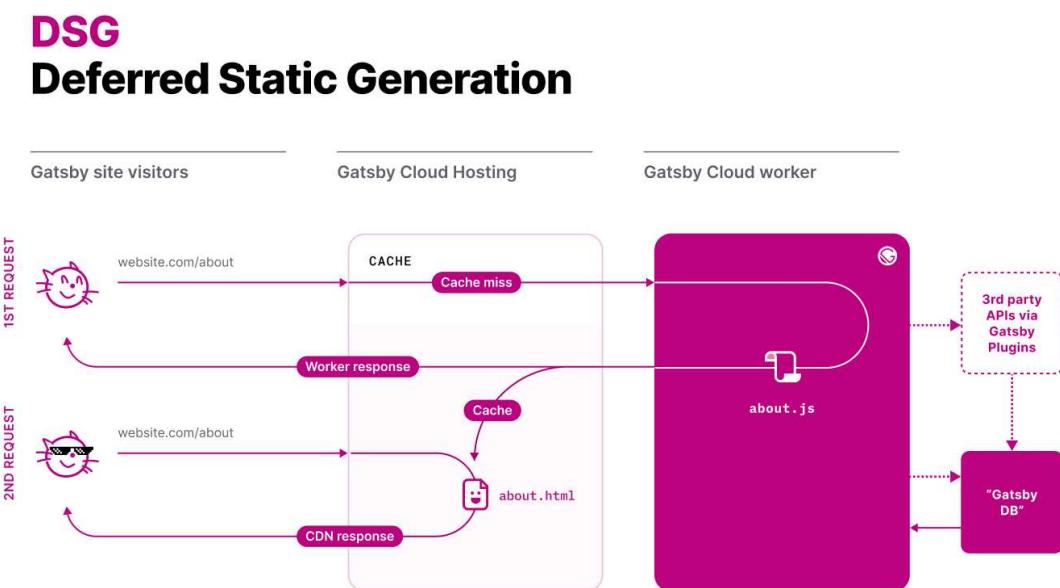
Segundo a [Gatsby \(2023\)](#), o **DSG** tem como principal vantagem a capacidade de reduzir significativamente o tempo de build em projetos com um número elevado de páginas, ao evitar a pré-renderização de rotas que têm baixo tráfego ou que não precisam

estar imediatamente disponíveis. Após a primeira solicitação, a página é armazenada em cache e, a partir daí, servida como conteúdo estático em acessos subsequentes.

O fluxo típico de geração diferida no DSG ocorre da seguinte forma:

1. Durante o processo de build, apenas páginas prioritárias são pré-geradas.
2. Páginas marcadas como `defer` são ignoradas temporariamente.
3. Quando uma página DSG é acessada pela primeira vez, um *worker* do Gatsby gera a versão HTML da página com base em um componente React (ex: `about.js`).
4. A página gerada é armazenada em cache e servida como conteúdo estático nas próximas requisições.

Figura 8 – Funcionamento da estratégia Deferred Static Generation (DSG)



Fonte: ([Gatsby, 2023](#))

Na [Figura 8](#), observa-se que, ao acessar uma página como `/about` pela primeira vez, ocorre um *cache miss*, e um *worker* do Gatsby é acionado para gerar o arquivo `about.html` a partir do componente correspondente (`about.js`). Esse conteúdo pode incluir dados extraídos de plugins, *APIs* de terceiros ou do próprio *Gatsby DB*. Após o processamento, a resposta é enviada ao usuário e armazenada em cache. Requisições futuras à mesma rota são atendidas diretamente pela CDN, com desempenho semelhante ao de páginas SSG.

Segundo [Anurag \(2024\)](#), a estratégia do DSG é especialmente útil para projetos com milhares de páginas cujo acesso é desigual. Exemplos incluem catálogos de produtos legados, páginas de arquivos antigos ou conteúdos gerados a partir de sistemas CMS.

A seguir, um exemplo de configuração de uma rota com DSG em Gatsby:

```

1 // gatsby-node.js
2 exports.createPages = async ({ actions }) => {
3   const { createPage } = actions
4   createPage({
5     path: '/produto/exemplo',
6     component: require.resolve('../src/templates/produto.js'),
7     context: { id: 'produto-exemplo' },
8     defer: true, // ativa DSG
9   })
10 }

```

Código 2.5 – Exemplo de definição de página DSG no Gatsby

Assim como o ISR, o DSG busca combinar desempenho, escalabilidade e atualizações eficientes. No entanto, sua particularidade está em adiar completamente o custo da renderização até o momento da primeira requisição, tornando-o ideal em contextos onde o tempo de build precisa ser otimizado sem prejudicar a entrega do conteúdo a longo prazo.

2.3 Métricas e Ferramentas de Análise de Desempenho

A avaliação de desempenho em aplicações web modernas é um pilar fundamental para garantir a qualidade da experiência do usuário e o sucesso de um projeto. A análise vai muito além do simples tempo de carregamento total, abrangendo a percepção de velocidade, a capacidade de resposta a interações e a estabilidade visual da interface. Para quantificar esses aspectos de forma padronizada e acionável, o ecossistema de desenvolvimento web dispõe de um conjunto robusto de métricas e ferramentas, que foram essenciais para a coleta de dados e a análise comparativa deste trabalho.

2.3.1 Ferramentas de Análise de Mercado

Existem diversas ferramentas consolidadas para a análise de performance web. Soluções como o PageSpeed Insights, GTmetrix e WebPageTest oferecem diagnósticos detalhados sobre a velocidade de carregamento e a experiência do usuário, simulando diferentes condições de rede, dispositivos e localizações geográficas (Web Absoluta, 2024). Enquanto o PageSpeed Insights se destaca por integrar dados de campo (do Chrome User Experience Report) e de laboratório, o GTmetrix é conhecido por seus relatórios detalhados com gráficos em cascata (*waterfall charts*) que ajudam a identificar gargalos no carregamento de recursos (GTmetrix, 2025). O WebPageTest, por sua vez, oferece uma flexibilidade granular nos testes. Para este trabalho, a análise foi centrada em duas iniciativas principais do Google: o Lighthouse e, principalmente, os Core Web Vitals.

2.3.2 Google Lighthouse

O Google Lighthouse é uma ferramenta de auditoria automatizada e de código aberto, integrada às ferramentas de desenvolvedor do Google Chrome. Ele é projetado para analisar a qualidade de páginas web em cinco categorias principais: Performance, Acessibilidade, Boas Práticas, SEO e *Progressive Web App* (PWA). Ao final da análise, o Lighthouse gera um relatório com pontuações de 0 a 100 para cada categoria e apresenta um conjunto de oportunidades e diagnósticos açãoáveis. No contexto deste estudo, o Lighthouse foi utilizado como uma ferramenta de apoio diagnóstico em ambiente de laboratório, essencial para identificar gargalos de renderização, analisar o tempo de bloqueio da *thread* principal (*Total Blocking Time*) e validar aspectos técnicos de SEO e acessibilidade das aplicações desenvolvidas.

2.3.3 Core Web Vitals

O pilar da coleta de dados empíricos deste trabalho foi o conjunto de métricas conhecido como Core Web Vitals. Proposto pelo Google, este subconjunto dos Web Vitals foi projetado para medir a experiência do usuário no mundo real (*field data*), focando em três aspectos que impactam diretamente a percepção do usuário: velocidade de carregamento, interatividade e estabilidade visual (OSMANI; MILLER, 2025). A importância dessas métricas é reforçada pelo fato de serem um fator de ranqueamento nos resultados de busca do Google (Google Search Central, 2020).

2.3.3.1 Principais Métricas do Web Vitals

As métricas a seguir, que compõem os Core Web Vitals atuais, foram essenciais para a análise comparativa:

Largest Contentful Paint (LCP) Mede o tempo, em segundos, desde o início do carregamento da página até a renderização do maior elemento de conteúdo (imagem ou bloco de texto) visível na tela. É a principal métrica para a percepção de velocidade de carregamento.

Interaction to Next Paint (INP) Avalia a latência de todas as interações do usuário (cliques, toques e digitação) com a página, medindo a capacidade de resposta geral da aplicação. Um baixo INP indica que a página responde rapidamente às ações do usuário.

Cumulative Layout Shift (CLS) Quantifica a instabilidade visual da página. Mede a soma de todas as mudanças inesperadas de layout que ocorrem durante o carregamento, garantindo que os elementos não "pulem" na tela e frustram a interação do usuário.

2.3.3.2 Métricas Complementares

Além dos Core Web Vitals, outras métricas foram observadas para fornecer um diagnóstico mais completo do processo de carregamento:

Time to First Byte (TTFB) Indica a responsividade do servidor. Mede o tempo entre a requisição de um recurso e o momento em que o primeiro byte da resposta chega ao navegador. É um indicador crucial da saúde do *backend*.

First Contentful Paint (FCP) Marca o tempo em que o primeiro conteúdo do DOM (texto, imagem, etc.) é renderizado na tela, dando ao usuário o primeiro feedback visual de que a página está carregando.

Time to First Byte (TTFB) Medido em ambiente de laboratório (via Lighthouse), soma o tempo em que a *thread* principal ficou bloqueada por tarefas longas, impedindo a resposta a interações do usuário. É um bom proxy para a métrica de campo INP.

O domínio e a correta interpretação dessas métricas foram fundamentais para a análise comparativa entre as arquiteturas CSR e SSR, permitindo uma compreensão aprofundada do desempenho percebido pelo usuário e servindo como base para as conclusões técnicas apresentadas neste estudo.

2.4 Frameworks Web

O uso de bibliotecas e frameworks no desenvolvimento web moderno proporciona ganhos significativos de produtividade, desempenho e organização de código. Eles abstraem operações complexas e oferecem estruturas padronizadas para construção de aplicações escaláveis. A escolha da ferramenta está diretamente relacionada à abordagem de renderização adotada, seja no lado do cliente (CSR) ou do servidor (SSR).

2.4.1 Bibliotecas JavaScript

Bibliotecas JavaScript são conjuntos de funcionalidades reutilizáveis que fornecem recursos específicos para o desenvolvimento de aplicações web. Elas diferem dos frameworks por não imporem uma estrutura rígida, oferecendo maior flexibilidade ao desenvolvedor. A seguir, são apresentadas algumas das bibliotecas mais relevantes no contexto de renderização do lado do cliente (CSR):

- React: Desenvolvida pelo Facebook, React é uma biblioteca declarativa focada na construção de interfaces de usuário por meio de componentes reutilizáveis. Seu uso do *virtual DOM* permite renderizações mais eficientes, tornando-a ideal para

aplicações interativas com alto desempenho. Apesar de ser comumente chamada de framework, React atua apenas na camada de visualização, exigindo bibliotecas complementares para roteamento e gerenciamento de estado ([REACT, 2025](#)).

- jQuery: Uma das bibliotecas mais populares da era inicial do JavaScript moderno, jQuery simplifica tarefas comuns como manipulação do DOM, tratamento de eventos e requisições AJAX. Embora sua popularidade tenha diminuído com o surgimento de bibliotecas mais modernas e frameworks reativos, ela ainda é amplamente utilizada em sistemas legados e aplicações de menor complexidade ([FOUNDATION, 2025](#)).
- Alpine.js: Alpine.js é uma biblioteca leve e reativa voltada para a manipulação de componentes diretamente no HTML, com uma sintaxe declarativa inspirada em Vue.js. Ela é particularmente útil para adicionar interatividade a páginas estáticas ou aplicações simples, sendo uma alternativa eficiente em cenários onde o uso de bibliotecas maiores seria excessivo ([PORZIO, 2023](#)).

2.4.2 Frameworks para CSR

No modelo [CSR](#), a renderização da interface é realizada diretamente no navegador do usuário, após o carregamento dos arquivos JavaScript. Frameworks como os listados a seguir são amplamente utilizados para implementar essa abordagem:

- Vue.js: framework progressivo para construção de interfaces web interativas. Seu foco está na camada de visualização, com curva de aprendizado acessível e estrutura modular ([VUE, 2025](#)).
- Angular: framework completo mantido pelo Google, baseado em TypeScript, que oferece arquitetura robusta e recursos integrados como injecção de dependência e roteamento ([ANGULAR, 2025](#)).
- Svelte: framework que realiza a compilação dos componentes no momento do build, eliminando a necessidade de um *virtual DOM*, o que reduz o tempo de carregamento e o uso de recursos do navegador ([SVELTE, 2025](#)).

Esses frameworks tornam o desenvolvimento com [CSR](#) mais eficiente e sustentável, proporcionando experiências ricas ao usuário com foco em interatividade e responsividade.

2.4.3 Meta-frameworks para SSR

Para aplicações com foco em renderização no lado do servidor, os *meta-frameworks* oferecem soluções completas, otimizando tanto o desempenho inicial quanto a indexabilidade em mecanismos de busca. Eles operam sobre frameworks tradicionais (como Vue

ou Svelte) ou bibliotecas (como React), incorporando funcionalidades essenciais como roteamento, pré-renderização, recuperação de dados e *hydration*.

- Next.js: baseado em React, fornece recursos para [SSR](#), geração de sites estáticos e suporte a APIs integradas ([NEXT, 2024](#)).
- Nuxt.js: extensão do Vue.js que oferece SSR, geração estática e arquitetura modular ([NUXT, 2024](#)).
- SvelteKit: baseado em Svelte, permite renderização no servidor e no cliente, com foco em simplicidade e desempenho ([SVELTEKIT, 2024](#)).
- Angular Universal: solução oficial para SSR em Angular, melhora a indexação e o tempo de carregamento inicial ([ANGULAR2024, 2024](#)).
- Remix: framework full-stack para React que adota um modelo de dados centrado em carregadores e ações ([REMIX, 2024](#)).
- Astro: framework moderno que carrega apenas o JavaScript necessário, permitindo uso híbrido de componentes React, Vue, Svelte e outros ([ASTRO, 2024](#)).
- Qwik: introduz o conceito de aplicações *resumíveis*, com SSR e carregamento progressivo de interatividade ([QWIK, 2024](#)).

Esses meta-frameworks são especialmente indicados para aplicações que priorizam SEO, acessibilidade e desempenho no primeiro carregamento, como páginas institucionais, lojas virtuais e blogs.

2.4.4 Comparativo entre Frameworks CSR e SSR

A escolha entre frameworks focados em [CSR](#) e meta-frameworks que implementam [SSR](#) é uma decisão arquitetônica fundamental, com implicações diretas na performance, na experiência do usuário, no [SEO](#) e na complexidade da infraestrutura. Enquanto frameworks como Vue.js, Angular e Svelte (em sua forma pura) se destacam na criação de interfaces ricas e dinâmicas no cliente, os meta-frameworks como Next.js, Nuxt.js e SvelteKit foram projetados para otimizar o carregamento inicial e a indexabilidade por meio da renderização no servidor.

O quadro a seguir sintetiza os principais pontos de contraste entre essas duas abordagens, destacando seus respectivos pontos fortes e casos de uso ideais, de modo a orientar a seleção da ferramenta mais adequada para os requisitos de cada projeto.

Tabela 2 – Comparação entre frameworks para CSR e SSR

Critério	Frameworks CSR (Vue, Angular, Svelte)	Meta-frameworks SSR (Next.js, Nuxt, SvelteKit, etc.)
Renderização Inicial	O conteúdo é montado no navegador após o carregamento do JavaScript	O conteúdo é gerado no servidor e entregue já renderizado ao navegador
Tempo de Carregamento	Maior tempo de carregamento inicial (dependente do JS)	Melhor desempenho no carregamento inicial (TTFB menor)
SEO	Pode ser limitado, pois bots podem não processar JavaScript adequadamente	Excelente, já que o HTML completo está disponível para rastreadores
Interatividade	Alta, com foco em aplicações ricas e dinâmicas	Boa, com necessidade de <i>hydration</i> após o carregamento
Complexidade de Infraestrutura	Menor, geralmente servido por CDNs e arquivos estáticos	Maior, exige servidores para processar cada requisição
Casos de Uso Ideais	SPAs, dashboards, aplicações com muitas interações em tempo real	Landing pages, blogs, e-commerce, sites que dependem de SEO

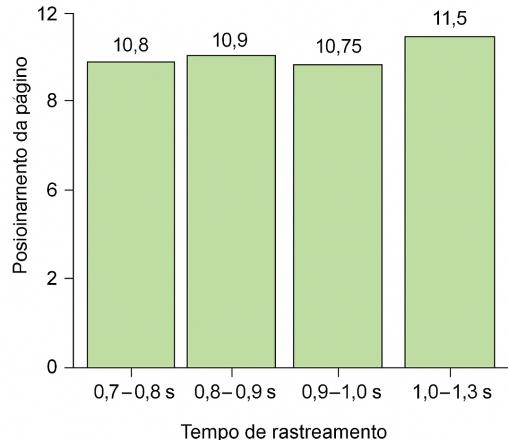
2.5 Experiência do Usuário

A **User Experience (UX)** é um aspecto crítico no desenvolvimento de aplicações web, influenciando diretamente a satisfação e a eficácia da interação do usuário com o sistema (**EMADAMERHO-ATORI, 2023**). Para alcançar uma **UX** satisfatória, a escolha entre **CSR** e **SSR** deve considerar fatores como: **SEO**, velocidade de carregamento, interatividade e acessibilidade. Conforme **Emadamerho-Atori (2024)**, a **UX** vai além da interface gráfica, englobando toda a jornada do usuário desde a navegação até a conclusão de tarefas.

2.5.1 *Search Engine Optimization (SEO)*

O **Search Engine Optimization (SEO)** consiste em um conjunto integrado de práticas de otimização, tanto no aspecto técnico quanto no de conteúdo, com três objetivos principais: maximizar a visibilidade orgânica nos mecanismos de busca, posicionar estratégicamente páginas-chave e garantir uma experiência de usuário qualificada durante o processo de busca. Essas práticas são essenciais para garantir que o conteúdo de um site seja facilmente encontrado e indexado pelos motores de busca, aumentando a probabilidade de atrair visitantes qualificados. Entre os fatores mais conhecidos, destaca-se a velocidade de carregamento da página, que impacta diretamente a experiência do usuário e a classificação nos resultados de busca (**MURPHY, 2022**).

Figura 9 – Tempo de Rastreamento e Posicionamento da Página



Fonte: ([WAGNER, 2016](#))(adaptado)

A [Figura 9](#) ilustra a relação entre o tempo de rastreamento e o posicionamento da página. O tempo de rastreamento refere-se ao tempo que os mecanismos de busca levam para acessar e indexar uma página. Quanto mais rápido o tempo de rastreamento, maior a probabilidade de a página ser indexada rapidamente e, consequentemente, melhor seu posicionamento nos resultados de busca. Isso destaca a importância de otimizar o desempenho do site para garantir uma boa classificação nos motores de busca.

2.5.2 *Velocidade de carregamento*

A velocidade de carregamento de uma página refere-se ao tempo necessário para que todo o seu conteúdo esteja visível e interativo no navegador, desde a solicitação inicial do usuário. Esse tempo pode ser influenciado por diversos fatores, como o tamanho dos arquivos, a complexidade do conteúdo, a qualidade da conexão com a internet e o desempenho do servidor ([SHOPIFY, 2024](#)).

Esse fator é determinante tanto para a [UX](#) quanto para o [SEO](#). Páginas que carregam rapidamente tendem a apresentar menores taxas de rejeição e melhores resultados em métricas de conversão. Além disso, o [SEO](#) utiliza a velocidade de carregamento como um dos critérios de ranqueamento nos mecanismos de busca ([MURPHY, 2022](#)).

De acordo com [Google \(2010\)](#), quanto mais rápido um site carregar, melhor tende a ser a experiência do usuário. Sites lentos comprometem a navegação e reduzem o tempo de permanência, afetando negativamente o engajamento.

A percepção de desempenho muitas vezes chamada de *velocidade percebida* também é um fator crucial para a usabilidade e pode ser tão relevante quanto o tempo de carregamento real. Nesse sentido, a escolha entre [SSR](#) e [CSR](#) influencia diretamente

essa percepção. O **SSR** geralmente proporciona carregamento inicial mais rápido, pois o conteúdo é renderizado no servidor e entregue ao navegador já pronto para exibição. Isso permite que os usuários visualizem o conteúdo principal imediatamente, mesmo que outros recursos ainda estejam sendo carregados (**EMADAMERHO-ATORI, 2024**).

Por outro lado, no **CSR**, o navegador precisa baixar, interpretar e executar o JavaScript antes de renderizar qualquer conteúdo. Isso pode resultar em uma exibição inicial em branco ou em telas de carregamento, o que compromete a percepção de desempenho especialmente em conexões lentas ou dispositivos com menor capacidade de processamento (**STUDIO, 2023b**).

2.5.3 Interatividade

A interatividade é um fator decisivo na experiência do usuário em aplicações web modernas, pois determina a forma como os usuários percebem a continuidade e a capacidade de resposta durante a navegação. Nas aplicações que utilizam **CSR**, o código JavaScript é executado diretamente no navegador, permitindo respostas imediatas a interações como cliques, preenchimento de formulários ou navegação entre páginas internas. Essa abordagem possibilita transições de página mais suaves e experiências semelhantes às de aplicativos nativos, sem a necessidade de recarregamentos completos (**STUDIO, 2023b**).

Segundo **Emadamerho-Atori (2024)**, a renderização no lado do cliente favorece experiências altamente dinâmicas, oferecendo um nível elevado de controle sobre os elementos da interface. Em contrapartida, o **SSR**, embora proporcione carregamento inicial mais rápido e visibilidade imediata do conteúdo, apresenta limitações em termos de interatividade. Alterações na interface em aplicações **SSR** geralmente demandam comunicações adicionais com o servidor, o que pode comprometer a continuidade da experiência do usuário (**EMADAMERHO-ATORI, 2024; WATTS, 2023**).

Para mitigar essas limitações, abordagens híbridas têm sido amplamente adotadas. Nelas, o conteúdo é inicialmente renderizado no servidor e, posteriormente, reativado no cliente com JavaScript, em uma estratégia conhecida como *hydration* (**WATTS, 2023**). Essa técnica busca aliar os benefícios de desempenho e **SEO** do **SSR** com a interatividade aprimorada do **CSR**.

2.5.4 Acessibilidade

A acessibilidade em aplicações web refere-se à capacidade de tornar conteúdos e funcionalidades utilizáveis por pessoas com deficiência, como visual, auditiva, motora ou cognitiva. É um princípio essencial para garantir a equidade no acesso à informação e à interação digital. De acordo com (**STUDIO, 2023a**), acessibilidade diz respeito a assegurar

que todos, independentemente de suas habilidades, possam acessar e interagir com o conteúdo da web. Para pessoas com deficiência, isso pode significar o uso de leitores de tela, navegação por teclado ou a dependência de outras tecnologias assistivas.

As abordagens de renderização, como [CSR](#) e [SSR](#), impactam diretamente a acessibilidade, especialmente na compatibilidade com essas tecnologias. Em aplicações que utilizam [CSR](#), o conteúdo geralmente é carregado de forma assíncrona após a execução do JavaScript, o que pode dificultar a leitura imediata por leitores de tela que dependem de uma estrutura HTML previamente carregada para interpretar a página corretamente ([STUDIO, 2023a](#)). Já no [SSR](#), o conteúdo é entregue completamente no carregamento inicial, facilitando a interpretação por essas ferramentas e proporcionando uma experiência mais estável para usuários com deficiência visual ([EMADAMERHO-ATORI, 2024](#)).

Além disso, em contextos com atualizações dinâmicas de conteúdo como ocorre em SPAs com [CSR](#) é necessário adotar práticas específicas para garantir a acessibilidade, como gerenciamento de foco, uso de alertas ARIA e atualização de leitores de tela após mudanças no DOM. Essas medidas são fundamentais para que as mudanças de visualização sejam percebidas corretamente por tecnologias assistivas, uma vez que alterações no DOM nem sempre são reconhecidas automaticamente por leitores de tela. O envio de foco a elementos interativos ou o uso de regiões ARIA ao vivo são técnicas recomendadas para anunciar mudanças de estado ao usuário ([SUTTON, 2018](#)).

Assim, embora o [SSR](#) ofereça uma base naturalmente mais acessível, ambas as abordagens podem ser igualmente inclusivas quando aplicadas com atenção às diretrizes e boas práticas de acessibilidade.

2.6 Ferramentas Modernas para Prototipação e Interfaces

Nesta seção são apresentadas duas ferramentas inovadoras utilizadas para acelerar o desenvolvimento frontend: a plataforma v0 e a biblioteca de componentes shadcn/ui. Ambas representam abordagens modernas para construção de interfaces dinâmicas, acessíveis e escaláveis.

2.6.1 v0

A v0 é uma plataforma assistida por Inteligência Artificial projetada para transformar descrições em linguagem natural em aplicações web funcionais. A partir de prompts descritivos, a ferramenta gera código utilizando tecnologias modernas como React, Next.js e Tailwind CSS, permitindo prototipação rápida e iteração sobre interfaces ([v0, 2025](#)). O fluxo básico inclui: escrever a ideia em texto, gerar a interface automaticamente, revisar e ajustar os elementos e exportar o código para integração no projeto.

2.6.2 shadcn/ui

O shadcn/ui é uma biblioteca de componentes open source baseada em Radix UI e estilizada com Tailwind CSS. Ao contrário de bibliotecas tradicionais, os componentes do shadcn/ui são copiados diretamente para o projeto, oferecendo ao desenvolvedor total controle sobre o código e possibilitando personalizações profundas ([shadcn/ui, 2025](#)). Essa abordagem favorece flexibilidade e consistência visual no desenvolvimento de aplicações modernas.

2.6.3 Integração entre v0 e shadcn/ui

A integração entre v0 e shadcn/ui proporciona uma experiência otimizada para geração de interfaces. Ao utilizar a plataforma v0, os componentes gerados são construídos com base na arquitetura do shadcn/ui, incluindo as práticas recomendadas de acessibilidade e estilização com Tailwind CSS ([v0, 2025](#); [shadcn/ui, 2025](#)). Essa integração permite que desenvolvedores iniciem com protótipos gerados automaticamente e, em seguida, façam ajustes diretamente no código dos componentes, mantendo um alto grau de personalização e performance. Além disso, ela favorece o uso de estratégias modernas como [SSR](#) e [CSR](#), oferecendo compatibilidade com frameworks como Next.js e Vite.

2.7 Processo de Desenvolvimento e Controle de Gestão de Software

O desenvolvimento de aplicações web modernas exige práticas que garantam organização, rastreabilidade e colaboração eficiente entre desenvolvedores. Nesse contexto, ferramentas de controle de versão como o Git e plataformas de hospedagem como o GitHub são fundamentais para o gerenciamento do ciclo de vida do software ([GitHub, 2025a](#)).

2.7.1 Git

O Git é um sistema de controle de versão distribuído criado por Linus Torvalds em 2005 com o objetivo de gerenciar projetos de forma rápida e eficiente, independentemente do tamanho ou complexidade ([CHACON; STRAUB, 2014](#)). Ele permite que múltiplos desenvolvedores trabalhem simultaneamente em um projeto, mantendo o histórico de alterações de forma segura e auditável.

Entre os conceitos fundamentais do Git, destacam-se:

- Reppositório (*Repository*): Estrutura que armazena o histórico completo do projeto, incluindo arquivos, diretórios e suas versões ao longo do tempo.
- Commits: Registros de alterações no projeto. Cada commit possui um identificador único (hash) e uma mensagem descritiva ([CHACON; STRAUB, 2014](#)).

- Branches: Ramificações independentes do repositório que permitem o desenvolvimento paralelo de funcionalidades, correções ou experimentos sem impactar o código principal.
- Merge: Integração de alterações realizadas em diferentes branches.
- Clone e Pull: Operações que permitem obter uma cópia local do repositório e sincronizar com atualizações remotas.

O modelo distribuído do Git permite que cada colaborador mantenha uma cópia completa do repositório em sua máquina local, o que garante maior resiliência e independência em relação ao servidor central (CHACON; STRAUB, 2014).

2.7.2 GitHub

O GitHub é uma plataforma de hospedagem de código baseada em Git, que amplia suas funcionalidades com recursos colaborativos e integração contínua (GitHub, 2025a). Fundada em 2008, a plataforma popularizou-se como um ambiente de colaboração para projetos de software de código aberto e privado.

Além de hospedar repositórios Git, o GitHub oferece funcionalidades como:

- Issues: Ferramenta para rastreamento de tarefas, bugs e melhorias (GitHub, 2025a).
- Pull Requests (PR): Fluxo de revisão e integração de código, permitindo que contribuições sejam analisadas antes de serem incorporadas ao branch principal.
- Actions: Automatização de processos com integração contínua (CI) e entrega contínua (CD).
- Wikis e Documentação: Área para criação de páginas informativas sobre o projeto.

2.7.3 GitHub Projects

O GitHub Projects é uma funcionalidade integrada à plataforma que possibilita o gerenciamento de projetos utilizando quadros visuais baseados em metodologias ágeis, como Kanban e Scrum (GitHub, 2025b). Essa ferramenta permite organizar tarefas, acompanhar o progresso e priorizar demandas de maneira colaborativa.

Entre os recursos oferecidos pelo GitHub Projects, destacam-se:

- Quadros Kanban: Visualização de tarefas em colunas (como *To Do*, *In Progress* e *Done*), facilitando o acompanhamento do fluxo de trabalho.

- Automação: Regras automáticas que movem cartões entre colunas com base em eventos, como o fechamento de issues ou merge de pull requests ([GitHub, 2025b](#)).
- Customização: Campos personalizados e filtros para adaptar o quadro às necessidades do projeto.
- Integração com Issues e Pull Requests: Possibilidade de vincular tarefas diretamente ao código, permitindo rastreabilidade completa entre planejamento e implementação.

2.8 News API

A News API é uma interface de programação que disponibiliza fluxos de notícias e artigos de forma estruturada, permitindo o acesso a conteúdos de mais de 150 000 fontes jornalísticas ao redor do mundo. Seu propósito é fornecer informações publicadas em tempo real, facilitando o desenvolvimento de soluções de agregação, análise e visualização de notícias ([News API, 2025](#)).

2.8.1 Escopo e cobertura

A plataforma oferece dois principais tipos de consulta:

- Busca por artigos (`/everything`): permite pesquisar notícias com base em termos, operadores booleanos, intervalos de datas, domínios específicos e critérios de ordenação, como relevância, popularidade ou data de publicação ([News API, 2025](#)).
- Manchetes principais (`/top-headlines`): retorna as notícias mais recentes, com filtros por país, categoria jornalística e fontes específicas.

Além disso, a News API disponibiliza o endpoint `/sources`, que fornece metadados sobre os veículos indexados, como nome, idioma, categoria e URL oficial.

2.8.2 Autenticação e formato de resposta

Para utilizar os endpoints, é necessário um `key`, que autentica as requisições e controla o uso da plataforma. As respostas são retornadas no formato `JSON`, contendo campos como `status`, `totalResults` e uma lista de objetos `articles`, com atributos como título, autor, descrição, e data de publicação ([News API, 2025](#)).

2.8.3 Funcionalidade e integração

A News API suporta buscas avançadas com:

- termos exatos ou operadores lógicos (`AND`, `OR`, `NOT`);
- filtros por idioma, domínios específicos e intervalos de datas (`from`, `to`);
- paginação e controle de volume de resultados, com limite máximo de 100 artigos por página.

Essas características permitem integração tanto em aplicações com [CSR](#), que realizam chamadas diretamente do navegador após o carregamento, quanto em soluções com [SSR](#), onde as requisições são feitas na camada servidor, possibilitando que o conteúdo seja renderizado previamente antes de ser entregue ao cliente.

2.9 Infraestrutura de Contêineres e Docker

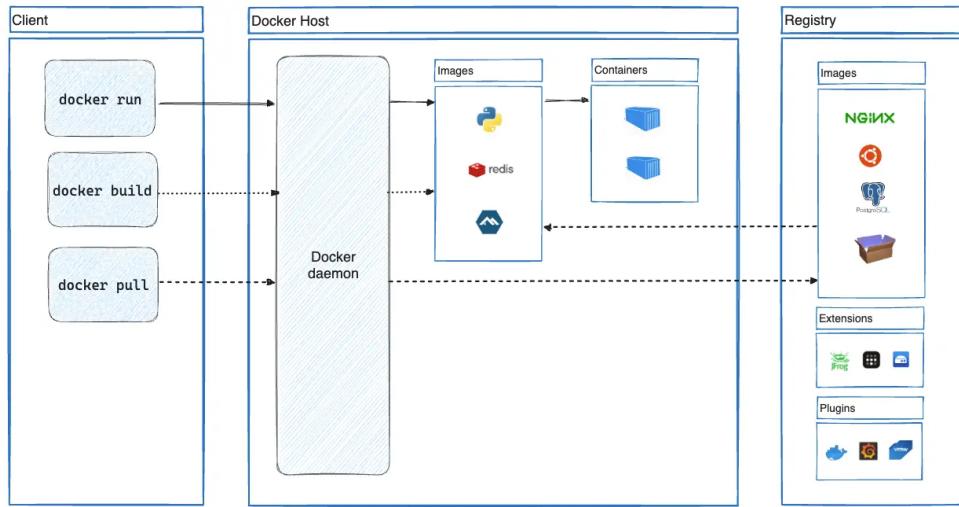
Este capítulo apresenta os conceitos fundamentais de contêineres e da plataforma Docker, abordando sua arquitetura, seus principais objetos e o processo de construção de imagens com o Dockerfile. Adicionalmente, são discutidas boas práticas de otimização e a integração dessa tecnologia com o estudo comparativo entre [CSR](#) e [SSR](#). ([Docker, Inc., 2025b](#); [Docker, Inc., 2025a](#))

2.9.1 Fundamentos da Plataforma Docker

O Docker é uma plataforma de código aberto que viabiliza o desenvolvimento, a distribuição e a execução de aplicações em ambientes isolados denominados contêineres. Esses contêineres são leves, portáteis e auto-suficientes, pois empacotam a aplicação com todas as suas dependências, bibliotecas e configurações. Essa abordagem padroniza os ambientes de execução, reduzindo inconsistências entre as máquinas de desenvolvimento, teste e produção, e acelera os ciclos de integração e entrega contínua (/) ([Docker, Inc., 2025b](#)).

A arquitetura do Docker segue o modelo cliente-servidor, onde o cliente (`docker CLI`) envia comandos para o serviço (daemon `dockerd`), que gerencia a construção, execução e distribuição dos objetos Docker. A comunicação ocorre por meio de uma REST, via *Unix sockets* ou interface de rede. Para facilitar a distribuição, as imagens são armazenadas em *registries*, como o Docker Hub (público) ou registros privados, permitindo a portabilidade entre ambientes locais, datacenters e nuvens ([Docker, Inc., 2025b](#)).

Figura 10 – Arquitetura simplificada do Docker (cliente, daemon, registries e objetos).



Fonte: Adaptado da documentação oficial do Docker ([Docker, Inc., 2025b](#)).

Os principais objetos manipulados pela plataforma são as imagens e os contêineres. Uma imagem é um modelo (*template*) imutável, composto por um sistema de arquivos em camadas (*layers*), onde cada camada representa uma instrução do processo de construção. Um contêiner, por sua vez, é uma instância executável e isolada de uma imagem. Ele pode ser criado, iniciado, parado e removido, e suas alterações internas são, por padrão, efêmeras, sendo descartadas ao final de seu ciclo de vida, a menos que se utilizem volumes para persistência de dados ([Docker, Inc., 2025b](#)).

2.9.2 Construção de Imagens com Dockerfile

A construção de imagens é definida por meio de um Dockerfile, um arquivo de texto com um conjunto de instruções sequenciais. Cada instrução gera uma nova camada na imagem, otimizando reconstruções futuras por meio de cache. As instruções essenciais incluem `FROM` para definir a imagem base; `RUN` para executar comandos de instalação ou compilação; `COPY` para adicionar arquivos do contexto local; e `CMD` ou `ENTRYPOINT` para especificar o comando de execução padrão do contêiner. Outras instruções como `WORKDIR`, `ENV` e `EXPOSE` configuram o ambiente de execução ([Docker, Inc., 2025a](#)). Para otimizar o processo, o arquivo `.dockerignore` é utilizado para excluir arquivos e diretórios desnecessários do contexto de build, reduzindo o tamanho da imagem e o tempo de construção.

2.9.3 Boas Práticas e Otimização de Builds

Para criar imagens eficientes, seguras e enxutas, é fundamental adotar boas práticas. A principal delas é o uso de multi-stage builds, uma técnica que permite utilizar múltiplos

estágios FROM em um mesmo Dockerfile. Com isso, é possível compilar a aplicação em um estágio com todas as ferramentas de desenvolvimento (*toolchain*) e copiar apenas os artefatos finais para um estágio de produção mínimo, baseado em uma imagem enxuta como a `alpine` ou `scratch`. O resultado é uma imagem final drasticamente menor, com superfície de ataque reduzida e mais rápida para distribuir (Docker, Inc., 2025a).

```

1 # syntax=docker/dockerfile:1
2 # Estgio de build
3 FROM node:20-alpine AS build
4 WORKDIR /app
5 COPY package*.json .
6 RUN npm ci --omit=dev
7 COPY ..
8 RUN npm run build
9
10 # Estgio de produo (runtime)
11 FROM node:20-alpine AS runtime
12 WORKDIR /app
13 ENV NODE_ENV=production
14 COPY --from=build /app/.next ./next
15 COPY --from=build /app/package*.json .
16 EXPOSE 3000
17 CMD ["node", ".next/standalone/server.js"]

```

Código 2.6 – Exemplo de multi-stage build para aplicação SSR com Node/Next.js

A otimização do cache de build também é crucial. Como cada instrução gera uma camada, a ordem no Dockerfile deve ser da menos para a mais volátil, permitindo que o cache seja aproveitado ao máximo. Para operações que frequentemente mudam, como a instalação de pacotes, o motor de build moderno (BuildKit) oferece montagens de cache avançadas, como `RUN -mount=type=cache`, que persistem o cache de gerenciadores de pacotes sem invalidar as camadas da imagem. Além disso, a segurança deve ser uma prioridade: dados sensíveis, como tokens e chaves SSH, nunca devem ser inseridos na imagem via `ENV` ou `ARG`. A abordagem correta é utilizar montagens de segredos, como `RUN -mount=type=secret`, que disponibilizam as credenciais apenas durante o build, sem armazená-las nas camadas finais (Docker, Inc., 2025a).

2.9.4 Padrões de Infraestrutura para Aplicações CSR e SSR

A operacionalização de aplicações containerizadas segue um ciclo de vida bem definido, que envolve as etapas de construir a imagem (`docker build`), publicá-la em um registry (`docker push`) e executá-la (`docker run`). Nessa última etapa, são configurados aspectos da infraestrutura como mapeamento de portas, volumes para dados persistentes e conexões de rede. A forma como essa infraestrutura é configurada varia significativamente conforme a arquitetura de renderização.

No contexto deste estudo, dois padrões de infraestrutura se destacam. Para aplicações CSR ou SSG, que resultam em um conjunto de arquivos estáticos (HTML, CSS, JS), o padrão é utilizar um servidor web leve e de alta performance, como o Nginx, cuja única responsabilidade é servir esses arquivos eficientemente.

```
1 # Padrão de infraestrutura para aplicação CSR: servir arquivos estáticos com
2 # Nginx
3 FROM nginx:1.27-alpine
4 COPY ./dist/ /usr/share/nginx/html
5 # (Opcional) Copiar nginx.conf customizado para rotas, etc.
6 # COPY ./nginx.conf /etc/nginx/conf.d/default.conf
7 EXPOSE 80
```

Código 2.7 – Servidor estático com Nginx para aplicação CSR/SSG

Por outro lado, para aplicações SSR, a infraestrutura requer um ambiente de execução de servidor, como o Node.js, para processar as requisições e renderizar as páginas dinamicamente. A containerização, nesse caso, garante que todo o ambiente e suas dependências estejam empacotados e prontos para execução. Conforme apresentado neste capítulo, o Docker fornece uma base consistente para essa tarefa, com isolamento leve, portabilidade e um ciclo de vida claro (construir, distribuir, executar). O Dockerfile, aliado a boas práticas como *multi-stage builds* e otimização de cache, viabiliza imagens menores, builds mais rápidos e maior segurança (Docker, Inc., 2025b; Docker, Inc., 2025a). A aplicação desses recursos é o que permite que ambas as arquiteturas (CSR e SSR) sejam avaliadas em ambientes padronizados e isolados, eliminando variáveis de infraestrutura. Em síntese, o Docker fornece a base metodológica consistente e reproduzível necessária para validar as conclusões deste estudo, focando a análise puramente nas diferenças de performance entre as estratégias de renderização.

3 Trabalhos Relacionados

Este capítulo apresenta os trabalhos relacionados ao objeto de pesquisa, obtidos por meio de um mapeamento sistemático da literatura. O objetivo desse mapeamento foi identificar, analisar e sintetizar estudos acadêmicos que abordam comparações entre as abordagens de renderização **CSR** (Client-Side Rendering) e **SSR** (Server-Side Rendering) no contexto do desenvolvimento de aplicações web. Buscou-se compreender como essas estratégias impactam aspectos como desempenho, tempo de carregamento, **SEO**, experiência do usuário e escalabilidade. O protocolo adotado, descrito nas seções seguintes, foi elaborado para garantir a abrangência, a precisão e a relevância dos resultados encontrados.

3.1 Questões de pesquisa

Para guiar o mapeamento sistemático da literatura e estruturar a análise dos artigos, foram formuladas as seguintes questões de pesquisa. O objetivo principal é investigar os impactos das abordagens **CSR** e **SSR** na experiência do usuário e em métricas de performance, além de identificar os desafios de implementação e as recomendações propostas na literatura. As questões que nortearam este trabalho são apresentadas a seguir:

- Q1: De que maneira a escolha entre **CSR** e **SSR** influenciam a experiência do usuário?
- Q2: Como as abordagens **CSR** e **SSR** afetam métricas de performance, tempo de carregamento e tempo até a interatividade em aplicações web?
- Q3: Quais são os principais desafios e *trade-offs* na implementação de **CSR** e **SSR**?
- Q4: Quais trabalhos relacionados existem na literatura que abordam recomendações sobre quando usar o **CSR** ou **SSR**?

3.2 Estratégia de busca

Esta seção apresenta a estratégia de buscas de artigos científicos e livros relacionados à pesquisa. As ferramentas utilizadas para realizar as buscas são:

- Periódicos Capes: É uma ferramenta disponibilizada pelo governo federal para uso de estudantes e pesquisadores. Acessando através da instituição de ensino ou pesquisa, é possível ter acesso completo a uma grande quantidade de artigos

científicos publicados em variadas revistas, conferências e universidades. A principal vantagem dessa ferramenta é a possibilidade de ler o conteúdo integral de grande parte das publicações disponíveis. Por outro lado, as expressões de busca atualmente suportadas são bem limitadas.

- *Scopus*: Trata-se de um ferramenta similar ao Periódicos Capes. No entanto, o *Scopus* permite a elaboração de expressões de buscas mais complexas e sofisticadas, servindo para descobrir publicações não detectadas pelas outras plataformas. Além disso, possui um acervo bem mais amplo que o Periódicos Capes. Entretanto, algumas publicações não podem ser vistas na íntegra de forma gratuita.
- *PICOC*: A técnica *PICOC* foi utilizada para estruturar e refinar a estratégia de busca. Essa abordagem consiste em definir cinco elementos principais que auxiliam na formulação da expressão booleana para a pesquisa:
 - P (População/Problema): Define os estudos ou o grupo de interesse, ou seja, o problema ou a população que se deseja investigar. Por exemplo, “artigos que tratem da integração de tecnologias digitais na educação.”
 - I (Intervenção/Interesse): Refere-se à intervenção, prática ou fenômeno que está sendo analisado. Neste caso, pode ser a “inserção de tecnologias digitais nos processos de ensino e aprendizagem.”
 - C (Comparação): Descreve o(s) elemento(s) com os quais a intervenção ou situação é comparada, como “ensino tradicional” ou a comparação entre diferentes estratégias digitais, quando aplicável.
 - O (Outcome/Desfecho): Indica os resultados ou efeitos esperados da intervenção. Por exemplo, “melhora do desempenho acadêmico” ou “maior engajamento dos alunos.”
 - C (Contexto): Considera o ambiente ou cenário onde a intervenção ocorre, como “instituições de ensino, universidades” ou “publicações indexadas em bases internacionais.”

A partir da definição desses elementos, foi possível construir uma expressão booleana que unisse os principais termos de interesse para a pesquisa. Esse método colaborou para refinar os resultados, tornando a busca mais precisa e abrangente, conforme exemplificado no [Quadro 1](#).

3.3 Quadro PICOC

Com base na metodologia PICOC descrita anteriormente, foi possível delimitar de forma estruturada os elementos centrais desta pesquisa. O quadro a seguir detalha a

aplicação de cada componente População (P), Intervenção (I), Comparação (C), Resultado (O) e Contexto (C), definindo o escopo exato para a formulação da expressão de busca e a subsequente análise da literatura.

Quadro 1 – Estrutura PICOC aplicada à pesquisa

Elemento	Descrição
P (População/Problema)	Equipes de desenvolvimento web, arquitetos de software e gestores de TI que precisam escolher estratégias de renderização (CSR ou SSR) para aplicações web modernas, visando otimizar desempenho, SEO e experiência do usuário.
I (Intervenção)	Adoção de técnicas de CSR (Client-Side Rendering): todo (ou quase todo) o conteúdo gerado no lado do cliente, utilizando frameworks/libraries como React, Vue, Angular etc.
C (Comparação)	Implementação de SSR (Server-Side Rendering): conteúdo pré-renderizado no servidor antes de ser enviado ao cliente, usando <i>meta-frameworks</i> como Next.js, Nuxt.js, SvelteKit, Angular Universal, entre outros.
O (Outcome / Resultado)	<ul style="list-style-type: none"> • Métricas de desempenho (tempo de carregamento, <i>time-to-first-byte</i>, <i>largest contentful paint</i>, etc.) • Impacto no SEO (indexabilidade, posicionamento em buscadores) • Experiência do usuário e usabilidade • Escalabilidade do sistema (uso de recursos de servidor/cliente)
C (Contexto)	Aplicações web modernas que buscam equilibrar interatividade, rapidez de carregamento, otimização para motores de busca e redução de custos operacionais. O estudo pode ser aplicado a sistemas de e-commerce, portais de conteúdo, <i>landing pages</i> , etc.

Fonte: os autores

3.3.1 Expressão de busca

Com os elementos do PICOC definidos, foi elaborada uma expressão de busca booleana para ser utilizada nas bases de dados. A *string*, detalhada no quadro a seguir, foi estruturada para combinar os termos centrais da pesquisa as tecnologias de renderização [CSR](#) e [SSR](#) com um conjunto de sinônimos relacionados aos resultados esperados (*Outcome*), como performance, [SEO](#) e experiência do usuário. Essa abordagem permitiu maximizar a recuperação de artigos relevantes, mantendo o foco nos objetivos delineados.

Quadro 2 – Expressão de busca utilizada

Expressão de Busca
$(TITLE-ABS-KEY("Client-Side Rendering" OR "CSR" OR "Server-Side Rendering" OR "SSR")) AND (TITLE-ABS-KEY("web performance" OR "page speed" OR "web optimization" OR "SEO" OR "search engine optimization" OR "user experience" OR "UX" OR "usability"))$

Fonte: os autores

3.4 Estratégia de seleção

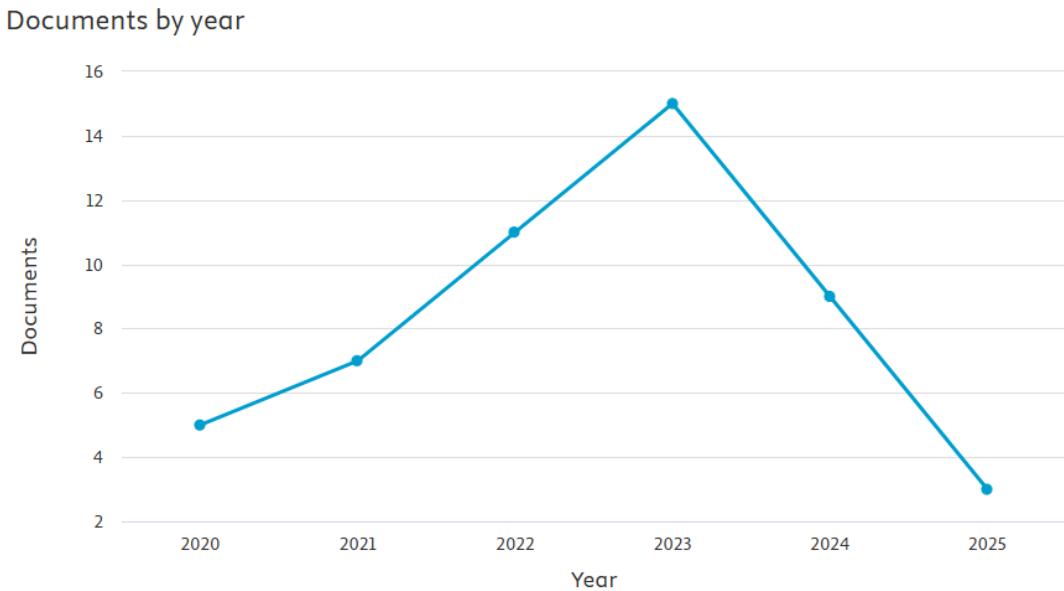
A estratégia de seleção dos artigos foi baseada em critérios de inclusão e exclusão, conforme descrito a seguir:

- Critérios de inclusão:
 - Artigos publicados entre 2020 e 2025.
 - Artigos que abordem o impacto de **CSR** e **SSR** em aplicações web.
 - Artigos que apresentem resultados de experimentos ou estudos de caso relacionados a **CSR** e **SSR**.
- Critérios de exclusão:
 - Artigos que não estejam disponíveis na íntegra.
 - Artigos que não abordem diretamente o tema da pesquisa.
 - Artigos que sejam duplicados ou muito semelhantes a outros já selecionados.

3.5 Caracterização de pesquisa

Na [Figura 11](#), é apresentado o número de publicações identificadas por ano, no intervalo entre 2020 e 2025. Observa-se um crescimento progressivo de 2020 a 2023, culminando em um pico em 2023, com 15 documentos publicados. A partir desse ponto, nota-se uma queda significativa: em 2024, o número de publicações cai para 9, e em 2025 esse número se reduz ainda mais, atingindo apenas 3 documentos. Essa redução pode ser parcialmente atribuída ao fato de que a coleta foi realizada no mês de abril de 2025, o que possivelmente não contempla todas as publicações previstas para o ano. No total, foram encontrados 50 documentos relevantes, distribuídos de forma desigual, evidenciando uma tendência crescente de interesse pelo tema até 2023, seguido de uma possível estabilização ou atraso na indexação dos dados mais recentes.

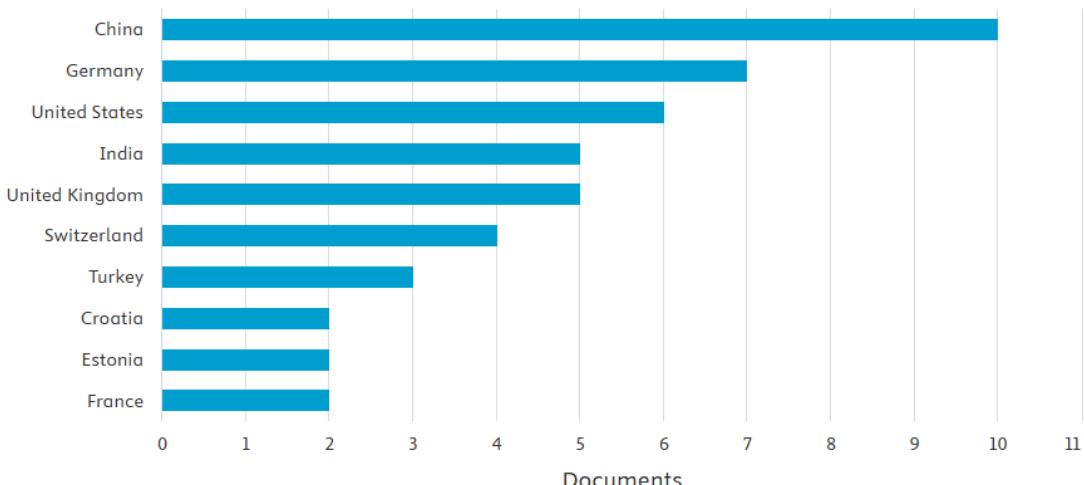
Figura 11 – Número de publicações ao longo dos anos



Fonte: Scopus

Na [Figura 12](#) são apresentados os países com maior número de publicações relacionadas ao tema desta pesquisa. Observa-se que a China lidera com 10 documentos, seguida pela Alemanha (7) e pelos Estados Unidos (6). Em seguida, aparecem Índia, Reino Unido e Suíça, cada um com 5 documentos publicados. A Turquia também se destaca com 4 publicações, enquanto Croácia, Estônia e França completam a lista com 2 documentos cada. Esses dados evidenciam uma concentração relevante de estudos em países com infraestrutura tecnológica consolidada, especialmente na Ásia, Europa Ocidental e América do Norte, o que reforça o caráter global do interesse em torno da comparação entre abordagens de renderização no desenvolvimento de aplicações web.

Figura 12 – Distribuição das publicações por país



Fonte: Scopus

3.6 Artigos selecionados

Após realizadas as leituras preliminares, apenas 13 publicações mostraram-se relevantes para responder às questões de pesquisa e/ou apoiar na elaboração do estudo de caso. Esta seleção considerou critérios de alinhamento temático, profundidade técnica e aplicabilidade ao escopo da pesquisa. A listagem completa dos artigos selecionados pode ser consultada no Quadro 3.

Título	Ano
<i>Progressive Server-Side Rendering with Suspendable Web Templates</i>	2025
<i>Requirements for the Development of a Website Builder with Adaptive Design</i>	2024
<i>Enhancing SEO in Single-Page Web Applications in Contrast With Multi-Page Applications</i>	2024
<i>Web Development Using ReactJS</i>	2023
<i>Improving Universal Rendering Performance on NuxtJS-based Web Application</i>	2023
<i>Comparison between client-side and server-side rendering in the web development</i>	2020
<i>Methods of Improving and Optimizing React Web-applications</i>	2021
<i>Improving ruby on rails-based web application performance</i>	2021
<i>An integrated framework of user experience-oriented smart service requirement analysis for smart product service system development</i>	2022
<i>A Research Framework for B2B Green Marketing Innovation: the Design of Sustainable Websites</i>	2022
<i>Corporate Social Responsibility: Hiring Requisition in Media Companies?</i>	2023
<i>Single page optimization techniques using react</i>	2024
<i>Proceedings of the 19th International Conference on Web Information Systems and Technologies, WEBIST 2023</i>	2023

Quadro 3 – Artigos selecionados sobre estratégias de renderização e desempenho em aplicações web

3.7 Resultados e Discussão

A análise dos 13 artigos selecionados permitiu identificar padrões, lacunas e contribuições relevantes no debate sobre as abordagens de renderização **CSR** e **SSR**. A discussão a seguir foi organizada de modo a responder, de forma integrada, às questões de pesquisa previamente definidas ([seção 3.1](#)).

Em relação à primeira questão de pesquisa (Q1), sobre a influência na experiência do usuário, diversos estudos destacaram o impacto direto da estratégia de renderização na percepção do usuário. O trabalho de ([ZHOU et al., 2022](#)), por exemplo, indica que tempos de resposta mais rápidos geralmente associados à renderização no servidor influenciam positivamente a satisfação. De forma complementar, ([LACOM; SAGOT, 2022](#)) reforça que usuários tendem a abandonar páginas lentas, afetando o engajamento. Em contrapartida, ([POKHRIYAL et al., 2024](#)) e ([KESHARI et al., 2023](#)) apontam que aplicações **CSR**, embora mais interativas após o carregamento, podem apresentar uma latência inicial maior, prejudicando a primeira impressão do usuário.

Essa experiência está intrinsecamente ligada às métricas de performance (Q2). Estudos como os de ([ISKANDAR et al., 2020](#)) e ([ANGKASA et al., 2023](#)) apresentam comparações diretas e mostram que a abordagem **SSR** tende a apresentar desempenho superior em métricas de carregamento inicial, como TTFB e FCP. No entanto, ([PAVIC; BRKIC, 2021](#)) ressalta que otimizações no lado do cliente podem mitigar parte dessas desvantagens do **CSR**. Além disso, trabalhos de ([BEKMANOVA et al., 2024](#)) e ([KLOCHKOV; MULAWKA, 2021](#)) demonstram que a performance final depende mais da arquitetura geral e de boas práticas do que apenas da escolha da renderização, evidenciando a importância de uma visão holística.

A busca por essa performance ideal, contudo, revela os principais desafios e *trade-offs* (Q3). A literatura aponta que o **SSR** pode introduzir maior complexidade ao código e à manutenção ([CARVALHO, 2025](#)), além de aumentar a carga no servidor, o que impacta a escalabilidade e os custos de infraestrutura em cenários de alto tráfego ([ANGKASA et al., 2023](#)). Por outro lado, o **CSR** enfrenta desafios significativos de indexabilidade por mecanismos de busca ([SEO](#)), uma barreira importante para sites orientados a conteúdo ([KOWALCZYK; SZANDALA, 2024](#)).

Diante desses desafios, a quarta questão de pesquisa (Q4) buscou identificar recomendações na literatura sobre quando utilizar cada abordagem. Autores como ([ISKANDAR et al., 2020](#)) sugerem **SSR** para sites com forte dependência de **SEO** (como blogs e e-commerce) e **CSR** para aplicações altamente interativas (como dashboards e sistemas internos). O estudo da WEBIST 2023 ([PROCEEDINGS..., 2023](#)) também aponta para abordagens mistas como uma solução de equilíbrio. Apesar dessas orientações, nota-se

uma carência de diretrizes sistematizadas que auxiliem as equipes de desenvolvimento na escolha da abordagem ideal com base em um conjunto mais amplo de variáveis, como tipo de aplicação, volume de tráfego e metas de negócio. Essa lacuna representa uma oportunidade relevante para pesquisas futuras, incluindo o estudo de caso prático proposto neste trabalho.

4 Metodologia

Este capítulo detalha a abordagem metodológica adotada para a condução do trabalho. A metodologia foi estruturada em quatro etapas principais: o levantamento teórico para fundamentar a pesquisa, o mapeamento sistemático da literatura para identificar o estado da arte, o desenvolvimento de um estudo de caso prático para a coleta de dados empíricos e, por fim, os procedimentos de teste e análise utilizados para comparar as arquiteturas [CSR](#) e [SSR](#).

4.1 Levantamento Teórico

O levantamento teórico consistiu na revisão e sistematização dos principais conceitos, tecnologias e práticas relacionadas à renderização de conteúdo em aplicações web, com foco nas abordagens de [Client-Side Rendering \(CSR\)](#) e [Server-Side Rendering \(SSR\)](#). Essa etapa teve como objetivo fornecer o embasamento necessário para o desenvolvimento do estudo de caso e da análise comparativa proposta neste trabalho.

A fundamentação iniciou-se com a exploração dos princípios do desenvolvimento web moderno, incluindo a arquitetura cliente-servidor, o funcionamento do protocolo [HTTP](#) e as tecnologias essenciais do *frontend*: [HTML](#), [CSS](#) e [JavaScript](#). Estes elementos formam a base para compreender como as estratégias de renderização operam, tanto no lado do cliente quanto no lado do servidor.

Em seguida, foram estudadas em detalhes as abordagens [CSR](#) e [SSR](#). A renderização no lado do cliente ([CSR](#)) foi analisada quanto ao seu funcionamento típico em aplicações [Single Page Application \(SPA\)](#), caracterizadas por uma única página carregada inicialmente, com atualizações dinâmicas de conteúdo via [JavaScript](#). Essa abordagem oferece vantagens como maior interatividade e fluidez na navegação, além de transições rápidas entre páginas internas. No entanto, apresenta desvantagens como maior tempo de carregamento inicial e limitações de indexação por mecanismos de busca.

Por outro lado, a renderização no lado do servidor ([SSR](#)) foi abordada sob a ótica de desempenho inicial otimizado e maior compatibilidade com [SEO](#), pois o conteúdo é entregue já renderizado ao navegador. Essa abordagem é comumente utilizada em aplicações do tipo [Multi Page Application \(MPA\)](#), que possuem múltiplas páginas distintas e se beneficiam da pré-renderização para melhorar a performance inicial, a acessibilidade e a visibilidade em mecanismos de busca. Como contraponto, o [SSR](#) demanda maior processamento no servidor e pode aumentar a complexidade da infraestrutura.

O levantamento teórico foi aprofundado com uma análise dos principais *frameworks* e bibliotecas que materializam as arquiteturas *CSR* e *SSR*. Foi dada ênfase especial ao *React*, como principal expoente da renderização no lado do cliente, e ao *Next.js*, *framework* que o estende para possibilitar uma renderização robusta no lado do servidor. A discussão foi enriquecida com o estudo do ecossistema de outras ferramentas, como *Vue.js* e *Angular*, para consolidar a compreensão sobre os impactos de cada abordagem na experiência do usuário (*UX*), no *SEO*, na acessibilidade e na interatividade.

Essa base teórica consolidada foi, portanto, essencial para embasar as escolhas metodológicas, delimitar o escopo do mapeamento da literatura e, principalmente, para orientar o desenvolvimento e a avaliação do estudo de caso prático detalhado a seguir.

4.2 Mapeamento Sistemático da Literatura

O mapeamento sistemático da literatura teve como objetivo identificar, selecionar e analisar estudos acadêmicos relevantes que abordassem comparações entre as abordagens de renderização *CSR* e *SSR* no contexto do desenvolvimento de aplicações web. Essa etapa foi essencial para compreender o estado da arte, bem como identificar lacunas e oportunidades para a realização do estudo de caso proposto neste trabalho.

A estratégia de busca foi estruturada com o apoio da metodologia *PICOC*, que define os elementos População, Intervenção, Comparação, Resultado e Contexto, com o intuito de guiar a construção das expressões de busca e garantir abrangência e precisão nos resultados. As principais bases de dados utilizadas incluíram Periódicos Capes e Scopus, por oferecerem amplo acervo e suporte a pesquisas refinadas.

Foram utilizadas expressões booleanas combinando termos como *Client-Side Rendering*, *Server-Side Rendering*, *Web Performance*, *SEO*, *UX* e *Frontend Architecture*. Após a aplicação dos critérios de inclusão e exclusão, os artigos resultantes foram classificados e analisados de acordo com sua relevância, tipo de abordagem estudada, metodologias utilizadas e principais conclusões.

A seleção final contemplou trabalhos que abordavam métricas de desempenho, tempo de carregamento, interatividade, *SEO* e experiência do usuário. Além disso, foram considerados estudos que analisavam o uso de frameworks modernos como *React*, *Next.js*, *Nuxt.js* e *Angular Universal*, além de pesquisas aplicadas em contextos reais de produção.

Como resultado, foi possível consolidar uma visão abrangente sobre os desafios, vantagens e limitações de cada abordagem, fornecendo subsídios importantes para a execução do estudo de caso prático apresentado nos capítulos seguintes. O mapeamento sistemático também evidenciou a escassez de estudos nacionais aplicados ao tema, reforçando a

relevância deste trabalho no cenário acadêmico e profissional brasileiro.

4.3 Estudo de Caso Prático

Para materializar a análise comparativa proposta, esta pesquisa se baseia em um estudo de caso prático: o desenvolvimento de uma aplicação web de notícias sobre tecnologia. Este contexto foi escolhido por ser altamente representativo de um vasto segmento de aplicações focadas em conteúdo, onde a velocidade de carregamento inicial e a otimização para mecanismos de busca ([SEO](#)) são fatores críticos para a retenção de usuários e o sucesso do produto.

Com o objetivo de estabelecer condições equivalentes para comparação, foram desenvolvidas duas versões funcionalmente idênticas da plataforma. A primeira utiliza [Client-Side Rendering \(CSR\)](#), implementada com a biblioteca *React (SPA)*, enquanto a segunda adota [Server-Side Rendering \(SSR\)](#), com o *framework Next.js (MPA)*. Ambas as versões consomem dados da mesma *API* externa de notícias e apresentam a mesma interface e funcionalidades como listagem de artigos, busca e visualização de detalhes, assegurando que as diferenças de desempenho observadas possam ser diretamente atribuídas à arquitetura de renderização empregada.

Cada versão foi implementada seguindo os princípios fundamentais da sua respectiva abordagem. Na aplicação [CSR](#), a renderização ocorre predominantemente no navegador do usuário, ao passo que na versão [SSR](#), o conteúdo HTML é pré-renderizado no servidor e enviado pronto ao cliente. Ambas as implementações foram submetidas a um protocolo de testes para a coleta de dados, conforme detalhado na seção seguinte, permitindo uma análise equitativa e baseada em evidências.

4.4 Coleta de Dados e Testes

Esta seção descreve o procedimento de coleta e análise dos dados obtidos a partir da comparação entre as aplicações desenvolvidas com [CSR](#) e [SSR](#). O objetivo é mensurar o desempenho, a eficiência e a experiência do usuário proporcionada por cada aplicação sob condições controladas.

4.4.1 Definição das Métricas

A medição de desempenho em aplicações web modernas evoluiu de uma simples análise do tempo de carregamento total para uma avaliação multifacetada da experiência do usuário (UX). Para padronizar essa medição, o Google introduziu a iniciativa Web Vitals, um conjunto de métricas que quantificam aspectos cruciais da jornada do usuário.

O subconjunto mais importante, conhecido como Core Web Vitals, foca em três pilares da experiência: o carregamento (medido pelo [LCP](#)), a interatividade (medida pelo [INP](#)) e a estabilidade visual (medida pelo [CLS](#)).

Adotando essa metodologia como padrão de mercado, este trabalho selecionou um conjunto de indicadores para realizar uma avaliação completa, abrangendo não apenas a percepção do usuário, mas também os custos computacionais no servidor. Para garantir uma análise estruturada, as métricas foram categorizadas da seguinte forma:

- Web Vitals (Núcleo): Representam a base da análise de experiência do usuário. Foram coletadas as métricas [Time to First Byte \(TTFB\)](#), [First Contentful Paint \(FCP\)](#), [Largest Contentful Paint \(LCP\)](#), [Cumulative Layout Shift \(CLS\)](#) e [Interaction to Next Paint \(INP\)](#). Esses dados foram registrados diretamente no cliente durante a navegação real e persistidos em formato NDJSON para análise posterior.
- Auditoria de Laboratório (Apoio): Utilizando o Lighthouse, foram realizadas auditorias complementares para analisar o [Total Blocking Time \(TBT\)](#) e identificar oportunidades de otimização. As auditorias também validaram aspectos de Acessibilidade e [SEO](#) de forma automatizada.
- Recursos do Servidor (Custo): Para contextualizar o custo de execução de cada abordagem, o consumo de recursos dos contêineres foi monitorado com o comando `docker stats`, registrando o uso de CPU e memória durante os testes.

Adicionalmente, foram observados: número de requisições HTTP e uso de cache (no cliente), bem como consumo de recursos do contêiner (CPU e memória) durante os testes. O [Time to Interactive \(TTI\)](#) foi empregado apenas como métrica de *laboratório* via Lighthouse, não integrando o conjunto atual de Core Web Vitals. O tratamento estatístico detalhado das métricas será apresentado posteriormente.

4.4.2 Ferramentas de Teste

Foram utilizadas ferramentas complementares, com ênfase na coleta contínua no navegador (*field*) e apoio de auditoria em *laboratório*:

- Web Vitals ([Real User Monitoring \(RUM\)](#)): Instrumentação no cliente para coletar as métricas TTFB, FCP, LCP, CLS e INP. O envio dos dados a um endpoint interno é feito via `navigator.sendBeacon`, uma API que garante a transmissão confiável das métricas mesmo quando o usuário encerra a página, com os dados persistidos em formato NDJSON;

- Google Lighthouse (auxiliar): Auditoria em *laboratório* para Performance (incluindo FCP, LCP, CLS, Speed Index e TBT), executada localmente sobre os serviços em Docker;
- Chrome DevTools: Inspeção de rede, *waterfall*, cache e verificação dos *POSTs* de métricas;
- docker stats: Acompanhamento do uso de CPU e memória dos contêineres durante a execução dos testes.

4.4.3 Ambiente de Testes

Os testes para ambas as versões ([CSR](#) e [SSR](#)) foram conduzidos em um ambiente local, utilizando contêineres *Docker* para garantir a consistência. A escolha por essa abordagem foi uma consequência direta das restrições do plano de desenvolvedor (gratuito) da NewsAPI. Especificamente, a política de uso deste plano restringe as chamadas da API a requisições que partem de localhost, o que inviabilizou um teste em produção. No caso da versão [CSR](#), a política de CORS bloquearia as chamadas diretas do navegador em um domínio público; já para a versão [SSR](#), as chamadas de servidor para servidor seriam igualmente bloqueadas, pois o servidor de produção não seria localhost. Portanto, a execução local não apenas contornou essa limitação, mas também assegurou um maior controle sobre as variáveis do experimento e a total reprodutibilidade dos resultados. Ambos os serviços foram executados com paridade de recursos:

- CPU: `-cpus="1.0"` e `-cpuset-cpus="0"`;
- Memória: `-memory="1g"` e `-memory-swap="1g"`;
- Sistema de arquivos: `-read-only` com `-tmpfs /tmp`;
- Persistência de métricas: volume em `/data` com `METRICS_PATH=/data/webvitals.ndjson`.

A aplicação [SSR](#) (Next.js) foi empacotada em modo *standalone*. A aplicação [CSR](#) (React + Vite) foi servida por um processo `Node.js` simples que também expõe o endpoint de métricas. Os serviços públicos locais utilizados nos testes foram:

- SSR/Next.js: `http://localhost:3001`
- CSR/React: `http://localhost:3002`

4.4.4 Execução dos Testes

A execução foi conduzida em quatro etapas, sendo:

1. Aquecimento: duas visitas iniciais à mesma rota em cada aplicação, para estabilização de caches e recursos;
2. Coleta principal (Web Vitals): navegação real em janela anônima, registrando TTFB, FCP, LCP, CLS e INP por meio do endpoint interno e persistindo em arquivo NDJSON;
3. Coleta auxiliar (Lighthouse): auditorias repetidas em modo desktop, com parâmetros de *throttling* consistentes entre cenários, para fornecer referência de *laboratório*;
4. Observabilidade do contêiner: monitoramento pontual com `docker stats` para CPU e memória durante as execuções.

Para reduzir a variabilidade, cada teste foi repetido no mínimo cinco vezes por cenário. A análise detalhada dos dados coletados, incluindo a quantidade total de repetições e o método de agregação estatística (mediana/p50 e p95), é apresentada no **Capítulo de Resultados e Discussão** ([Capítulo 7](#)).

4.4.5 Tratamento e Análise dos Dados

Os dados coletados pela instrumentação ([RUM](#)) foram armazenados em arquivos NDJSON separados por abordagem, contendo os campos de identificação da métrica, valor e carimbo temporal. As auditorias do Lighthouse foram salvas em arquivos JSON/HTML para futura referência. Em seguida, os dados foram consolidados em planilhas, organizados por data, métrica e abordagem ([CSR](#)/[SSR](#)), preparando o conjunto de dados para a análise comparativa.

A análise foi conduzida por meio de estatísticas descritivas, com foco nas medianas p50 e p95, além de médias e desvios padrão quando apropriado. As comparações entre [CSR](#) e [SSR](#) foram realizadas métrica a métrica (TTFB, FCP, LCP, CLS e INP), considerando os *trade-offs* de cada abordagem. Os gráficos e tabelas de síntese resultantes dessa análise são apresentados no capítulo de Resultados e Discussões.

5 Estudo de Caso

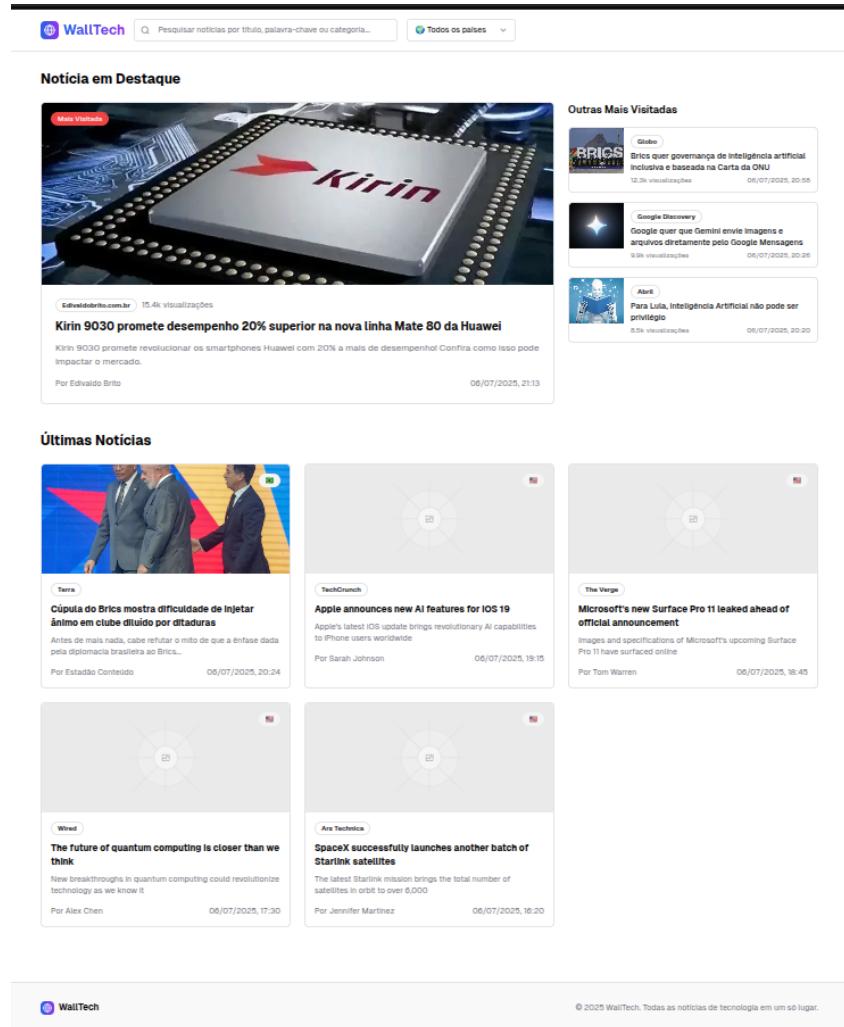
Este capítulo detalha a construção do estudo de caso prático, que se baseia no cenário de uma empresa fictícia e no desenvolvimento de uma plataforma de notícias sobre tecnologia, denominada "WallTech". Serão apresentados o contexto do projeto, os requisitos funcionais, a organização do processo de desenvolvimento, os métodos de design, a implementação das arquiteturas [CSR](#) e [SSR](#), e os testes comparativos realizados.

5.1 Contexto

Este estudo de caso parte do cenário de uma empresa fictícia do setor de notícias sobre tecnologia, cujo objetivo é fornecer conteúdos atualizados e relevantes sobre inovações e tendências do mercado. O público-alvo inclui entusiastas de tecnologia, profissionais da área e usuários que desejam acompanhar as novidades do setor em diferentes dispositivos, graças a uma interface responsiva.

A Figura 13 apresenta o *wireframe* da plataforma WallTech, gerado com o auxílio da ferramenta **V0.dev**, que utiliza inteligência artificial para criar protótipos de interfaces a partir de descrições textuais. Esse modelo visual orienta a organização dos elementos na interface antes da implementação das versões em [CSR](#) e [SSR](#).

Figura 13 – Wireframe da plataforma WallTech

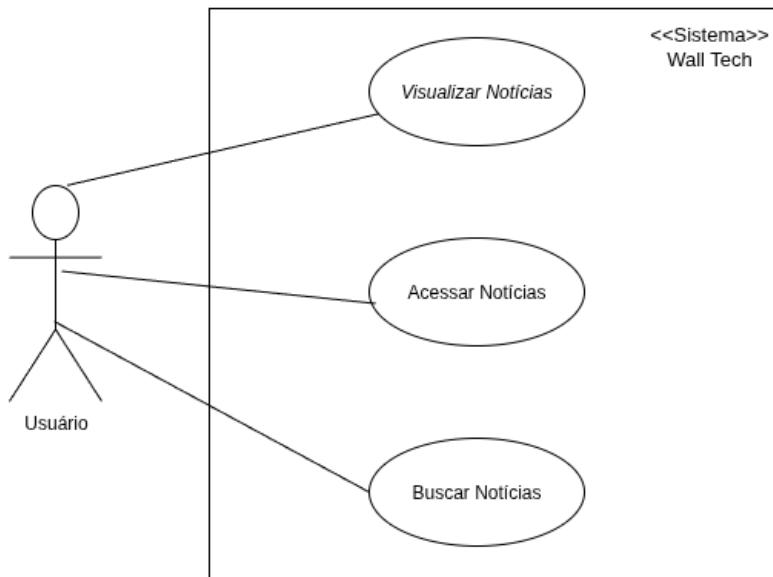


Fonte: os autores.

Para analisar o impacto de diferentes abordagens de renderização web, foram desenvolvidos dois protótipos independentes da plataforma WallTech. O primeiro segue a abordagem de [Client-Side Rendering \(CSR\)](#), estruturado como uma [Single Page Application \(SPA\)](#) utilizando a biblioteca *React*. O segundo utiliza [Server-Side Rendering \(SSR\)](#), implementado como uma [Multi Page Application \(MPA\)](#) com o framework *Next.js*.

A Figura 14 apresenta o diagrama de caso de uso da plataforma, destacando as principais funcionalidades acessíveis a usuários anônimos, como visualizar notícias recentes, acessar detalhes e realizar buscas por palavras-chave.

Figura 14 – Diagrama de caso de uso da plataforma WallTech



Fonte: os autores.

5.2 Processo de Desenvolvimento

O processo de desenvolvimento utilizado para construir a plataforma WallTech segue a metodologia ágil *Kanban*. Essa metodologia de desenvolvimento ágil é baseada em um quadro de tarefas, no qual cada tarefa é representada por um cartão (GOMES, 2014). O quadro Kanban é dividido em colunas que representam o estado atual de cada tarefa. As colunas mais comuns são: *To Do*, *In Progress* e *Done*, e o quadro é atualizado conforme as tarefas são realizadas. Além dessas colunas, o processo foi adaptado para incluir colunas adicionais como *Docs* e *Test*, permitindo que a documentação e os testes fossem gerenciados de forma organizada e eficiente durante o desenvolvimento.

A Figura 15 apresenta um exemplo do quadro Kanban utilizado no *Github Projects*, mostrando a organização das tarefas e o progresso do desenvolvimento da plataforma WallTech. O quadro reflete a estrutura de colunas adaptada, proporcionando uma visão clara do fluxo de trabalho da equipe, o que facilita o acompanhamento das tarefas em diferentes estágios.

Após a definição das funcionalidades principais do sistema, as tarefas foram inicialmente documentadas como *user stories*. As *user stories* são descrições simples e compreensíveis das funcionalidades a serem implementadas, permitindo uma comunicação clara entre a equipe de desenvolvimento e as partes interessadas. Cada *user story* é associada a um conjunto de requisitos específicos e uma definição de pronto, facilitando a compreensão do que precisa ser desenvolvido.

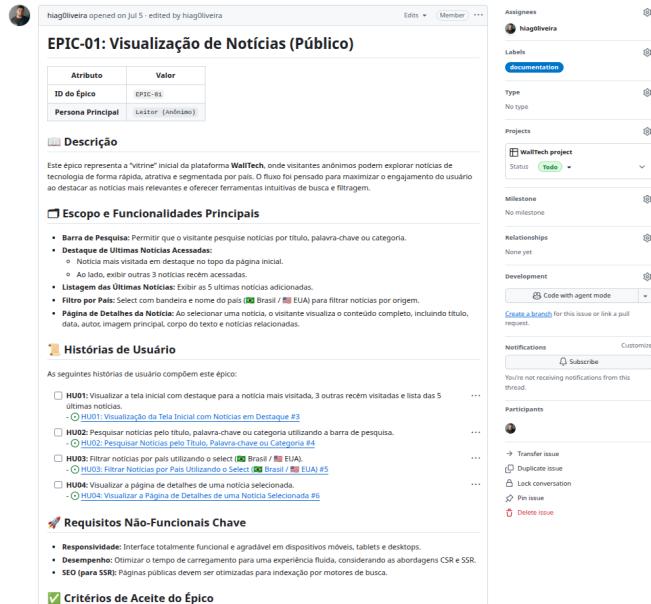
A partir dessas *user stories*, as *issues* foram criadas no *GitHub Projects*. Cada *issue* representa uma tarefa específica que deve ser realizada, baseada nas *user stories*. No *GitHub Projects*, essas *issues* são organizadas nas colunas do quadro Kanban, permitindo que a equipe visualize o progresso de cada tarefa e as mova conforme o andamento do trabalho.

A Figura 16 mostra um exemplo de cartão *issue* no *GitHub Projects*, ilustrando como as *user stories* são transformadas em tarefas e organizadas dentro do quadro Kanban. Cada *issue* possui detalhes sobre a tarefa, como descrições, prioridade e prazo, facilitando o gerenciamento e a execução das atividades no time de desenvolvimento.

Figura 15 – Exemplo de quadro Kanban no *GitHub Projects*

The screenshot shows a Kanban board with five columns: Todo, Docs, In Progress, Done, and Test. Each column has a summary row at the top and individual items listed below. The 'Todo' column has one item: 'React-CSR #2 EPIC-01: Visualização de Notícias (Público)'. The 'Docs' column has two items: 'React-CSR #6 HU04: Visualizar a Página de Detalhes de uma Notícia Selecionada' and 'React-CSR #5 HU03: Filtrar Notícias por País Utilizando o Select (Brasil / EUA)'. The 'In Progress' column has two items: 'React-CSR #3 HU01: Visualização da Tela Inicial com Notícias em Destaque' and 'React-CSR #4 HU02: Pesquisar Notícias pelo Título, Palavra-chave ou Categoria'. The 'Done' column has one item: 'React-CSR #1 EPIC-01: Visualização de Notícias (Público)' with the note 'This has been completed'. The 'Test' column is empty. At the bottom of each column, there is a '+ Add item' button.

Fonte: os autores.

Figura 16 – Exemplo de cartão *issue* no GitHub Projects

Fonte: os autores.

5.2.1 Planejamento Funcional: Épico e Histórias de Usuário

Para guiar o desenvolvimento da plataforma WallTech, foi adotada uma abordagem baseada em práticas ágeis, especialmente no uso de *épicos* e *histórias de usuário*. Essa técnica visa descrever funcionalidades a partir da perspectiva do usuário, proporcionando clareza sobre o propósito e os objetivos de cada componente da aplicação.

As histórias de usuário foram redigidas segundo o modelo dos 3Ws (Who, What, Why), uma técnica comum em análise de requisitos que busca responder:

- Who (Quem): Quem está solicitando ou interagindo com a funcionalidade?
- What (O quê): Qual é a ação ou funcionalidade desejada?
- Why (Por quê): Qual é o benefício ou valor esperado para o usuário?

Esse modelo torna a narrativa mais centrada no usuário e contribui para um entendimento compartilhado entre desenvolvedores, designers e partes interessadas. Além disso, as funcionalidades foram agrupadas em épicos, que representam blocos de funcionalidades coesas e de maior escala dentro do sistema.

A seguir, apresenta-se o EPIC-01, responsável por organizar as funcionalidades relacionadas à visualização pública de notícias na plataforma WallTech, seguido de suas respectivas histórias de usuário.

EPIC-01: Visualização de Notícias (Público)

- ID do Épico: EPIC-01
- Persona Principal: Leitor (Anônimo)

Descrição: Este épico representa a “vitrine” da plataforma WallTech, onde visitantes anônimos podem explorar notícias de tecnologia de forma rápida, atrativa e segmentada por país. O fluxo foi pensado para maximizar o engajamento do usuário ao destacar as notícias mais relevantes e oferecer ferramentas intuitivas de busca e filtragem.

Funcionalidades Principais:

- Barra de pesquisa (por título, palavra-chave ou categoria);
- Destaque para a notícia mais visitada e outras três recentemente acessadas;
- Listagem das cinco últimas notícias;
- Filtro por país (Brasil ou EUA);
- Página de detalhes da notícia com conteúdo completo e relacionadas.

Requisitos Não-Funcionais:

- Interface responsiva em diferentes dispositivos;
- Otimização de desempenho para carregamento rápido;
- Suporte a SEO nas páginas públicas (relevante em SSR).

Critérios de Aceite:

- Visitantes conseguem navegar e visualizar notícias sem necessidade de login;
- É possível realizar pesquisas e aplicar filtros por país;
- A navegação é responsiva, intuitiva e eficiente;
- A página de detalhes apresenta informações completas da notícia.

HU01: Visualização da Tela Inicial com Notícias em Destaque

- Who: Visitante anônimo
- What: Ver na página inicial uma barra de pesquisa, uma notícia mais visitada em destaque, três outras recém acessadas e uma lista com as cinco últimas notícias publicadas.
- Why: Explorar rapidamente o conteúdo mais relevante e recente sobre tecnologia.

História: *Como um visitante, quero ver na página inicial uma barra de pesquisa, a notícia mais visitada em destaque, outras três recém visitadas ao lado e as cinco últimas notícias abaixo, para que eu possa explorar rapidamente o conteúdo mais relevante sobre tecnologia.*

Critérios de Aceite:

- A página inicial contém uma barra de pesquisa funcional;
- A notícia mais visitada aparece em destaque;
- Outras três notícias aparecem ao lado da principal;
- Abaixo, as cinco últimas notícias são exibidas em ordem cronológica;
- Interface é responsiva e com bom desempenho.

HU02: Pesquisar Notícias por Palavra-chave ou Categoria

- Who: Visitante
- What: Utilizar a barra de pesquisa para buscar por título, palavra-chave ou categoria.
- Why: Encontrar conteúdos relevantes de forma eficiente.

História: *Como um visitante, quero pesquisar notícias por título, palavra-chave ou categoria, para que eu possa encontrar facilmente conteúdos relevantes sem precisar navegar por toda a lista.*

Critérios de Aceite:

- Campo de texto para busca;
- Filtro por categoria;
- Resultados relevantes são exibidos com base na busca;

- A busca pode ser combinada com o filtro por categoria;
- Mensagem amigável exibida em caso de nenhum resultado.

HU03: Filtrar Notícias por País

- Who: Visitante
- What: Filtrar as notícias por país usando um seletor com bandeiras.
- Why: Visualizar conteúdos específicos da localidade desejada.

História: *Como um visitante, quero filtrar notícias por país (Brasil ou EUA), para que eu possa visualizar apenas conteúdos relevantes para minha localidade ou interesse regional.*

Critérios de Aceite:

- Menu com seleção de país exibindo bandeira e nome;
- Lista de notícias é atualizada dinamicamente conforme a seleção;
- O filtro mantém o estado ao navegar;
- Filtro compatível com busca por palavra-chave ou categoria.

HU04: Visualizar Detalhes de uma Notícia

- Who: Visitante
- What: Acessar a página de detalhes de uma notícia específica.
- Why: Ler o conteúdo completo e visualizar informações adicionais.

História: *Como um visitante, quero clicar em uma notícia para acessar sua página de detalhes, para que eu possa ler o conteúdo completo e visualizar informações relacionadas.*

Critérios de Aceite:

- Acesso a uma URL única da notícia;
- Exibição completa do conteúdo (título, autor, data, imagem, corpo do texto, país de origem);
- Exibição de até três notícias relacionadas;

- Página de erro amigável no caso de URL inválida;
- Botão para retornar à lista ou pesquisa anterior.

5.3 Requisitos

O levantamento e a definição dos requisitos da plataforma WallTech foram guiados pelas necessidades dos usuários e organizados por meio da técnica de *histórias de usuário*, agrupadas em *épicos*. A partir da análise funcional do EPIC-01 — Visualização de Notícias (Público), foram identificadas as principais funcionalidades esperadas para a aplicação, considerando tanto o fluxo de interação do visitante quanto os objetivos de usabilidade, desempenho e escalabilidade.

A seguir, são apresentados os requisitos funcionais, que representam as funcionalidades diretamente percebidas pelos usuários, e os não funcionais, que tratam de atributos como desempenho, acessibilidade e responsividade.

Requisitos Funcionais:

- O sistema deve permitir que visitantes visualizem uma lista de notícias recentes e em destaque.
- O sistema deve disponibilizar uma barra de pesquisa por palavra-chave, título ou categoria.
- O sistema deve permitir a filtragem de notícias por país (Brasil ou EUA).
- O sistema deve possibilitar o acesso aos detalhes completos de uma notícia selecionada.
- O sistema deve armazenar localmente os acessos recentes para melhorar a experiência do usuário.
- O sistema deve apresentar mensagens claras em situações de ausência de conteúdo.

Requisitos Não Funcionais:

- A interface deve ser responsiva, adaptando-se corretamente a diferentes tamanhos de tela.
- O carregamento das páginas deve ser otimizado, proporcionando uma experiência fluida mesmo em conexões lentas.

- O sistema deve estar preparado para indexação por motores de busca (SEO), quando aplicado via Server-Side Rendering.
- A navegação deve ser acessível, com suporte a teclado e leitores de tela.
- A interface deve manter a consistência visual e funcional entre suas versões SPA e MPA.

5.4 Design do Sistema

O design do sistema foi orientado para refletir as diferenças estruturais entre as abordagens **SPA** e **MPA**, levando em consideração os requisitos funcionais da plataforma *WallTech*. Esta vitrine digital exibe notícias de tecnologia obtidas por meio da *NewsAPI*, com recursos de busca, filtragem e destaque de conteúdo segmentado por país. Não há backend próprio, sendo todas as chamadas feitas diretamente para a API externa, o que simplifica a arquitetura e acentua o papel do frontend na renderização de conteúdo.

Para modelagem arquitetural e comportamental, foram utilizados diagramas da UML, incluindo:

- Diagrama de Caso de Uso, para representar as principais funcionalidades acessadas pelos usuários visitantes.
- Diagrama de Sequência, a fim de ilustrar o fluxo de interação entre navegador e a *NewsAPI* durante operações como busca e carregamento de notícias.
- Diagrama de Componentes, para representar os módulos da aplicação, como a interface, o serviço de requisição à API, e os componentes de renderização.

As decisões de design foram fundamentadas em boas práticas para renderização web discutidas por ([OSMANI; MILLER, 2025](#)), bem como nas diretrizes da literatura especializada em arquitetura de frontend, como apresentado pela ([EMADAMERHO-ATORI, 2024](#)).

Segundo ([OSMANI; MILLER, 2025](#)), a escolha entre renderização no cliente ou no servidor deve considerar o contexto da aplicação, os requisitos de desempenho e os objetivos de SEO. Já o artigo da ([EMADAMERHO-ATORI, 2024](#)) destaca que SPAs tendem a oferecer maior fluidez e interatividade, enquanto MPAs são mais eficazes em aplicações que dependem de indexação e acessibilidade.

5.5 Abordagens de Renderização e Navegação

Para o estudo comparativo, a plataforma foi desenvolvida em duas arquiteturas distintas de renderização e navegação: Server-Side Rendering (SSR) com Multi Page Application (MPA) e Client-Side Rendering (CSR) com Single Page Application (SPA). Cada abordagem foi implementada com tecnologias adequadas ao seu paradigma Next.js para SSR/MPA e React para CSR/SPA permitindo observar diferenças de desempenho, interatividade e otimização para mecanismos de busca (SEO).

A seguir, cada abordagem é apresentada com seu fluxo típico de funcionamento, diagrama de componentes (representando a estrutura modular) e diagrama de sequência (detalhando as interações passo a passo).

5.5.1 SSR/MPA - Fluxo e Arquitetura

Na abordagem Server-Side Rendering (SSR) com Multi Page Application (MPA), cada página é renderizada no servidor a partir de uma requisição HTTP completa. O HTML final já é entregue com os dados integrados, permitindo que o navegador exiba o conteúdo imediatamente, favorecendo o tempo de carregamento inicial e a indexação por *crawlers* ([EMADAMERHO-ATORI, 2024](#)).

- O usuário acessa o site;
- O navegador envia uma requisição GET ao servidor Next.;
- O servidor obtém os dados mais recentes na NewsAPI;
- O servidor monta a página HTML já com os dados;
- O HTML é enviado ao navegador e exibido;
- Cada navegação subsequente gera nova requisição completa ao servidor;
- Scripts no cliente utilizam LocalStorage para melhorar a experiência.

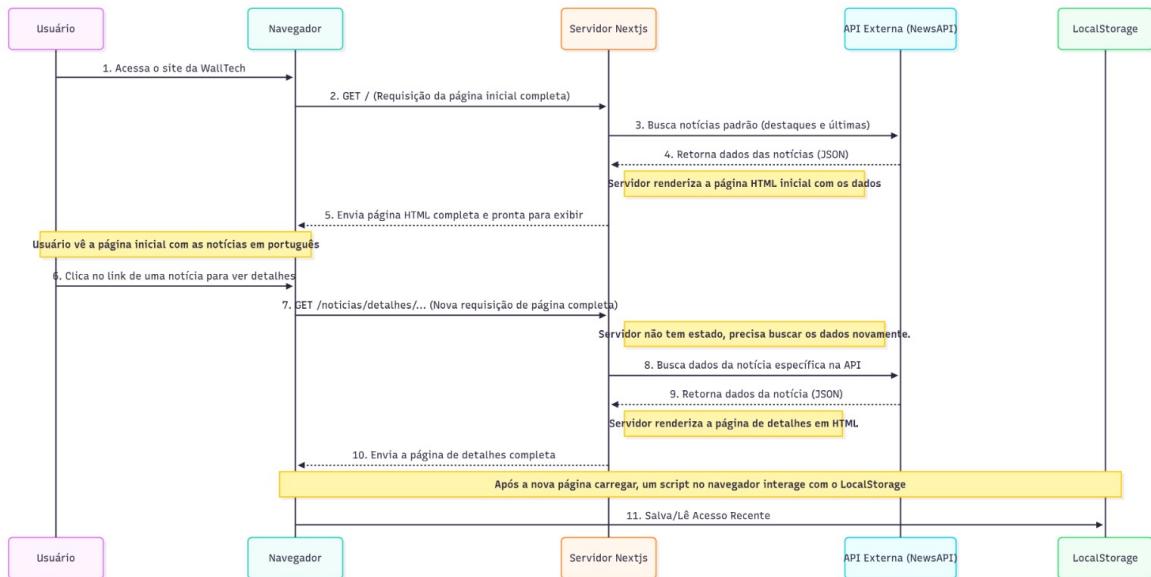
A Figura 18 ilustra como, nesta arquitetura, o navegador interage diretamente com o Servidor Next.js para cada página requisitada. O servidor, por sua vez:

1. Identifica a rota usando o roteamento baseado em arquivos;
2. Executa a lógica de busca de dados (`getServerSideProps`);
3. Faz chamadas à NewsAPI para obter o conteúdo;

4. Renderiza o HTML e o envia ao cliente.

A Figura 17 detalha a ordem das interações. O usuário inicia a navegação, o servidor processa a requisição, consulta a API externa, renderiza o HTML e envia a resposta pronta para o cliente. O uso de *hydration* ativa a interatividade após o carregamento inicial.

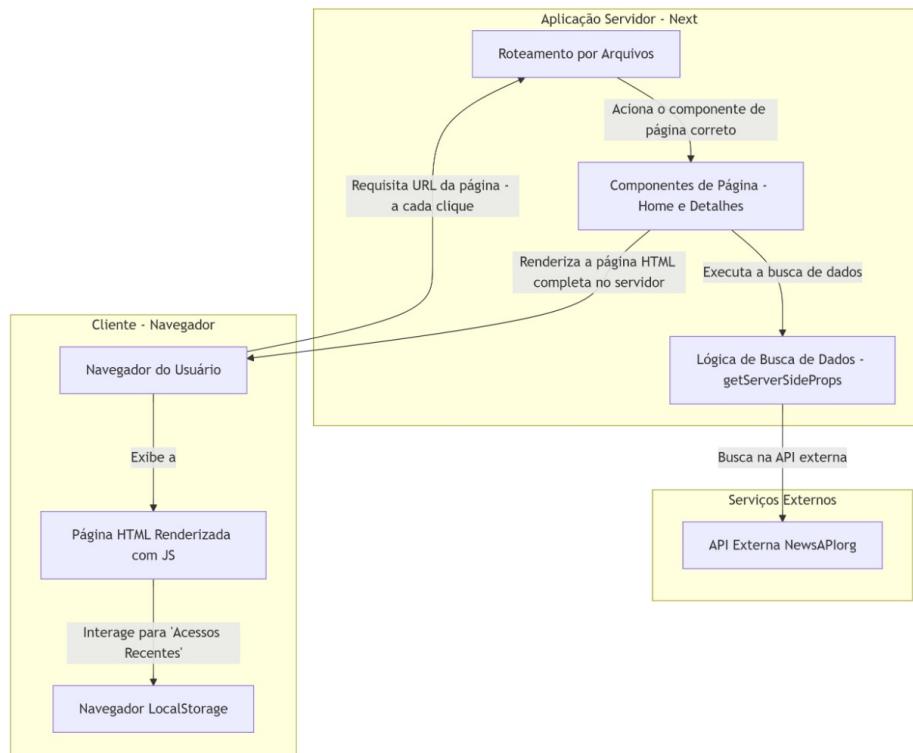
Figura 17 – Diagrama de sequência - SSR/MPA



Fonte: os autores.

A Figura 18 mostra que, nessa arquitetura, o servidor Next.js centraliza o roteamento, a obtenção de dados e a renderização das páginas, entregando ao navegador HTML já pré-renderizado. Após o carregamento, scripts ativam a interatividade e permitem o uso do LocalStorage, uma interface da Web Storage API que possibilita armazenar dados no formato chave-valor de forma persistente no navegador do usuário, ideal para guardar informações como o histórico de notícias acessadas.

Figura 18 – Diagrama de Componentes - MPA (SSR em Next.js)



Fonte: os autores.

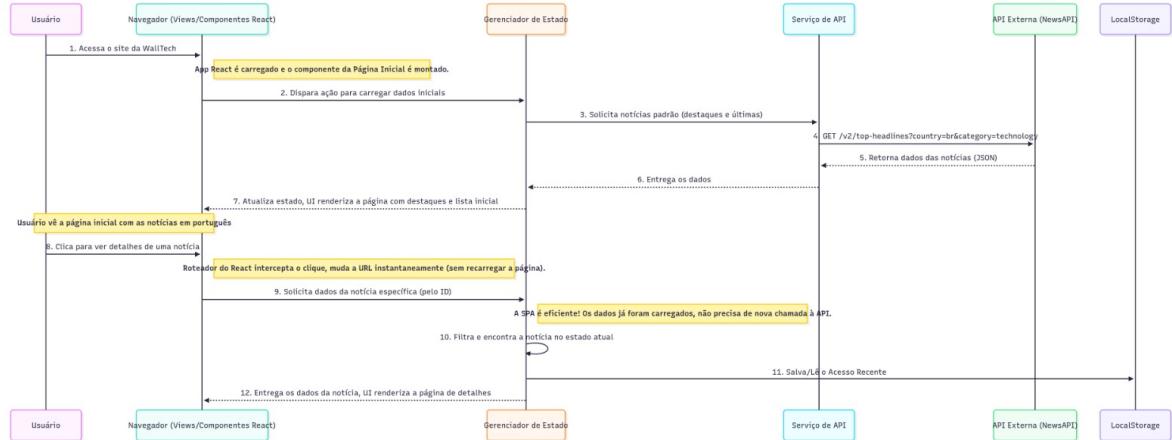
5.5.2 CSR/SPA - Fluxo e Arquitetura

Na abordagem Client-Side Rendering (CSR) com Single Page Application (SPA), o carregamento inicial envia um HTML mínimo e um pacote JavaScript que contém toda a lógica da aplicação. A partir daí, a navegação e renderização são executadas no navegador, sem recarregar a página.

- O usuário acessa o site;
- O navegador baixa o bundle React e monta a interface inicial;
- O gerenciador de estado solicita dados à NewsAPI;
- A interface é atualizada dinamicamente com os dados recebidos;
- Ao navegar, o roteador do React altera a URL e renderiza novos componentes;
- Dados podem ser lidos ou salvos no LocalStorage.

A Figura 19 descreve como o navegador processa as interações: primeiro carrega a aplicação, depois solicita dados conforme necessário e atualiza a interface sem recarregar a página, garantindo uma navegação contínua.

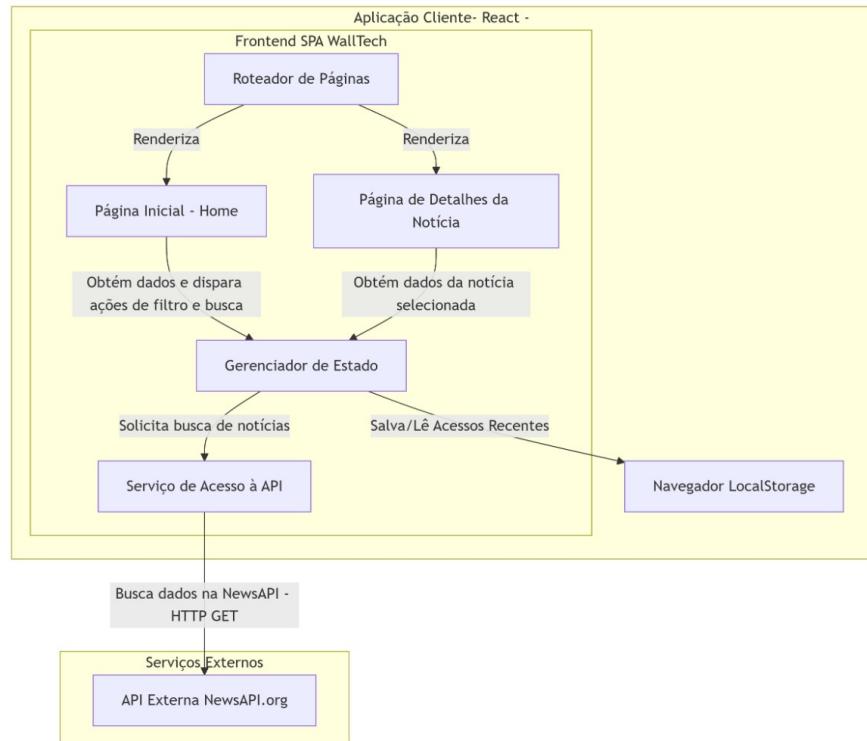
Figura 19 – Diagrama de sequência - CSR/SPA



Fonte: os autores.

A Figura 20 mostra que, nessa arquitetura, o navegador concentra toda a lógica de renderização. O Roteador React decide quais componentes de página serão exibidos, enquanto o Gerenciador de Estado controla o fluxo de dados entre a interface e a NewsAPI.

Figura 20 – Diagrama de Componentes - SPA (React)



Fonte: os autores.

5.6 Implementação

Esta seção descreve o conjunto de tecnologias, bibliotecas e ferramentas que são empregadas para a construção das duas versões do sistema de prova de conceito, detalhando a fundamentação para a escolha de cada componente do ecossistema de desenvolvimento. O gerenciamento do código-fonte e do ciclo de vida do projeto é realizado com o sistema de controle de versão Git e a plataforma de hospedagem GitHub, conforme as práticas descritas na Seção 2.7.

Ambas as implementações consomem dados da mesma fonte externa, a News API, uma RESTful que fornece o conteúdo jornalístico para a aplicação, como detalhado na Seção 2.8. Para garantir a consistência visual e a qualidade da interface entre as duas arquiteturas, utiliza-se a biblioteca de componentes shadcn/ui, que oferece um conjunto de componentes acessíveis e personalizáveis, conforme apresentado na Seção 2.6.

5.6.1 Implementação da Aplicação SPA

A implementação da [Single Page Application \(SPA\)](#) foi desenvolvida utilizando a biblioteca React na sua versão 18. O React é uma biblioteca JavaScript declarativa, mantida pela Meta, focada na construção de interfaces de usuário a partir de componentes reutilizáveis. Sua adoção neste projeto se dá por sua vasta popularidade no mercado e ao seu paradigma de componentização, que facilita a criação de UIs modulares e de fácil manutenção ([REACT, 2025](#)). A eficiência da renderização é otimizada pelo uso de um DOM Virtual, um conceito central da biblioteca que minimiza as manipulações diretas no navegador.

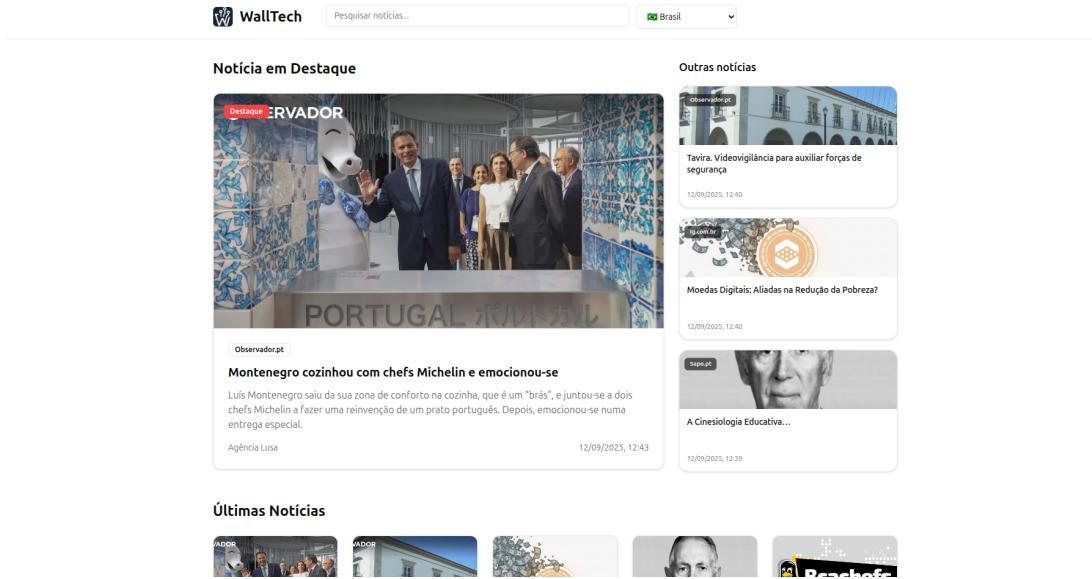
Para a estruturação inicial do projeto e o gerenciamento do ambiente de desenvolvimento, utiliza-se a ferramenta de *build* Vite. O Vite é um ecossistema de desenvolvimento frontend moderno que oferece um servidor de desenvolvimento com recarregamento rápido (*Hot Module Replacement*) e um processo de compilação (*build*) otimizado, que resulta em pacotes de produção menores e mais eficientes ([Evan You and Vite contributors, 2025](#)).

O roteamento no lado do cliente, uma característica fundamental da arquitetura SPA, implementa-se com a biblioteca React Router. Trata-se da solução padrão para navegação em aplicações React, que possibilita a criação de uma experiência de usuário fluida e sem recarregamentos de página ao manipular a barra de Histórico do navegador ([Remix, 2025](#)). A comunicação com a News API é realizada por meio da `fetch`, nativa dos navegadores modernos.

A [Figura 21](#) mostra a página inicial da plataforma WallTech na implementação SPA com [CSR](#), em conformidade com o wireframe ([Figura 13](#)). Nessa página, o usuário visualiza a notícia mais visitada em destaque, três outras recentemente acessadas ao lado,

uma lista com as cinco últimas notícias publicadas, além de barra de pesquisa e filtro por país (Brasil ou EUA).

Figura 21 – Aplicação SPA com React (CSR)



Fonte: os autores.

5.6.2 Implementação da Aplicação MPA

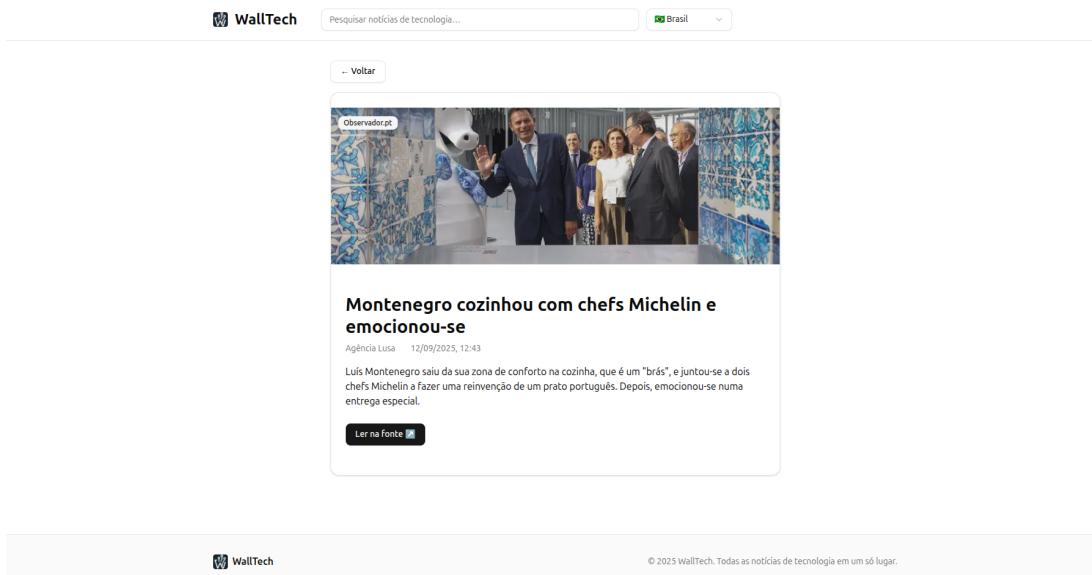
A implementação da [Multi Page Application \(MPA\)](#), com foco em [Server-Side Rendering \(SSR\)](#), é desenvolvida com o *framework* Next.js na versão 14. O Next.js é um meta-framework baseado em React, mantido pela Vercel, que se posiciona como uma solução completa para a construção de aplicações web de produção. Sua escolha para este estudo de caso justifica-se por ser a principal referência de mercado para a implementação de [SSR](#) no ecossistema React, oferecendo uma estrutura robusta e opinativa ([NEXT, 2024](#)).

O *framework* opera sobre um ambiente Node.js, o que permite a execução de código JavaScript no lado do servidor ([NODE, 2025](#)). Essa capacidade é a base da renderização no servidor, onde o Next.js utiliza funções específicas, como a `getServerSideProps`, para buscar dados de fontes externas e pré-renderizar o HTML completo de uma página antes de enviá-la ao navegador. Além disso, o Next.js implementa um sistema de roteamento baseado no sistema de arquivos, onde a estrutura de diretórios da pasta `pages` define automaticamente as rotas da aplicação, simplificando a configuração e a manutenção do projeto.

A [Figura 22](#) mostra a página de detalhes de uma notícia na plataforma WallTech, na implementação [MPA](#) com [SSR](#). Ao clicar em uma notícia na listagem, o usuário é direcionado para esta tela, onde pode visualizar imagem, título, metadados (agência e data/hora) e resumo. Também é possível acessar a matéria original pelo botão “Ler na

fonte” e retornar à página anterior pelo botão “Voltar”.

Figura 22 – Aplicação MPA com Next.js ([SSR](#))



Fonte: os autores.

Com as duas versões da plataforma WallTech implementadas e funcionais, o alicerce para a análise comparativa foi estabelecido. O capítulo seguinte detalha o protocolo experimental empregado para a execução dos testes, a configuração do ambiente controlado e os procedimentos de coleta de dados, garantindo que a comparação entre as abordagens [CSR](#) e [SSR](#) seja precisa e reproduzível.

6 Coleta de Dados

Com as duas versões da plataforma WallTech desenvolvidas, este capítulo descreve o processo prático de coleta de dados que fundamenta a análise de desempenho comparativa. O objetivo é detalhar o ambiente de testes controlado, as ferramentas de instrumentação, o empacotamento das aplicações e os protocolos de execução utilizados. A padronização e o detalhamento dessas etapas são fundamentais para assegurar a validade, a confiabilidade e a reproduzibilidade dos dados que serão apresentados no capítulo de Resultados.

6.1 Configuração do Ambiente Experimental

Esta seção detalha como as duas versões da plataforma WallTech uma [MPA](#) com [SSR](#) (Next.js) e uma [SPA](#) com [CSR](#) (React+Vite) foram instrumentadas para coleta de *Web Vitals*, empacotadas em contêineres *Docker* com paridade de recursos e executadas em ambiente local controlado para assegurar reproduzibilidade do estudo.

6.1.1 Limitações de Execução e Decisão Metodológica

Durante o desenvolvimento, a *NewsAPI* impôs restrições de uso em produção (por exemplo, políticas de CORS e/ou limitações do plano vigentes no período do estudo), o que impactaria a coleta de dados em ambiente hospedado. Para eliminar interferências externas e garantir controle experimental, decidiu-se executar ambos os protótipos localmente em contêineres *Docker*, com os mesmos limites de CPU e memória, sistema de arquivos *read-only* e partições temporárias (*tmpfs*) para reduzir variações de I/O. Assim, a comparação [CSR](#) vs. [SSR](#) foca no *modelo de renderização* e não em diferenças de *hosting*.

6.1.2 Instrumentação de Web Vitals

Em ambos os protótipos, a instrumentação é realizada no cliente, e os dados são enviados para um endpoint interno exposto pelo próprio contêiner. Utiliza-se `navigator.sendBeacon` como via principal (por não bloquear a navegação) e `fetch` como *fallback* com `keepalive`. As métricas coletadas são os *Core Web Vitals* atuais: [TTFB](#), [FCP](#), [LCP](#), [CLS](#) e [INP](#).

Os registros são persistidos em NDJSON (uma linha por métrica), contendo, dentre outros, os campos: `{id, name, value, rating, navigationType, attribution, entries, ts}`. A análise estatística (p.ex., p50/p95) está *deliberadamente breve* aqui e será desenvolvida em capítulos posteriores.

Na versão SSR (App Router), um *Client Component* global usa `useReportWebVitals` para escutar e enviar as métricas ao endpoint `/api/vitals`. No servidor, a rota grava o NDJSON em um caminho configurável: por padrão `/tmp/webvitals.ndjson` (partição `tmpfs`), ou no caminho definido por `$METRICS_PATH` quando se deseja persistência via volume.

Código 6.1 – Endpoint de métricas no SSR/Next.js (visão de servidor)

```

1 // src/app/api/vitals/route.ts
2 import { NextRequest, NextResponse } from "next/server";
3 import { appendFile } from "node:fs/promises";
4
5 export const runtime = "nodejs";
6 export const dynamic = "force-dynamic";
7
8 const filePath = process.env.METRICS_PATH ?? "/tmp/webvitals.ndjson";
9
10 export async function POST(req: NextRequest) {
11   const metric = await req.json();
12   const doc = { ...metric, ts: new Date().toISOString() };
13   await appendFile(filePath, JSON.stringify(doc) + "\n", "utf8");
14   return NextResponse.json({ ok: true });
15 }
```

Código 6.2 – Envio de métricas no cliente (SSR/Next.js)

```

1 "use client";
2 import { useReportWebVitals } from "next/web-vitals";
3 import { useCallback } from "react";
4
5 export default function WebVitals() {
6   const report = useCallback((metric: any) => {
7     const body = JSON.stringify({
8       id: metric.id, name: metric.name, value: metric.value,
9       rating: metric.rating, navigationType: metric.navigationType,
10      attribution: metric.attribution, entries: metric.entries, ts: Date.now(),
11    });
12
13    if (navigator.sendBeacon) {
14      navigator.sendBeacon("/api/vitals",
15        new Blob([body], { type: "application/json" }));
16    } else {
17      fetch("/api/vitals", {
18        method: "POST", body, keepalive: true,
19        headers: { "Content-Type": "application/json" },
20      });
21    }
22  }, []);
23
24  useReportWebVitals(report);
25  return null;
26 }
```

Na versão [CSR](#), a coleta usa `web-vitals/attribution` (para enriquecer *attribution* em [LCP/CLS/INP](#)). O envio é feito para `/api/web-vitals`. Como não há servidor *framework* (p. ex., Next), a aplicação é servida por um processo Node.js simples (`server.cjs`) que entrega o `dist/` e expõe o endpoint de métricas, gravando no mesmo esquema (`/tmp` ou `$METRICS_PATH`).

Código 6.3 – Envio de métricas no cliente (CSR/React+Vite)

```

1 import { useEffect } from "react";
2 import { onCLS, onINP, onLCP, onFCP, onTTFB, type MetricWithAttribution } from "
   web-vitals/attribution";
3
4 export default function WebVitals() {
5   useEffect(() => {
6     const report = (m: MetricWithAttribution) => {
7       const body = JSON.stringify({
8         id: m.id, name: m.name, value: m.value, rating: m.rating,
9         navigationType: m.navigationType, attribution: m.attribution,
10        entries: m.entries, ts: Date.now(),
11      });
12
13      if (navigator.sendBeacon) {
14        navigator.sendBeacon("/api/web-vitals",
15          new Blob([body], { type: "application/json" }));
16      } else {
17        fetch("/api/web-vitals", {
18          method: "POST", body, keepalive: true,
19          headers: { "Content-Type": "application/json" },
20        });
21      }
22    };
23
24    onCLS(report); onINP(report); onLCP(report); onFCP(report); onTTFB(report);
25  }, []);
26
27  return null;
28}
```

Código 6.4 – Servidor estático + endpoint (CSR/React+Vite)

```

1 // server.cjs - Node HTTP simples (serve dist/ e exp e /api/web-vitals)
2 const http = require("node:http");
3 const { appendFile, readFile, stat } = require("node:fs/promises");
4 const { createReadStream } = require("node:fs");
5 const path = require("node:path");
6
7 const PORT = process.env.PORT || 3000;
8 const DIST = path.join(process.cwd(), "dist");
9 const METRICS_PATH = process.env.METRICS_PATH || "/tmp/webvitals.ndjson";
```

```

10
11 const server = http.createServer(async (req, res) => {
12   const url = new URL(req.url, `http://${req.headers.host}`);
13
14   if (url.pathname === '/api/web-vitals') {
15     if (req.method === 'POST') {
16       const chunks = []; req.on("data", c => chunks.push(c));
17       req.on("end", async () => {
18         const json = JSON.parse(Buffer.concat(chunks).toString("utf8") || "{}");
19         await appendFile(METRICS_PATH, JSON.stringify({ ...json, ts: Date.now()
20           }) + "\n", "utf8");
21         res.writeHead(200, { "Content-Type": "application/json" });
22         res.end(JSON.stringify({ ok: true }));
23       });
24     }
25     if (req.method === 'GET') {
26       const data = await readFile(METRICS_PATH, "utf8").catch(() => "");
27       res.writeHead(200, { "Content-Type": "application/x-ndjson" });
28       res.end(data); return;
29     }
30     res.writeHead(405).end(); return;
31   }
32
33 // estático + fallback SPA
34 let p = decodeURIComponent(url.pathname);
35 if (p === '/') p = '/index.html';
36 const file = path.join(DIST, p);
37 try {
38   const s = await stat(file); if (s.isDirectory()) throw 0;
39   createReadStream(file).pipe(res);
40 } catch {
41   createReadStream(path.join(DIST, "index.html")).pipe(res);
42 }
43 });
44 server.listen(PORT);

```

6.1.3 Empacotamento Docker: SSR/MPA (Next.js) e CSR/SPA (React+Vite)

Para a aplicação com **SSR**, adotou-se o modo standalone do Next.js e *multi-stage build*: (i) instalação de dependências, (ii) *build* e (iii) *runner* minimalista. O contêiner expõe a porta 3000, roda como usuário `node` e lê variáveis em tempo de execução (importante para segredos e chaves).

Código 6.5 – Dockerfile da aplicação SSR/MPA (Next.js)

```

1 # ----- 1) deps -----
2 FROM node:20-alpine AS deps
3 WORKDIR /app
4 COPY package*.json ./

```

```

5 RUN npm ci --ignore-scripts
6
7 FROM node:20-alpine AS builder
8 WORKDIR /app
9 ENV NEXT_TELEMETRY_DISABLED=1
10 COPY --from=deps /app/node_modules ./node_modules
11 COPY . .
12 RUN npm run build
13
14 FROM node:20-alpine AS runner
15 WORKDIR /app
16 ENV NODE_ENV=production
17 ENV NEXT_TELEMETRY_DISABLED=1
18
19 COPY --from=builder /app/.next/standalone ./
20 COPY --from=builder /app/public ./public
21 COPY --from=builder /app/.next/static ./next/static
22
23 USER node
24 EXPOSE 3000
25 CMD ["node", "server.js"]

```

Código 6.6 – Build e execução do container SSR com limites e volume de métricas

```

1 docker build --no-cache -t next-ssr:prod .
2
3 docker run --name next-ssr \
4   -p 3001:3000 \
5   -e METRICS_PATH=/data/webvitals.ndjson \
6   -v "$PWD/metrics:/data:rw" \
7   --cpus="1.00" --cpuset-cpus="0" \
8   --memory="1g" --memory-swap="1g" \
9   --pids-limit=256 \
10  --read-only \
11  --tmpfs /tmp \
12  --tmpfs /app/.next/cache \
13  next-ssr:prod

```

Já para a aplicação com [CSR](#), o *build* é produzido pelo Vite e servido por `server.cjs`. Diferentemente do [SSR](#), as variáveis `VITE_*` são resolvidas em tempo de build; por isso, a `.env` é copiada para o estágio *builder* (ou alternativamente injetada com `-build-arg`). Em execução, o caminho do arquivo de métricas é definido por `$METRICS_PATH`.

Código 6.7 – Dockerfile da aplicação CSR/SPA (React+Vite)

```

1 # ----- 1) deps -----
2 FROM node:20-alpine AS deps
3 WORKDIR /app
4 COPY package*.json ./
5 RUN npm ci --ignore-scripts

```

```

6
7 # ----- 2) build -----
8 FROM node:20-alpine AS builder
9 WORKDIR /app
10 COPY --from=deps /app/node_modules ./node_modules
11 COPY . .
12 # garante que o Vite leia suas VITE_* do host
13 COPY .env ./env
14 RUN npm run build
15
16 # ----- 3) runner -----
17 FROM node:20-alpine AS runner
18 WORKDIR /app
19 ENV NODE_ENV=production
20 ENV TZ=UTC
21 ENV NODE_OPTIONS=--max-old-space-size=512
22 COPY --from=builder /app/dist ./dist
23 COPY server.cjs ./server.cjs
24 USER node
25 EXPOSE 3000
26 CMD ["node", "server.cjs"]

```

Código 6.8 – Build e execução do container CSR com limites e volume de métricas

```

1 docker build -t react-csr:prod .
2
3 docker run --name react-csr \
4   -p 3002:3000 \
5   -e METRICS_PATH=/data/webvitals.ndjson \
6   -v "$PWD/metrics-csr:/data:rw" \
7   --cpus="1.00" --cpuset-cpus="0" \
8   --memory="1g" --memory-swap="1g" \
9   --pids-limit=256 --read-only --tmpfs /tmp \
10  react-csr:prod

```

6.1.4 Paridade de Recursos, Observabilidade e Procedimento de Coleta

Para evitar viés de ambiente, ambos os contêineres executam com paridade de recursos: `--cpus="1.0"`, `--cpuset-cpus="0"` (fixação na CPU 0), `--memory="1g"`, `--memory-swap="1g"`, `--pids-limit=256`, `--read-only` e `--tmpfs /tmp` (arquivo de métricas em memória quando não houver volume). No `SSR`, `/app/.next/cache` é montado em `tmpfs` para reduzir variações de disco.

Durante os testes, o uso de recursos foi acompanhado com:

```

1 docker stats next-ssr
2 docker stats react-csr

```

Esse monitoramento fornece CPU% e memória em tempo real, complementando a coleta de *Web Vitals*. O procedimento adotado foi: (i) *warm-up* com 2 acessos iniciais à mesma rota para estabilizar caches; (ii) coleta de 10–15 carregamentos por cenário (janela anônima); (iii) persistência NDJSON por cenário em volumes distintos (por exemplo, `./metrics` para **SSR** e `./metrics-csr` para **CSR**). A análise de métricas (estatísticas descritivas e percentis como p50 e p95) será apresentada em capítulo específico.

6.1.5 Reppositórios e Reprodutibilidade

O código-fonte, instruções de execução e artefatos de instrumentação estão disponíveis nos repositórios públicos da empresa fictícia WallTech:

- CSR/SPA (React+Vite): <<https://github.com/WallTechTCC/React-CSR>>
- SSR/MPA (Next.js): <<https://github.com/WallTechTCC/Next-SSR>>

Esses repositórios permitem reproduzir o experimento localmente com os mesmos limites de CPU/RAM, sistema de arquivos *read-only* e coleta de *Web Vitals* no formato NDJSON, preservando a comparabilidade entre **CSR** e **SSR**.

6.1.6 Coleta de CPU/RAM do contêiner e exportação em CSV

Para registrar o uso de CPU, memória e número de *PIDs* dos contêineres durante os cenários de teste, foi utilizada a telemetria do próprio Docker via `docker stats` (sem streaming contínuo). As amostras foram coletadas a cada 1 segundo e gravadas em arquivos CSV separados por aplicação, no diretório `metrics-host/`.

Cada linha do arquivo CSV gerado contém as seguintes colunas para a análise dos dados:

- ts: timestamp da coleta no host (YYYY-MM-DD HH:MM:SS);
- name: nome do contêiner;
- cpu_perc: percentual de CPU do contêiner (string com “%” conforme `docker stats`);
- mem_usage: uso de memória reportado (formato humano, ex.: “123.4MiB / 1.00GiB”);
- mem_perc: percentual de memória (string com “%”);
- pids: quantidade de processos/threads visíveis ao cgroup do contêiner.

O comando utilizado para a coleta no contêiner SSR (`next-ssr`), que cria o diretório de métricas e inicia a captura contínua em `metrics-host/docker-stats-ssr.csv`, é apresentado a seguir.

Código 6.9 – Captura de CPU/RAM do contêiner SSR e exportação para CSV

```

1 mkdir -p metrics-host
2 (
3   echo "ts,name,cpu_perc,mem_usage,mem_perc,pids";
4   while true; do
5     TS=$(date +"%F %T");
6     docker stats --no-stream \
7       --format "{{.Name}},{{.CPUPerc}},{{.MemUsage}},{{.MemPerc}},{{.IDs}}" \
8     next-ssr | sed "s/^/$TS,/";
9     sleep 1;
10   done
11 ) >> metrics-host/docker-stats-ssr.csv

```

De forma análoga, a coleta para o contêiner CSR (`react-csr`) foi realizada com o script abaixo, gravando os dados em `metrics-host/docker-stats-csr.csv`.

Código 6.10 – Captura de CPU/RAM do contêiner CSR e exportação para CSV

```

1 (
2   echo "ts,name,cpu_perc,mem_usage,mem_perc,pids";
3   while true; do
4     TS=$(date +"%F %T");
5     docker stats --no-stream \
6       --format "{{.Name}},{{.CPUPerc}},{{.MemUsage}},{{.MemPerc}},{{.IDs}}" \
7     react-csr | sed "s/^/$TS,/";
8     sleep 1;
9   done
10 ) >> metrics-host/docker-stats-csr.csv

```

6.1.7 Uso complementar do Lighthouse (Chrome DevTools)

O *Lighthouse* foi utilizado como apoio diagnóstico em ambiente laboratorial, para identificar gargalos e oportunidades que expliquem diferenças observadas nas medições de campo.

No contexto deste estudo, o Lighthouse foi empregado com os seguintes propósitos:

- Interatividade em laboratório: leitura do TBT (*Total Blocking Time*) como indicador de trabalho bloqueante no *main thread*;
- Oportunidades/Auditórias: insumos para orientar correções de build e de carregamento (JS/CSS, imagens, dicas de rede);

- Acessibilidade e SEO: verificação rápida de barreiras e sinalização para mecanismos de busca.

Os testes foram executados no Google Chrome, em *DevTools* → Lighthouse, no modo *Navigation*, com throttling padrão de rede/CPU, em *Mobile* e, quando pertinente, *Desktop*. Para cada cenário (página inicial, resultados de busca, detalhes da notícia) foram feitas 3 execuções, registrando-se a mediana do TBT e as principais oportunidades reportadas. Os relatórios foram exportados em JSON e HTML.

Na análise de performance, priorizou-se:

- **TBT** (*numericValue* e evidências de *long tasks*);
- Oportunidades e auditorias diretamente relacionadas a custo de carregamento e execução:
 - *render-blocking-resources*;
 - *unused-javascript* e *unused-css-rules*;
 - *total-byte-weight* e *legacy-javascript*;
 - *mainthread-work-breakdown* e *diagnostics*;
 - *modern-image-formats*, *uses-text-compression*, *preload*, *preconnect*.

Na parte de acessibilidade, foram verificados itens que impactam navegação por teclado, leitores de tela e clareza de interface:

- Semântica e estrutura: hierarquia de *headings* e *landmarks* (banner, main, nav);
- Foco e tabulação: ordem lógica, foco visível, ausência de *focus traps*;
- Rótulos e nomes acessíveis: uso adequado de `label` e atributos `aria-*`;
- Contraste e legibilidade: verificação de contraste de cor e tamanhos de fonte;
- Alternativos de mídia: textos alternativos em imagens e propósito claro de links/botões.

Dos relatórios JSON, foram lidos os `audits` mais acionáveis, como `color-contrast`, `image-alt`, `label`, `link-name`, `heading-order`, `landmark-one-main`, `focus-traps` e `logical-tab-order`.

A verificação de **SEO** concentrou-se em sinalização técnica e rastreabilidade:

- Sinalização: título e meta description presentes; `rel=canonical` consistente;
- Rastreabilidade: links rastreáveis, `robots.txt`/meta robots válidos;
- Internacionalização e dados estruturados: `hreflang` (quando aplicável) e *structured data* sem erros;
- Compatibilidade móvel: viewport configurado, fontes legíveis e *tap targets* adequados.

Foram consultados, entre outros, os audits: `document-title`, `meta-description`, `canonical`, `is-crawlable`, `robots-txt`, `hreflang`, `structured-data`, `viewport`, `font-size` e `tap-targets`.

Os relatórios foram versionados com convenção por cenário (*ex., lighthouse_csr_home_mobile*). Do JSON, foram consumidos apenas campos necessários para explicar diferenças observadas nos resultados de campo e para orientar correções de implementação.

Resultados do Lighthouse são laboratoriais, dependentes de emulação de CPU/rede e do ambiente local do avaliador. Por isso, são utilizados como indícios de causa e para priorização de melhorias, e não como substitutos das medições coletadas em execução real.

7 Resultados e Discussões

Este capítulo apresenta os achados empíricos do estudo, com foco na experiência do usuário medida por *Core Web Vitals* (coleta em campo, no cliente) e, de forma complementar, em diagnósticos laboratoriais do Lighthouse. Como contexto operacional, reporta-se também o consumo de CPU, memória e PIDs dos contêineres durante as execuções. A metodologia, o ambiente controlado e os procedimentos de coleta já foram descritos no Cap. 5; aqui concentraremos a atenção em resultados e implicações.

7.1 Resultados

Os testes cobriram as duas variantes da WallTech ([SSR/MPA](#) em Next.js e [CSR/SPA](#) em React) nas páginas: inicial, resultados de busca e detalhes da notícia. Para cada cenário, foram realizados 10–15 carregamentos após *warm-up*, e as leituras são analisadas por mediana (p50) e, quando pertinente, p95.

As fontes de dados utilizadas foram: Web Vitals (núcleo), com [TTFB](#), [FCP](#), [LCP](#), [CLS](#), [INP](#) registrados no cliente e persistidos em NDJSON; e Lighthouse (apoio), com [TBT](#) e principais oportunidades/auditorias de desempenho, além de verificações de Acessibilidade e SEO.

7.1.1 Análise e Visualização dos Dados

Os dados coletados, incluindo as métricas de *Web Vitals* e as medições de recursos do servidor, foram exportados e analisados com o auxílio do Power BI. A ferramenta foi utilizada para gerar gráficos e visualizações interativas que facilitam a interpretação dos resultados e a comparação entre os diferentes cenários ([SSR](#) e [CSR](#)). O uso do Power BI permitiu a criação de dashboards dinâmicos que ilustram claramente as tendências das métricas ao longo das execuções e facilitam a análise de variações nas diferentes páginas testadas.

7.1.2 Tratamento dos Dados

Os dados do Web Vitals foram coletados no formato NDJSON (Newline Delimited JSON) e importados como arquivos de texto no Power BI. Cada linha do arquivo correspondia a um objeto JSON individual representando uma métrica registrada no navegador.

Para estruturar essas informações, foi utilizada a funcionalidade de transformação de dados do Power BI. Cada linha foi analisada como um objeto JSON, e, em seguida, esses objetos foram expandidos em colunas, convertendo os campos internos de cada métrica em atributos tabulares.

Essa expansão resultou nas seguintes colunas principais:

- id: identificador da interação registrada.
- name: nome da métrica (por exemplo, INP, CLS, LCP).
- value: valor numérico observado para a métrica.
- rating: classificação qualitativa (*good*, *needs improvement*, *poor*).
- navigationType: tipo de navegação associada à métrica (ex: reload, back/forward).
- attribution: detalhes adicionais sobre o elemento ou evento associado à métrica.
- entries: informações brutas sobre a entrada original da medição.
- ts: timestamp da coleta.

Com os dados estruturados dessa forma, foi possível aplicar filtros, agregações e análises comparativas entre SSR e CSR, utilizando os recursos visuais e analíticos do Power BI de maneira eficiente.

7.1.3 Metas de referência

Para uma interpretação objetiva dos resultados, adotaram-se os seguintes alvos para a análise. Estes valores são baseados nas diretrizes oficiais da iniciativa *Web Vitals* do Google, que define o 75º percentil (p75) das experiências de usuário como o limiar para uma classificação "Boa" (Good).

As metas foram divididas entre as métricas de experiência real (campo) e as pontuações de auditoria (laboratório):

- Métricas de Campo (Core Web Vitals e Diagnóstico): LCP \leq 2,5 s; CLS \leq 0,10; INP \leq 200 ms; TTFB \leq 800 ms.
- Métricas de Laboratório (Auditoria Lighthouse): TBT (Lighthouse) $<$ 200 ms; Acessibilidade (LH) \geq 90; SEO (LH) \geq 90.

Dessa forma, os dados empíricos apresentados nas seções seguintes deste capítulo serão diretamente confrontados com esses limiares. A análise de desempenho não será apenas relativa (comparando CSR com SSR), mas avaliará cada arquitetura **individualmente frente a esses padrões**, determinando se ela, isoladamente, foi capaz de entregar uma experiência de usuário considerada "Boa".

7.1.4 Resultados Aplicação SSR

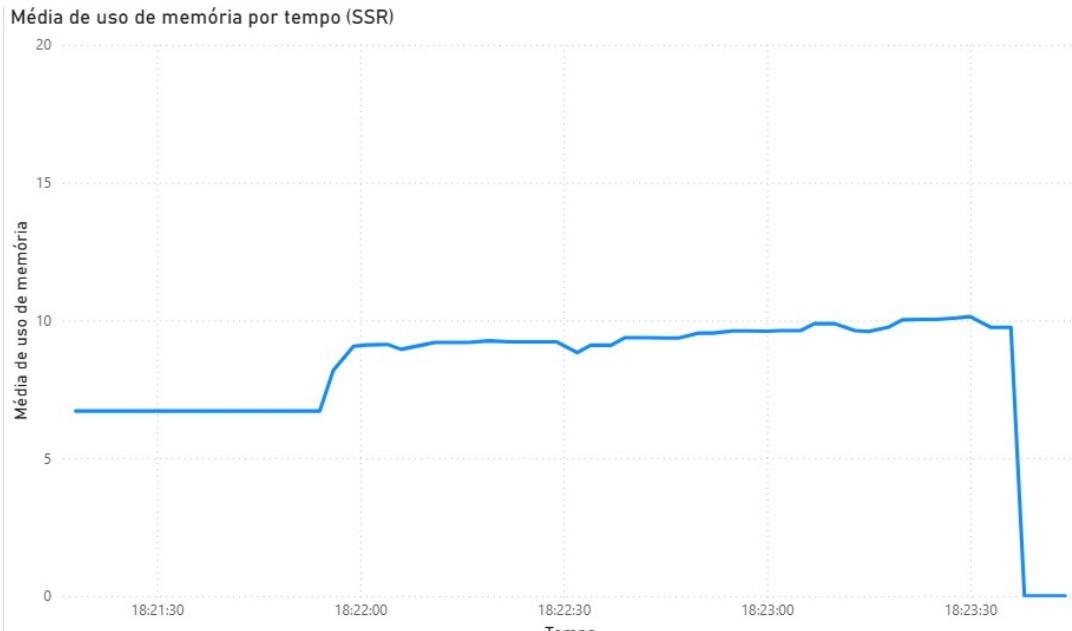
Durante os testes com a aplicação SSR (Next.js), o contêiner apresentou uso de CPU consistentemente baixo e estável, variando aproximadamente entre 8% e 13%, sem registros de picos relevantes. A redução observada ao final da série coincide com o encerramento do experimento, conforme ilustrado na Figura 23. No que diz respeito à memória, o consumo manteve-se praticamente constante ao longo das execuções, com variação mínima e uma leve queda apenas ao término dos testes, como visto na Figura 24. Esse comportamento é compatível com a arquitetura SSR, que realiza processamento sob demanda a cada requisição, e não indica sinais de saturação ou uso excessivo de recursos.

Figura 23 – Média de uso de CPU por tempo (SSR)



Fonte: os autores.

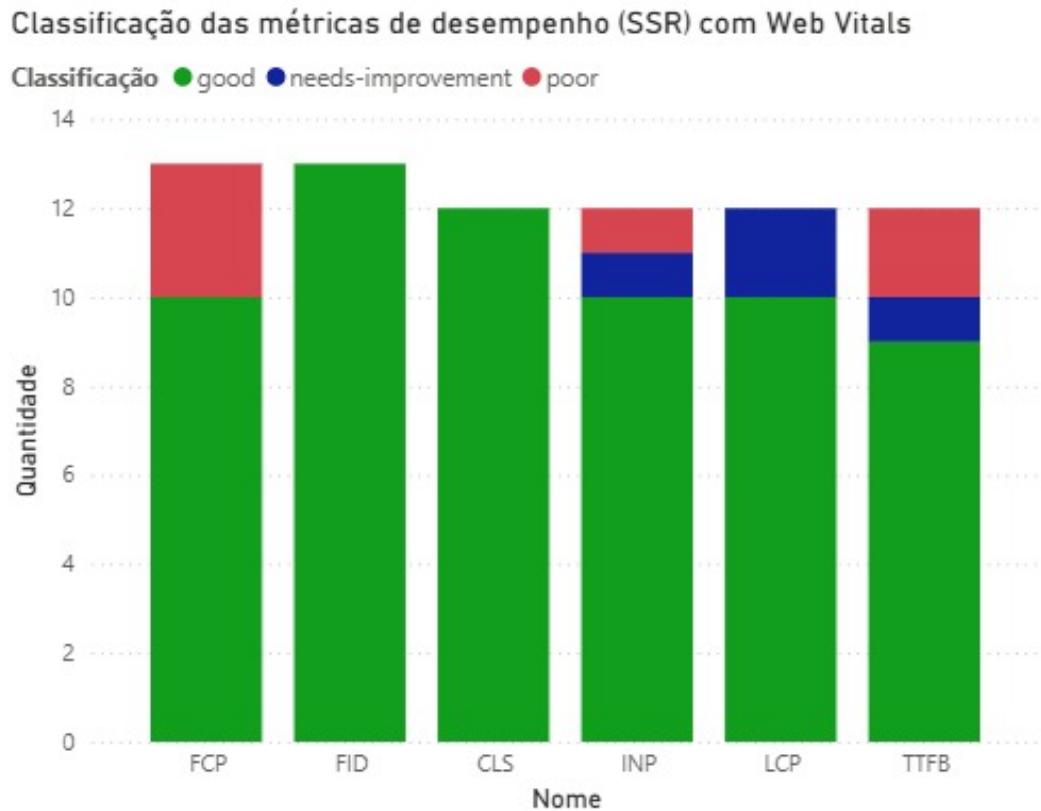
Figura 24 – Média de uso de memória por tempo (SSR)



Fonte: os autores.

Conforme a Figura 25, as leituras de campo foram, em sua maioria, classificadas como *good*. Os poucos casos de *needs-improvement* concentraram-se em LCP e INP, e houve uma fração reduzida de FCP em *poor*. Esse comportamento é compatível com custos de *hydration* e com elementos visuais de grande porte na dobraria (imagens *hero*). Em relação às metas adotadas, os resultados situam-se, em geral, próximos ou dentro dos limites de referência: LCP \leq 2,5 s; CLS \leq 0,10; INP \leq 200 ms; TTFB \leq 800 ms.

Figura 25 – Classificação das métricas de desempenho (SSR) com Web Vitals



Fonte: os autores.

Para explicar variações residuais e priorizar melhorias, auditamos as rotas *home*, *busca* e *detalhe* no modo *Navigation* do Chrome DevTools, considerando três indicadores: TBT, Acessibilidade e SEO. A Tabela 3 consolida as medianas observadas nos relatórios HTML anexados (`ssrMobile.html`, `ssrMobileBusca.html`, `ssrtestePage.html`).

Tabela 3 – Lighthouse (SSR) — mediana por rota

Rota	TBT (ms)	Acessibilidade (%)	SEO (%)
Home	12	95	100
Busca	10	99	100
Detalhe	0–7 ¹	100	100

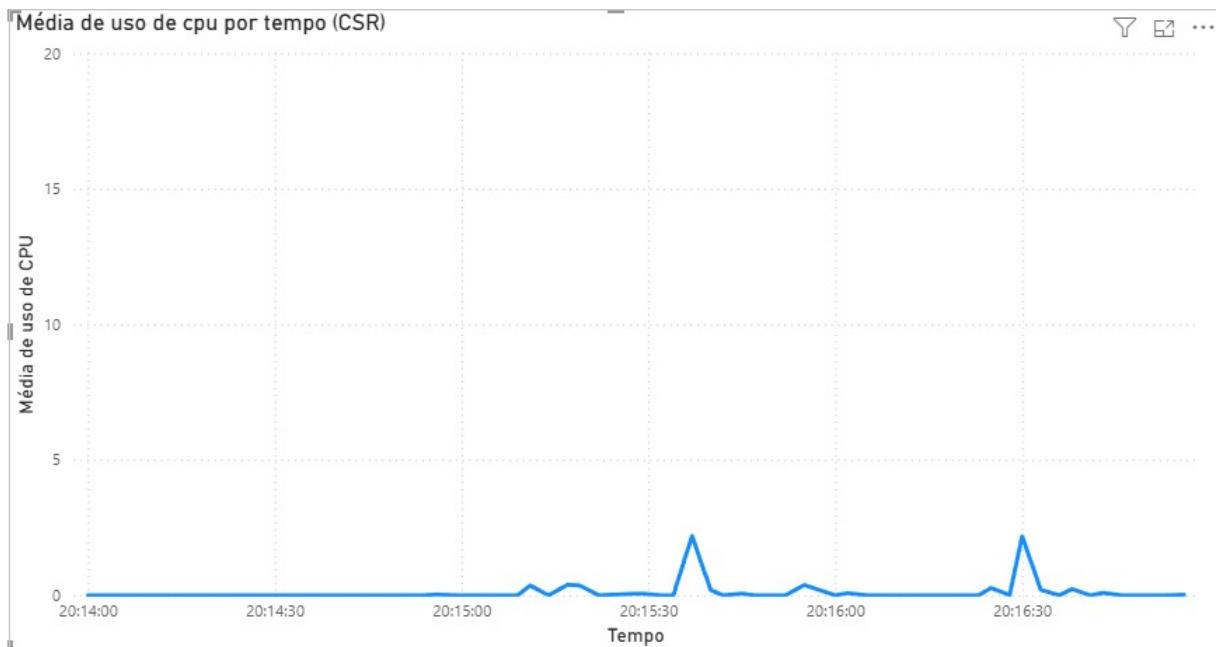
Leitura. (i) TBT muito abaixo da meta interna (< 200 ms) em todas as rotas, corroborando INP em *good*; (ii) Acessibilidade alta (95–100%), com ajustes residuais (hierarquia de headings, foco visível, rótulos) de fácil endereçamento; (iii) SEO consistente (100

¹ Variações muito baixas entre execuções; mediana ≈ 5 ms.

7.1.5 Resultados Aplicação CSR

O gráfico apresentado ([Figura 26](#)) mostra a média de uso de CPU ao longo do tempo para a aplicação *SPA* utilizando a abordagem CSR. Observa-se que o uso da CPU permanece baixo e estável durante a maior parte do período analisado, com pequenas flutuações ocasionais, que não ultrapassam 5%. Essas oscilações podem ser atribuídas a momentos específicos de interação mais intensiva ou carregamento de dados dinâmicos, mas, no geral, a aplicação apresenta um desempenho eficiente em termos de consumo de CPU.

Figura 26 – Média de uso de CPU por tempo (CSR)



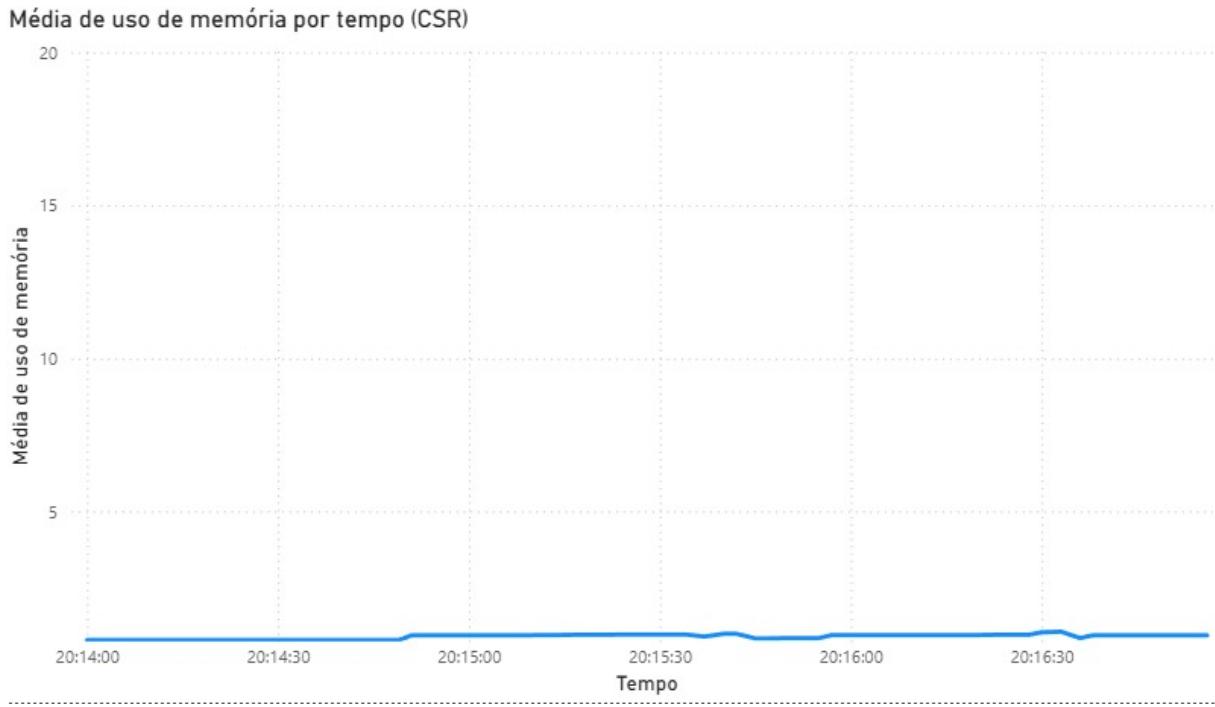
Fonte: os autores.

Os picos de uso da CPU observados são discretos e ocorrem em intervalos relativamente curtos, o que sugere que a aplicação é capaz de lidar com as operações de renderização e interação sem causar grandes sobrecargas no sistema. Esse comportamento é típico de aplicações bem otimizadas que não exigem processamento pesado durante a navegação e que priorizam a responsividade sem comprometer a estabilidade.

Antes de analisarmos os resultados de CPU, é importante observar o uso de memória da aplicação com a abordagem CSR. A eficiência do consumo de memória é um indicador crucial para a performance da aplicação, especialmente em dispositivos com recursos limitados. O gráfico a seguir ilustra a média de uso de memória por tempo durante o ensaio.

A memória permaneceu praticamente constante e em patamar reduzido durante todo o ensaio, coerente com a entrega estática e a execução da lógica no cliente ([Figura 27](#)).

Figura 27 – Média de uso de memória por tempo (CSR)

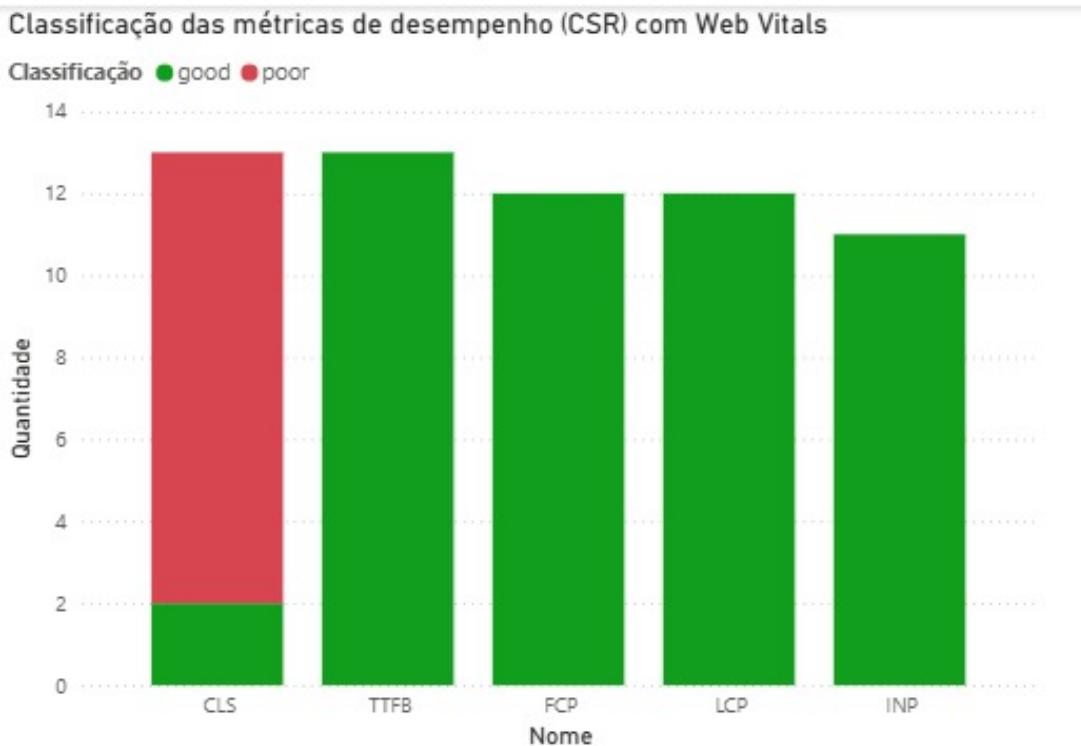


Fonte: os autores.

Não se observaram sinais de saturação, o que indica uma boa otimização da aplicação para evitar o consumo excessivo de recursos, mesmo durante o uso prolongado. Esse comportamento sugere que a aplicação CSR está bem adaptada para dispositivos com diferentes capacidades de memória, oferecendo uma experiência estável sem sobrecarregar o sistema. Esse uso controlado de memória é uma característica importante para garantir que o desempenho da aplicação seja mantido, mesmo sob condições de uso intensivo.

As leituras de campo indicam predomínio de *good* em **TTFB**, **FCP**, **LCP** e **INP** (Figura 28). O ponto fora da curva é a **CLS**, com concentrações em *poor*. Esse padrão é típico de SPAs quando ocorrem *layout shifts* durante a hidratação ou quando imagens/slots não reservam espaço antes do carregamento (dimensões ausentes, fontes sem *preload*, inserções acima da dobra). Em relação às metas, as leituras ficaram, em geral, dentro ou próximas dos limiares de referência ($LCP \leq 2,5\text{ s}$; $CLS \leq 0,10$; $INP \leq 200\text{ ms}$; $TTFB \leq 800\text{ ms}$), com a ressalva da **CLS**.

Figura 28 – Classificação das métricas de desempenho (CSR) com Web Vitals



Fonte: os autores.

Para explicar oscilações residuais e orientar priorizações, auditamos as rotas críticas no *Chrome DevTools* (modo *Navigation*), com três indicadores: TBT, Acessibilidade e SEO. Os achados indicam que o TBT se manteve em patamar muito baixo (dezenas de milissegundos), compatível com **INP** em *good*. Os escores de Acessibilidade foram altos na *home* (93–94 %), com ajustes residuais envolvendo hierarquia de *headings*, foco visível e rótulos/*alt*, conforme identificado nos relatórios “Home (Web)” e “Home (Mobile)”. Já em SEO, a *home* obteve cerca de 83 % (Web e Mobile), apontando oportunidades como preenchimento de *meta description*, definição de *canonical*, configurações de *robots* e uso de *hreflang*, quando aplicável.

Em conjunto, os relatórios apontam que o gargalo do CSR não está em bloqueio de *main thread* (TBT baixo), mas em estabilidade visual e em alguns sinais de SEO. Na prática, os seguintes ajustes tendem a capturar os ganhos: (i) reservar dimensões de imagens e

cards acima da dobra (`width/height` ou `aspect-ratio`); (ii) *preload* de fontes e imagens *hero* (com `font-display: swap`); (iii) evitar inserções DOM acima da dobra durante a hidratação; (iv) completar metadados de SEO (título/descrição/`canonical/robots`) e revisar indexabilidade.

Servidor praticamente ocioso (CPU e RAM baixos); *Web Vitals* em boa forma para **TTFB/FCP/LCP/INP**, com atenção especial à **CLS**. O *Lighthouse* confirmou TBT reduzido, Acessibilidade alta e SEO com melhorias de rápida implementação, coerentes com o perfil de SPA e suas responsabilidades no cliente.

7.1.6 Comparação entre SSR e CSR

Esta seção compara, de forma integrada, experiência do usuário (*Web Vitals* em campo), diagnóstico laboratorial (*Lighthouse*: TBT, Acessibilidade e SEO) e custo operacional (CPU/Memória nos contêineres). A análise revela que a escolha arquitetônica representa um trade-off fundamental sobre onde a carga computacional deve residir: no servidor ou no cliente. A leitura está ancorada nas medianas (p50) e, quando pertinente, no p95 dos cenários home, busca e detalhe.

Tabela 4 – Síntese comparativa dos resultados (SSR × CSR) neste estudo

Aspecto	SSR (MPA/Next.js)	CSR (SPA/React)
CPU no servidor	Baixa e estável (~8–13%), sem picos relevantes.	Muito baixa; oscilações discretas com picos < 5%.
Memória no servidor	Estável, variação estreita; queda apenas ao término dos testes.	Muito baixa e praticamente constante.
Web Vitals (campo)	Predominância de <i>good</i> ; pequenos trechos <i>needs-improvement</i> em LCP/INP e fração <i>poor</i> em FCP (páginas com imagens <i>hero</i> e pós-hidratação).	TTFB/FCP/LCP/INP majoritariamente <i>good</i> ; CLS com ocorrências <i>poor</i> (shifts na montagem/hidratação e elementos sem reserva de espaço).
<i>Lighthouse</i> : TBT	Muito baixo (home ≈12 ms; busca ≈10 ms; detalhe ≈5 ms).	Muito baixo (ordem de dezenas de ms na home).
<i>Lighthouse</i> : Acessibilidade	Alta (95–100%).	Alta (93–94% na home).
<i>Lighthouse</i> : SEO	100% nas rotas auditadas.	~83% na home; indica metadados/sinalizações a completar.
Leitura operacional	Parte da renderização no servidor melhora TTFB/ <i>first paint</i> e favorece SEO; cuidado com <i>hydration</i> e imagens de grande porte.	Renderização no cliente alivia o servidor e mantém TBT/INP baixos; requer disciplina de layout para evitar CLS e ajustes de SEO.

Entre os pontos fortes do SSR, destaca-se o tempo até a primeira resposta ou conteúdo visível. O HTML pré-renderizado acelera a exibição inicial, refletindo-se em *TTFB* e *FCP* consistentes e TBT residual. Essa agilidade é um fator psicológico importante, transmitindo eficiência e sendo crucial para reter a atenção do usuário nos primeiros segundos. Além disso, a descoberta e rastreabilidade são favorecidas: o SEO atingiu 100% nas rotas auditadas, reforçando a adequação do SSR para páginas públicas e conteúdo editorial. A estabilidade visual também se mostrou sólida, com índice de *CLS* praticamente nulo, e a previsibilidade de performance é uma vantagem em redes ou dispositivos mais limitados, pois a renderização no servidor reduz o risco de tarefas longas no *main thread* do navegador.

Por outro lado, o SSR apresenta alguns pontos fracos. O custo de hidratação e navegação pode afetar o desempenho, especialmente em páginas com muito JavaScript, degradando *LCP* e *FCP*. Além disso, a navegação subsequente, ao exigir nova requisição completa ao servidor, pode quebrar a fluidez da experiência. Embora o custo no servidor tenha sido baixo neste estudo, o modelo de processamento por requisição aumenta discretamente o uso de CPU e memória, com implicações para escalabilidade e infraestrutura. Também vale mencionar que estratégias mais avançadas, como *streaming* ou *Server Components*, elevam a complexidade do build e do deploy.

No CSR, os pontos fortes incluem a interatividade contínua: após o carregamento inicial, a navegação entre seções ocorre quase instantaneamente, com *TBT* e *INP* muito baixos, aproximando-se da experiência de um aplicativo nativo. O custo operacional também é significativamente menor, pois o servidor atua apenas como provedor de arquivos estáticos. Isso permite escalabilidade horizontal simples e integração natural com CDNs e cache.

No entanto, há desafios importantes. A estabilidade visual é comprometida — a métrica *CLS* apresentou ocorrências *poor* devido a *layout shifts* durante a montagem ou hidratação, especialmente quando imagens e slots não reservam espaço. Em SEO, os escores ficaram em torno de 83%, sugerindo a necessidade de completar metadados e sinalizações essenciais. Por fim, a dependência do JavaScript no cliente torna o *LCP* mais sensível: o tempo de carregamento inicial tende a ser mais lento, especialmente em dispositivos modestos, caso o *bundle JS* não esteja bem otimizado.

Por fim, cabe uma observação sobre a ausência do FID no CSR. Nessa arquitetura, o *First Input Delay (FID)* não é capturado de forma tradicional, pois a página precisa ser hidratada antes de se tornar interativa. Esse processo impede que o *FID* meça corretamente a primeira interação do usuário, tornando-o menos relevante em ambientes CSR. Por esse motivo, o *Interaction to Next Paint (INP)* é adotado como métrica alternativa, pois reflete o tempo entre a interação e a atualização visual, oferecendo uma avaliação mais precisa

da interatividade em páginas dinâmicas.

7.1.7 Discussão

Os resultados alinharam-se e fornecem validação empírica para as recomendações consolidadas na literatura, confirmando que não existe uma solução universalmente superior. A escolha é uma decisão estratégica que transcende a tecnologia.

(i) Recursos do servidor. O **SSR** consumiu levemente mais CPU/memória (ainda baixos), resultado do processamento por requisição. O **CSR**, por outro lado, opera como *static hosting*, com custo mínimo no host, transferindo a carga computacional para o dispositivo do usuário.

(ii) Experiência do usuário. Em **SSR**, a entrega inicial é perceptivelmente rápida e os escores de SEO e Acessibilidade são superiores. Em **CSR**, a interatividade após o carregamento é o grande trunfo, mas o principal cuidado é a CLS, que pode degradar a experiência.

(iii) Coerência entre campo e laboratório. Os *Web Vitals* em campo e o TBT do Lighthouse convergiram, indicando baixo bloqueio de *main thread* em ambos os modelos. As divergências em CLS (CSR) e SEO(CSR < SSR) são áreas classicamente sensíveis à estratégia de renderização.

(iv) Análise Econômica e Viabilidade. A análise econômica dos recursos reforça o apelo do **CSR** em cenários de alto tráfego com orçamento restrito, devido ao seu modelo de *static hosting*, que minimiza custos com servidores e escalabilidade. Em contraste, o **SSR**, embora ofereça vantagens competitivas em SEO que podem resultar em maior retorno sobre o investimento (ROI) para plataformas dependentes de tráfego orgânico (como notícias), exige um investimento inicial e operacional maior em infraestrutura para processar a renderização no lado do servidor. A viabilidade econômica, portanto, está intrinsecamente ligada à fonte de receita e à escala de tráfego esperada.

7.1.8 Implicações práticas

A decisão entre as arquiteturas deve ser ponderada à luz dos objetivos do produto, das expectativas do usuário e das capacidades da equipe de desenvolvimento.

- Priorizar SSR para aplicações de conteúdo, como portais de notícias, plataformas de e-commerce e blogs, onde a otimização para mecanismos de busca (SEO) é vital, e a velocidade da primeira impressão (TTFB e FCP) é um fator determinante para o sucesso do negócio.

- Priorizar CSR para aplicações web ricas e complexas, como dashboards analíticos, sistemas de gestão interna (ERPs) e plataformas de software como serviço (SaaS), que geralmente são acessadas após autenticação e onde a fluidez da interatividade contínua (INP) é mais valorizada do que a indexabilidade de páginas individuais.

7.1.9 Ajustes recomendados

Server-Side Rendering (SSR) (mitigar LCP/INP). Otimizar e dimensionar imagens (usar `priority` para *hero* e *lazy* abaixo da dobra), considerar *streaming/partial hydration/Server Components*, reduzir não crítico e declarar `preload/dns-prefetch` para recursos essenciais.

Client-Side Rendering (CSR) (mitigar CLS e elevar SEO). Reservar dimensões de mídia (`width/height` ou `aspect-ratio`); evitar inserções acima da dobra durante a hidratação; `font-display: swap` com `preload` de fontes e imagens *hero*; placeholders do tamanho final; completar metadados e sinalizações (`title/descriptioncanonical/robots/hreflang`, quando aplicável).

7.1.10 Síntese

No cenário testado, **SSR** e **CSR** entregaram boa experiência de uso com baixa pressão de recursos. A **Server-Side Rendering (SSR)** destacou-se por **SEO** e exibição inicial mais previsível, sendo ideal para aplicações de conteúdo. A **Client-Side Rendering (CSR)** reduziu o custo no servidor e garantiu interatividade fluida, com a contrapartida de maior propensão a **CLS** e desafios de **SEO**, mostrando-se adequada para aplicações ricas e logadas. A decisão final deve considerar o perfil de tráfego, a exigência de descoberta e o tipo de interação, aplicando as otimizações para mitigar os pontos fracos identificados em cada abordagem.

8 Conclusão

Este trabalho de conclusão de curso realizou uma análise comparativa aprofundada entre as arquiteturas de renderização no lado do cliente ([Client-Side Rendering \(CSR\)](#)) e no lado do servidor ([Server-Side Rendering \(SSR\)](#)), com o intuito de dissecar suas implicações na performance, na experiência do usuário e nos desafios de implementação. A partir do desenvolvimento e teste de duas versões da plataforma "WallTech" em um ambiente controlado, foi possível validar empiricamente os trade-offs inerentes a cada abordagem.

A investigação confirmou que a escolha arquitetônica molda fundamentalmente a jornada do usuário. A abordagem [SSR](#) demonstrou ser superior na fase inicial da interação, entregando um primeiro conteúdo visual ([First Contentful Paint \(FCP\)](#)) de forma notavelmente rápida e garantindo excelente estabilidade visual ([Cumulative Layout Shift \(CLS\)](#)). Essa performance inicial, aliada à otimização natural para mecanismos de busca ([Search Engine Optimization \(SEO\)](#)), a torna ideal para aplicações onde a primeira impressão e a descoberta de conteúdo são críticas.

Em contrapartida, o [CSR](#) redefiniu a experiência após o carregamento inicial, oferecendo uma interatividade superior e navegação quase instantânea, refletida em um baixo [Interaction to Next Paint \(INP\)](#), que emula a fluidez de um aplicativo nativo. Contudo, essa vantagem vem ao custo de um carregamento inicial mais lento e de uma maior suscetibilidade a instabilidades de layout, exigindo uma disciplina de desenvolvimento rigorosa para mitigar os pontos fracos da abordagem.

Portanto, este trabalho cumpriu seus objetivos ao demonstrar, com dados concretos, que a decisão entre [CSR](#) e [SSR](#) é uma escolha estratégica. Ela se resume a um trade-off fundamental sobre onde a carga computacional deve residir e o que se deve priorizar: a velocidade da primeira renderização e a indexabilidade do [SSR](#), ou a interatividade contínua e a redução de custos de servidor do [CSR](#).

A principal contribuição desta pesquisa reside na sua metodologia sistemática e reproduzível, que oferece um roteiro prático para a avaliação comparativa de arquiteturas frontend. Como limitação, reconhece-se que o ambiente de teste local, embora essencial para o controle experimental, não captura a variabilidade de redes e dispositivos do mundo real.

Para trabalhos futuros, sugere-se a análise de arquiteturas híbridas, como [Static Site Generation \(SSG\)](#) e [Incremental Static Regeneration \(ISR\)](#), que prometem unir o melhor dos dois mundos. Adicionalmente, seria de grande valia investigar o impacto de paradigmas

emergentes, como os *React Server Components* e o *Streaming SSR*, e complementar os dados quantitativos com estudos qualitativos de usabilidade, para aprofundar a compreensão sobre a percepção real do usuário.

Referências

Amazon Web Services. *Front-end x back-end — Diferença entre desenvolvimento de aplicações*. 2024. Acessado em 14 abr. 2025. Disponível em: <<https://aws.amazon.com/pt/compare/the-difference-between-frontend-and-backend/>>. Citado na página 23.

ANGKASA, H. et al. Improving universal rendering performance on nuxtjs-based web application. In: . [s.n.], 2023. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85187776927&doi=10.1109%2fCITSM60085.2023.10455297&partnerID=40&md5=48f7f413665c85291a128368743ec2d1>>. Citado 2 vezes nas páginas 60 e 61.

ANGULAR. 2025. Acessado em: 11 de abril de 2025. Disponível em: <<https://angular.io/guide/architecture>>. Citado na página 42.

ANGULAR2024. 2024. Acessado em: 10 de abril de 2025. Disponível em: <<https://angular.io/guide/universal>>. Citado na página 43.

ANURAG. What the heck is web rendering? *Locofy Blog*, 2024. Acesso em: 15 maio 2025. Disponível em: <<https://www.locofy.ai/blog/what-the-heck-is-web-rendering>>. Citado na página 38.

ASTRO. 2024. Acessado em: 10 de abril de 2025. Disponível em: <<https://astro.build/>>. Citado na página 43.

BALLERINI, R. Html, css e javascript: qual a diferença entre essas linguagens? *Alura*, 2023. Acessado em 14 abr. 2025. Disponível em: <<https://www.alura.com.br/artigos/html-css-e-js-definicoes>>. Citado 2 vezes nas páginas 23 e 24.

BEKMANOVA, G. et al. Requirements for the development of a website builder with adaptive design. In: . [s.n.], 2024. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85215519877&doi=10.1109%2fUBMK63289.2024.10773412&partnerID=40&md5=6192f2d980aa9bbad06a979bd633a64f>>. Citado 2 vezes nas páginas 60 e 61.

BOEHNCKE, G. Corporate social responsibility: Hiring requisition in media companies? *CSR, Sustainability, Ethics and Governance*, 2023. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85151569634&doi=10.1007%2f978-3-031-18976-0_7&partnerID=40&md5=87204e857ad89df0c6a61759a475f0fe>. Citado na página 60.

BOSE, T. Frontend rendering: Ssg vs isr vs ssr vs csr - when to use which? *DEV.to*, 2022. Acessado em 11 maio 2025. Disponível em: <<https://dev.to/rupphysuppy/frontend-rendering-ssg-vs-isr-vs-ssr-vs-csr-when-to-use-which-5jp>>. Citado 3 vezes nas páginas 34, 35 e 36.

CARVALHO, F. M. Progressive server-side rendering with suspendable web templates. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2025. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85211240232&doi=10.1007%2f978-3-031-18976-0_7&partnerID=40&md5=87204e857ad89df0c6a61759a475f0fe>. Citado na página 23.

[2f978-981-96-0576-7_33&partnerID=40&md5=48a28d5d4b14294477317d41fefa0c1f>](https://doi.org/10.5281/zenodo.981960). Citado 2 vezes nas páginas 60 e 61.

CHACON, S.; STRAUB, B. *Pro Git*. 2. ed. Apress, 2014. Acessado em 14 de Julho de 2025. Disponível em: <<https://git-scm.com/book/en/v2>>. Citado 2 vezes nas páginas 48 e 49.

Cloudflare. *O que é HTTP?* 2025. Acessado em 12 de abril de 2025. Disponível em: <<https://www.cloudflare.com/pt-br/learning/ddos/glossary/hypertext-transfer-protocol-http/>>. Citado na página 22.

CONNER, G. Tornando o instagram.com mais rápido: Parte 2. *Engenharia do Instagram*, 2019. Disponível em: <<https://instagram-engineering.com/making-instagram-com-faster-part-2-f350c8fba0d4>>. Citado na página 16.

ControleNet. *Cliente-Servidor, uma estrutura lógica para a computação centralizada*. 2025. Disponível em: <<https://www.controle.net/faq/cliente-servidor-uma-estrutura-para-a-computacao-centralizada>>. Citado na página 19.

Docker, Inc. *Dockerfile reference*. 2025. Acesso em: set. 2025. Disponível em: <<https://docs.docker.com/reference/dockerfile/>>. Citado 4 vezes nas páginas 51, 52, 53 e 54.

Docker, Inc. *What is Docker?* 2025. Acesso em: set. 2025. Disponível em: <<https://docs.docker.com/get-started/docker-overview/>>. Citado 3 vezes nas páginas 51, 52 e 54.

EMADAMERHO-ATORI, N. Ux tips for developers. *Prismic Blog*, 2023. Disponível em: <<https://prismic.io/blog/ux-tips-for-developers>>. Citado na página 44.

EMADAMERHO-ATORI, N. Client-side rendering (csr) vs. server-side rendering (ssr). *Prismic Blog*, 2024. Disponível em: <<https://prismic.io/blog/client-side-vs-server-side-rendering>>. Citado 11 vezes nas páginas 16, 26, 29, 30, 31, 32, 44, 46, 47, 78 e 79.

Evan You and Vite contributors. *Vite Documentation*. 2025. Acessado em 23 de Julho de 2025. Disponível em: <<https://vitejs.dev/>>. Citado na página 83.

FOUNDATION, T. jQuery. *jQuery - Fast, small, and feature-rich JavaScript library*. 2025. Acessado em: 11 maio 2025. Disponível em: <<https://jquery.com>>. Citado na página 42.

Gatsby. Rendering options in gatsby. *Gatsby Documentation*, 2023. Acesso em: 15 maio 2025. Disponível em: <<https://www.gatsbyjs.com/docs/conceptual/rendering-options>>. Citado 2 vezes nas páginas 37 e 38.

GitHub. *GitHub Docs: About GitHub*. 2025. Acessado em 14 de Julho de 2025. Disponível em: <<https://docs.github.com/pt>>. Citado 2 vezes nas páginas 48 e 49.

GitHub. *GitHub Docs: About Projects*. 2025. Acessado em 14 de Julho de 2025. Disponível em: <<https://docs.github.com/pt/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects>>. Citado 2 vezes nas páginas 49 e 50.

- GODBOLT, M. *Frontend Architecture for Design Systems*. O'Reilly Media, Inc., 2016. ISBN 9781491926734. Disponível em: <<https://www.oreilly.com/library/view/frontend-architecture-for/9781491926772/>>. Citado na página 16.
- GOMES, A. *Agile: Desenvolvimento de software com entregas frequentes e foco no valor de negócio*. Casa do Código, 2014. ISBN 9788566250992. Disponível em: <<https://books.google.com.br/books?id=zHCCCwAAQBAJ>>. Citado na página 71.
- GOOGLE. 2010. Acessado em: 14 de fevereiro de 2025. Disponível em: <<https://developers.google.com/search/blog/2010/04/using-site-speed-in-web-search-ranking>>. Citado 2 vezes nas páginas 15 e 45.
- Google Search Central. *Evaluating page experience for a better web*. 2020. Acessado em: 30 set. 2025. Disponível em: <<https://developers.google.com/search/blog/2020/05/evaluating-page-experience>>. Citado na página 40.
- GTmetrix. *Analyze your page's speed with GTmetrix*. 2025. Acessado em: 30 set. 2025. Disponível em: <<https://gtmetrix.com/>>. Citado na página 39.
- GÜNAÇAR, O. Time to first byte (ttfb) – easy to understand, difficult to improve. *OnCrawl*, 2025. Acessado em: 10 de abril de 2025. Disponível em: <<https://www.oncrawl.com/technical-seo/time-to-first-byte-what-and-why-important-part-1/>>. Citado na página 33.
- ISKANDAR, T. F. et al. Comparison between client-side and server-side rendering in the web development. In: . [s.n.], 2020. All Open Access, Gold Open Access. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85086446329&doi=10.1088%2f1757-899X%2f801%2f1%2f012136&partnerID=40&md5=1e76e6274991560833731fefce47c3a5>>. Citado 2 vezes nas páginas 60 e 61.
- JAVASCRIPT. 2025. Acessado em: 11 de abril de 2025. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Learn_web_development/Core/Scripting/What_is_JavaScript>. Citado na página 24.
- KATTAH, A. Html, css e javascript: Entenda as diferenças na prática. *HeroCode*, 2023. Acessado em 14 abr. 2025. Disponível em: <<https://herocode.com.br/blog/html-css-javascript-diferencias/>>. Citado 2 vezes nas páginas 23 e 25.
- KESHARI, P. et al. Web development using reactjs. In: . [s.n.], 2023. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85195810274&doi=10.1109%2fICAC3N60023.2023.10541743&partnerID=40&md5=7f9f99f1b7f492e6f4f5c43c2b6b5be3>>. Citado 2 vezes nas páginas 60 e 61.
- KLOCHKOV, D.; MULAWKA, J. Improving ruby on rails-based web application performance. *Information (Switzerland)*, 2021. All Open Access, Gold Open Access. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85113170428&doi=10.3390%2finfo12080319&partnerID=40&md5=23cc2185b9d757e35194c50632613e1f>>. Citado 2 vezes nas páginas 60 e 61.
- KOWALCZYK, K.; SZANDALA, T. Enhancing seo in single-page web applications in contrast with multi-page applications. *IEEE Access*, 2024. All Open Access, Gold Open Access. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=>>

2-s2.0-85182953693&doi=10.1109%2fACCESS.2024.3355740&partnerID=40&md5=20893a5fc228854aa2846fb5d47a9da4>. Citado 2 vezes nas páginas 60 e 61.

LACOM, P.; SAGOT, S. A research framework for b2b green marketing innovation: the design of sustainable websites. In: . [s.n.], 2022. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85148699908&doi=10.1109%2fICE%2fITMC-IAMOT55089.2022.10033239&partnerID=40&md5=acca2bbae04d3cd37bc8b94e433f9855>>. Citado 2 vezes nas páginas 60 e 61.

MDN Web Docs. *HTTP - Visão Geral*. 2024. Acessado em 12 de abril de 2025. Disponível em: <<https://developer.mozilla.org/pt-BR/docs/Web/HTTP>>. Citado 2 vezes nas páginas 21 e 22.

MURPHY, C. Guide: React seo considerations and solutions. *Prismic Blog*, 2022. Disponível em: <<https://prismic.io/blog/react-seo-solutions>>. Citado 2 vezes nas páginas 44 e 45.

NEARY, A. Rearchitecting airbnb's frontend. *Medium - Airbnb Engineering*, 2017. Disponível em: <<https://medium.com/airbnb-engineering/rearchitecting-airbnbs-frontend-5e213efc24d2>>. Citado na página 16.

News API. *News API Documentation*. 2025. Acessado em 14 de julho de 2025. Disponível em: <<https://newsapi.org/docs>>. Citado na página 50.

NEXT. 2024. Acessado em: 10 de abril de 2025. Disponível em: <<https://nextjs.org/>>. Citado 2 vezes nas páginas 43 e 84.

NODE. 2025. Acessado em: 11 de abril de 2025. Disponível em: <<https://nodejs.org/en/docs/>>. Citado 2 vezes nas páginas 24 e 84.

NUXT. 2024. Acessado em: 10 de abril de 2025. Disponível em: <<https://nuxtjs.org/>>. Citado na página 43.

OSMANI, A.; MILLER, J. Rendering on the web. 2025. Acessado em 15 maio 2025. Disponível em: <<https://web.dev/articles/rendering-on-the-web?hl=pt-br>>. Citado 6 vezes nas páginas 25, 26, 27, 28, 40 e 78.

PAVIC, F.; BRKIC, L. Methods of improving and optimizing react web-applications. In: . [s.n.], 2021. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85123053514&doi=10.23919%2fMIPRO52101.2021.9596762&partnerID=40&md5=104afe6916f3fd1c2a46c8c52ca04519>>. Citado 2 vezes nas páginas 60 e 61.

PERERA, P. Visual explanation and comparison of csr, ssr, ssg, and isr. *DEV.to*, 2022. Acessado em 11 maio 2025. Disponível em: <<https://dev.to/pahanperera/visual-explanation-and-comparison-of-csr-ssr-ssg-and-isr-34ea>>. Citado 3 vezes nas páginas 34, 36 e 37.

POKHRIYAL, A. et al. Single page optimization techniques using react. In: . [s.n.], 2024. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85199175705&doi=10.1201%2f9781032644752-63&partnerID=40&md5=667c007334673fbaaaedbfff1c1ce4c>>. Citado 2 vezes nas páginas 60 e 61.

PORZIO, C. *Alpine.js - A rugged, minimal framework for composing JavaScript behavior in your HTML*. 2023. Acessado em: 11 maio 2025. Disponível em: <<https://alpinejs.dev>>. Citado na página 42.

PROCEEDINGS of the 19th International Conference on Web Information Systems and Technologies, WEBIST 2023. In: . [s.n.], 2023. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85179780169&partnerID=40&md5=ded8f04f4e9bdb35c331c38030709335>>. Citado 2 vezes nas páginas 60 e 61.

QWIK. 2024. Acessado em: 10 de abril de 2025. Disponível em: <<https://qwik.builder.io/>>. Citado na página 43.

REACT. 2025. Acessado em: 11 de abril de 2025. Disponível em: <<https://react.dev/learn>>. Citado 2 vezes nas páginas 42 e 83.

REMIX. 2024. Acessado em: 10 de abril de 2025. Disponível em: <<https://remix.run/>>. Citado na página 43.

Remix. *React Router Documentation*. 2025. Acessado em 23 de Julho de 2025. Disponível em: <[\[https://reactrouter.com/en/main\]\(https://reactrouter.com/en/main\)](https://reactrouter.com/en/main)>. Citado na página 83.

shadcn/ui. *shadcn/ui Documentation*. 2025. Acessado em 14 de julho de 2025. Disponível em: <<https://ui.shadcn.com/docs>>. Citado na página 48.

SHOPIFY. *Website Load Time Statistics: Why Speed Matters in 2024*. 2024. <Https://www.shopify.com/blog/website-load-time-statistics>. Acesso em: 07 maio 2025. Citado na página 45.

SITEEFY. *How many websites are there?* 2021. Disponível em: <<https://siteefy.com/how-many-websites-are-there/>>. Citado na página 15.

SMITH, C. Time to first byte (ttfb) – easy to understand, difficult to improve. *OuterBox Design*, 2025. Acessado em: 10 de abril de 2025. Disponível em: <<https://www.outerboxdesign.com/uncategorized/time-to-first-byte-ttfb>>. Citado na página 33.

STUDIO, P. *The Impact of Client-Side Rendering on Accessibility*. 2023. Acessado em: 07 maio 2025. Disponível em: <<https://blog.pixelfreestudio.com/the-impact-of-client-side-rendering-on-accessibility>>. Citado 2 vezes nas páginas 46 e 47.

STUDIO, P. *The Impact of Client-Side Rendering on User Experience*. 2023. Acessado em: 07 maio 2025. Disponível em: <<https://blog.pixelfreestudio.com/the-impact-of-client-side-rendering-on-user-experience>>. Citado na página 46.

SUTTON, M. *Accessibility Tips in Single-Page Applications*. 2018. <Https://www.deque.com/blog/accessibility-tips-in-single-page-applications>. Deque Systems. Acesso em: 07 maio 2025. Citado na página 47.

SVELTE. 2025. Acessado em: 11 de abril de 2025. Disponível em: <<https://svelte.dev/docs>>. Citado na página 42.

- SVELTEKIT. 2024. Acessado em: 10 de abril de 2025. Disponível em: <<https://kit.svelte.dev/>>. Citado na página 43.
- TWITTER. 2015. Acessado em: 14 de fevereiro de 2025. Disponível em: <<https://www.industrialempathy.com/posts/tradeoffs-in-server-side-and-client-side-rendering/>>. Citado na página 16.
- v0. *v0 Documentation*. 2025. Acessado em 14 de julho de 2025. Disponível em: <<https://v0.dev/docs/introduction>>. Citado 2 vezes nas páginas 47 e 48.
- VIANA, J. Fundamentos da web. *Guia Web*, 2024. Disponível em: <<https://jesielviana.gitbook.io/guiaweb/2.-fundamentos-da-web>>. Citado 2 vezes nas páginas 19 e 20.
- VUE. 2025. Acessado em: 11 de abril de 2025. Disponível em: <<https://vuejs.org/guide/introduction.html>>. Citado na página 42.
- WAGNER, J. L. *Web Performance in Action*. Manning Publications, 2016. ISBN 9781617293771. Disponível em: <https://www.manning.com/books/web-performance-in-action?a_aid=webopt&a_bid=63c31090>. Citado 3 vezes nas páginas 15, 16 e 45.
- WATTS, S. *Server-Side Rendering: Benefiting UX and SEO*. 2023. Acessado em: 07 maio 2025. Disponível em: <https://www.splunk.com/en_us/blog/learn/server-side-rendering-ssr.html>. Citado na página 46.
- Web Absoluta. 4 Ferramentas para medir e monitorar a otimização do Core Web Vitals. 2024. Acessado em: 30 set. 2025. Disponível em: <<https://webabsoluta.com.br/seo-otimizacao-sites/4-ferramentas-para-medir-e-monitorar-a-otimizacao-do-core-web-vitals/>>. Citado na página 39.
- Wikipedia. *Protocolo de Transferência de Hipertexto*. 2024. Acessado em 12 de abril de 2025. Disponível em: <https://pt.wikipedia.org/wiki/Hypertext_Transfer_Protocol>. Citado 2 vezes nas páginas 21 e 23.
- Wikipedia. *Man-in-the-middle attack*. 2025. Acessado em 13 de abril de 2025. Disponível em: <https://en.wikipedia.org/wiki/Man-in-the-middle_attack>. Citado na página 23.
- ZHOU, T. et al. An integrated framework of user experience-oriented smart service requirement analysis for smart product service system development. *Advanced Engineering Informatics*, 2022. Disponível em: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-85120077519&doi=10.1016%2fj.aei.2021.101458&partnerID=40&md5=97385fab7168f6e32650452fde90c847>>. Citado 2 vezes nas páginas 60 e 61.