UNIVERSIDADE PAULISTA – UNIP

Ciência da computação

Victor Hugo de Carvalho Finotto

Luis Mario Carvalho Caldeira

ANÁLISE COMPARATIVA ENTRE BLAZOR WEBASSEMBLY E JAVASCRIPT TRADICIONAL NO DESENVOLVIMENTO WEB

São José do Rio Preto - SP

2025

UNIVERSIDADE PAULISTA – UNIP

Ciência da computação

Victor Hugo de Carvalho Finotto

Luis Mario Carvalho Caldeira

ANÁLISE COMPARATIVA ENTRE BLAZOR WEBASSEMBLY E JAVASCRIPT TRADICIONAL NO DESENVOLVIMENTO WEB

Trabalho de Conclusão de Curso apresentado ao curso de Ciência da Computação da Universidade Paulista – UNIP, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação, sob orientação do Prof. Me. Fabio Renato de Almeida

São José do Rio Preto - SP

2025

“*Escolher a tecnologia certa não é sobre qual é a mais popular, mas qual* atende melhor ao problema.”

(*Jeff Atwood)*

RESUMO

Na contemporaneidade, o cenário tecnológico tem sido marcado por uma busca incessante de novas tecnologias e ferramentas que atendam às crescentes demandas do desenvolvimento. Nesse contexto, as linguagens de programação, especialmente no desenvolvimento *web*, têm ganhado cada vez mais notoriedade e, com isso, instaurado a busca por soluções que equilibrem performance e eficiência. O JavaScript, consolidado como linguagem predominante no desenvolvimento *web*, continua sendo amplamente utilizado devido à sua versatilidade e vasto ecossistema, compatível com diversas tecnologias e plataformas. No entanto, com o advento de novas abordagens, como o Blazor WebAssembly, surge uma nova perspectiva na web para os desenvolvedores. Levando isso em consideração, este estudo tem como objetivo avaliar e comparar o JavaScript e o Blazor WebAssembly, em termos de desempenho e eficiência. Esse estudo oferece uma visão abrangente das capacidades e limitações de cada linguagem, facilitando a escolha mais apropriada para cada contexto no desenvolvimento *web*, especialmente no que se refere às necessidades de performance, tempo de resposta e consumo de recursos.

Palavras-chave: *Blazor WebAssembly*; *JavaScript*; desempenho; consumo de recursos; desenvolvimento *web*.

ABSTRACT

In contemporary times, the technological landscape has been marked by a relentless pursuit of new technologies and tools capable of meeting the growing demands of software development. In this context, programming languages—particularly in web development—have gained increasing prominence, leading to the search for solutions that balance performance and efficiency. JavaScript, established as the predominant language in web development, remains widely used due to its versatility and extensive ecosystem, which is compatible with various technologies and platforms. However, with the emergence of new approaches such as Blazor WebAssembly, a new perspective arises for web developers. Considering this scenario, this study aims to evaluate and compare JavaScript and Blazor WebAssembly in terms of performance and efficiency. This research provides a comprehensive view of the capabilities and limitations of each language, assisting in the decision-making process regarding the most appropriate choice for different web development contexts, especially with respect to performance requirements, response time, and resource consumption.

Keywords: Blazor WebAssembly; JavaScript; performance; resource consumption; web development.

SUMÁRIO

1. INTRODUÇÃO

A constante evolução do cenário tecnológico, especialmente na área de desenvolvimento *web*, tem resultado na busca frequente de novas tecnologias e linguagens de programação com o objetivo de suprir a demanda por soluções mais ágeis, eficientes e performáticas. Em meio à vasta gama de opções, as decisões sobre quais tecnologias adotar se tornaram um fator determinante para assegurar a qualidade da experiência do usuário, a escalabilidade e eficiência dos projetos.

Reconhecido como o padrão para aplicações no lado do cliente (*client-side*), o JavaScript se firmou, ao longo do tempo, beneficiando-se de um vasto ecossistema, ampla aceitação na comunidade e integração com inúmeras ferramentas e tecnologias. No entanto, o surgimento do WebAssembly, um formato binário para execução de código no navegador, introduziu novas possibilidades no desenvolvimento *web*. Entre as tecnologias que exploraram esse recurso desponta o Blazor WebAssembly (*client-side*), proposta da Microsoft que permite desenvolver aplicações web interativas utilizando a linguagem de programação C# diretamente no navegador, sem a necessidade de JavaScript. Esta abordagem tem despertado o interesse dos desenvolvedores, por unir a familiaridade da linguagem com a promessa de alto desempenho em aplicações *web*.

Considerando esse contexto, esse estudo propõe uma análise comparativa entre o JavaScript e o Blazor WebAssembly visando os aspectos relacionados ao desempenho, performance, tempo de resposta e consumo de recursos. Dessa forma, pretende-se contribuir para uma tomada de decisão mais apropriada para diferentes contextos no desenvolvimento *web*.

1.1 Objetivo geral

O presente estudo tem como objetivo promover uma comparação entre JavaScript e Blazor WebAssembly no contexto do desenvolvimento *web client-side*, considerando fatores como desempenho, tempo de resposta e consumo de recursos.

Por meio dessa proposta, pretende-se auxiliar desenvolvedores nas tomadas de decisões quanto à adoção da tecnologia mais apropriada pra cada contexto.

1.2 Objetivos específicos

Os objetivos específicos desse estudo são:

a) Desenvolver uma aplicação web em JavaScript e Blazor WebAssembly, a partir de um mesmo modelo base, visando comparar o desempenho em tarefas específicas como carregamento inicial, tempo de resposta e eventos da interface (filtros dinâmicos, ordenações e exclusões em massa de elementos da interface).
b) Utilizar ferramentas de análise de performance dos navegadores (*DevTools*) para coletar métricas sobre o uso de memória, tempo de renderização e armazenamento local, comparando o comportamento de ambas as tecnologias durante a execução do sistema.
c) Avaliar o tamanho total dos arquivos transferidos ao carregar a aplicação (*payload*), considerando seu impacto direto no tempo de carregamento, renderização inicial e experiência do usuário.
d) Analisar a eficiência do processamento local de dados (*client-side*), incluindo filtros, paginação e renderização dinâmica, observando o tempo de execução para manipulações em grandes volumes de dados.
e) Identificar os contextos mais adequados para o uso de cada tecnologia, apontando os principais pontos fortes e limitações.

1.3 Justificativa

A escolha deste tema se justifica pelos seguintes fatores:

a) Evolução das tecnologias *web*: O surgimento de novas abordagens para o desenvolvimento *client-side*, como o WebAssembly, tem desafiado o domínio tradicional do *JavaScript*, ganhando cada vez mais espaço entre os desenvolvedores. Esse cenário, demanda estudos comparativos que avaliem o real potencial dessas tecnologias emergentes.
b) Falta de estudos práticos comparativos: O Blazor WebAssembly, embora promissor e cada vez mais abordado no desenvolvimento *web*, ainda carece

de análises detalhadas. Avaliar seu desempenho por meio de métricas como tempo de carregamento, uso de memória e tamanho de carga (*payload*), em comparação direta com o JavaScript, é essencial para compreender os limites e vantagens de cada tecnologia.

c) Contribuição para a comunidade técnica: Os resultados desta análise se propõem a serem um recurso útil para a escolha consciente de tecnologias *web* em projetos modernos, que pode oferecer suporte prático à tomada de decisões na comunidade de desenvolvedores, especialmente no contexto de aplicações que envolvem manipulação de um grande volume de elementos no lado do cliente.

1.4 Metodologia

Para a realização deste estudo comparativo entre JavaScript e Blazor WebAssembly, será adotada uma abordagem experimental e quantitativa, baseada no desenvolvimento e análise de duas aplicações *web* com funcionalidades idênticas implementadas em ambas as tecnologias. As principais etapas abordadas na metodologia serão:

1.4.1 Levantamento de requisitos

- **Definição dos objetivos funcionais:** Identificar as principais funcionalidades que deverão ser implementadas nas aplicações, como carregamento e manipulação de dados em formato JSON.
- **Requisitos funcionais:** As duas versões da aplicação devem implementar, de forma idêntica como leitura e carregamento de um arquivo JSON com grande volume de dados, filtros dinâmicos com base em critérios variados, paginação e ordenação de registros e exclusão em massa de elementos da interface.
- **Requisitos não funcionais:** Tempo de carregamento inicial (*first load*), tempo de resposta nas interações do usuário, consumo de memória e armazenamento local, tamanho do *payload* transferido e eficiência na renderização e atualização dinâmica da interface.

1.4.2 Desenvolvimento

- **Estruturação dos projetos**: Serão criadas duas aplicações *web* com estrutura e funcionalidades idênticas, uma implementada com JavaScript e outra com Blazor WebAssembly (C#), ambas utilizando HTML e CSS.
- **Funcionalidades implementadas**: As aplicações terão como base a manipulação de um grande volume de dados em formato JSON, realizando operações como carregamento inicial, filtros dinâmicos, paginação e ordenação e exclusão em massa de elementos.

1.4.3 Avaliação das ferramentas e ambiente de teste

- **Padronização do ambiente**: A execução das aplicações será realizada em um ambiente controlado, utilizando o mesmo navegador, com cache limpo, e máquina, garantindo isonomia nos testes e resultados.
- **Ferramentas de medição**: Serão utilizadas ferramentas de desenvolvedor do navegador (*DevTools*) para monitoramento da aplicação durante o tempo de execução.

1.4.4 Análise e interpretação dos resultados

- **Organização dos dados**: Os dados coletados serão organizados em tabelas e gráficos comparativos, permitindo uma análise objetiva entre as tecnologias, destacando os pontos fortes e limitações de cada uma nas operações propostas.
- **Discussão dos resultados**: Os dados obtidos serão interpretados no contexto de desenvolvimento *web* moderno, permitindo identificar os cenários mais adequados para o uso de cada tecnologia.

2. FUNDAMENTAÇÃO TEÓRICA E LEVANTAMENTO BIBLIOGRÁFICO

Nesse capítulo, serão abordados conceitos essenciais que embasam os objetivos estabelecidos neste trabalho. Serão abordadas a evolução do desenvolvimento *web* e sua influência nas aplicações modernas, as principais arquiteturas de sistemas *web, client-side* e *server-side*, bem como o conceito e funcionamento do WebAssembly. Além disso, também serão exploradas as características técnicas, vantagens, limitações e particularidades das tecnologias JavaScript e Blazor WebAssembly.

2.1 Evolução do desenvolvimento web

2.1.1 A Web 1.0 e os primeiros navegadores

Com o surgimento e o crescimento exponencial da internet, a vida cotidiana tornou-se cada vez mais integrada ao ambiente digital. Esse avanço trouxe consigo a necessidade de novas formas de organizar, acessar e interagir com grandes volumes de dados e informações disponíveis. Para responder a essa demanda, a arquitetura da *Web* tem sido continuamente reinventada e aprimorada (DRAPER, 2022).

A primeira etapa dessa evolução teve início em 1991 com a proposta do cientista britânico Timothy John Berners-Lee, que desenvolveu o primeiro navegador *web* e idealizou um sistema de hipertexto distribuído, conhecida como a primeira geração *Web* ou *Web* 1.0, conforme relato de Berners-Lee (1991), citado por Gan et al. (2022). Ainda segundo os autores, as páginas *web* dessa época eram essencialmente estáticas, compostas por documentos HTML simples, com pouca ou nenhuma interatividade. Os *sites* serviam principalmente como repositórios de informação, com navegação limitada a *links* e menus básicos.

2.1.2 A transição para a Web 2.0

Contudo, essa estrutura começou a evoluir ao longo da década de 1990, impulsionado pela necessidade de tornar a *web* mais expressiva e interativa (OLIVEIRA, 2024). Segundo Smith (2024), em 1995, a empresa Netscape introduziu

o *JavaScript* em seu navegador, reconhecendo a importância de uma linguagem de *script* capaz de dinamizar a interação nas páginas *web*. Inicialmente chamada de *LiveScript*, a linguagem foi renomeada para *JavaScript* e, desde então, se consolidou como o principal mecanismo de *scripts* no lado do cliente. Paralelamente, em 1996, foi publicada a primeira especificação oficial das folhas de estilo em cascata (CSS1), proposta por Håkon Wium Lie e desenvolvida em colaboração com Bert Bos, com o objetivo de separar conteúdo e apresentação, melhorando o controle visual das páginas (VOGEL et al., 2022).

Esses avanços técnicos, especialmente das tecnologias – HTML*,* CSS *e JavaScript* - prepararam o terreno para a transição à *Web* 2.0, que se consolidou a partir de 2004. O principal marco dessa nova era foi o empoderamento dos usuários, que passaram a ser produtores de conteúdo, não apenas consumidores, transformando a *web* de um simples repositório informacional em um espaço colaborativo e interativo. O uso de tecnologias como AJAX, *JavaScript*, e folhas de estilo avançadas permitiu a criação de sites dinâmicos e ricos em funcionalidades.

2.1.3 O surgimento das Single Page Applications (SPAs)

Esse cenário impulsionou o surgimento de um novo modelo de aplicação: as *Single Page Applications (SPAs),* em que toda a lógica de navegação e atualização da interface ocorre no lado do cliente, sem recarregamento da página. Essa mudança de paradigma intensificou a demanda por performance *client-side*, especialmente em contextos com interfaces complexas e com grande volume de dados manipulados diretamente pelo navegador (SMITH, 2024).

2.1.4 Web 3.0, descentralização e novas demandas

A partir dessa evolução, surgiu o conceito de *Web* 3.0, como uma resposta à necessidade de dar mais controle aos usuários sobre seus próprios dados. Essa fase busca tornar a *web* mais inteligente, personalizada e conectada, com o uso de tecnologias como inteligência artificial, aprendizado de máquina e *blockchain*, conforme apontam Shaikh, Siddiki e Pathak (2025). Neste mesmo cenário, ganha destaque a chamada *Web3*, que vai além da ideia de personalização e propõe uma

internet verdadeiramente descentralizada, baseada em contratos inteligentes, criptomoedas e outras soluções distribuídas (DRAPER, 2020).

2.1.5 A busca por desempenho no cliente-side e o surgimento do WebAssembly

Com essa transformação, as aplicações *web* passaram a exigir cada vez mais das tecnologias *client-side*. Interfaces ricas, interações em tempo real e aplicações que rodam diretamente no navegador se tornaram comuns. Isso aumentou a cobrança por performance do lado do cliente - o que abriu espaço para novas soluções além do *JavaScript* tradicional. Nesse contexto, surgem tecnologias como o *WebAssembly*, que permitem executar código compilado no navegador com alto desempenho, como, por exemplo, o *Blazor WebAssembly*, trazendo a possibilidade de usar C# e .NET diretamente no *front-end*, com todas as vantagens de performance e estrutura que isso pode representar (SMITH, 2024).

2.2 Arquiteturas de Aplicações web

Neste tópico, serão abordadas as principais arquiteturas utilizadas no desenvolvimento de aplicações *web*. Serão discutidas as arquiteturas *Server-Side* e *Client-Side*, ressaltando suas características, vantagens e desvantagens, além de um comparativo entre elas para ilustrar suas diferentes aplicações no contexto moderno da *web*.

2.2.1 Arquitetura server-side

A arquitetura *Server-Side* é um modelo tradicional no desenvolvimento de aplicações *web*, onde o servidor é responsável por processar as requisições do cliente, executar a lógica de negócios e retornar os resultados para o navegador, conforme apontam Damke e Gregorini (2022). Neste modelo, ainda segundo os autores, as páginas *web* são geradas no servidor e enviadas ao cliente em forma de HTML, CSS e JavaScript, por exemplo, sendo renderizadas diretamente no navegador do usuário. O servidor, nesse contexto, gerencia as operações de manipulação de dados e interação com bancos de dados.

Segundo Iskandar et al. (2020), a principal vantagem da arquitetura *Server-Side* está em centralizar todo o processamento. Dessa forma, como o código que executa a lógica da aplicação permanece no servidor, os dados do sistema estão mais protegidos e a manutenção do código torna-se mais simples de ser realizada. Além disso, o modelo *Server-Side* dispõe de um excelente controle sobre o fluxo de dados e as interações do usuário, uma vez que a maior parte do processamento ocorre no servidor, garantindo uma diminuição considerável na complexidade e risco de inconsistências.

Porém, as desvantagens desse modelo incluem uma dependência maior de uma boa infraestrutura de servidores, que devem ser projetados e dimensionados para garantir o desempenho em ambientes de grande escala, com um fluxo maior de interações. Ademais, como cada interação do usuário requer uma nova comunicação com o servidor, o tempo de resposta e latência podem ser afetados diretamente, impactando a experiência dos usuários, apontam Damke e Gregorini (2022).

2.2.2 Arquitetura client-side

Em contraste com a arquitetura *Server-Side*, a arquitetura *Client-Side* transfere a maior parte da responsabilidade de processamento para o lado do cliente, ou seja, para o navegador do usuário. Nesse modelo, a lógica de aplicação é executada diretamente no cliente, utilizando tecnologias como JavaScript e frameworks como React, Angular, Vue.js, entre outros (SMITH, 2024).

A principal vantagem da arquitetura *Client-Side* é a agilidade no processamento das interações do usuário (FRAJTÁK et al., 2014 apud NEVES et al., 2023). Segundo Damke e Gregorini (2022), as aplicações podem responder de forma muito mais performática, sem a necessidade de estabelecer uma constante comunicação com o servidor para cada interação, uma vez que o código é executado diretamente no navegador. Dessa forma, ainda em consonância com a ideia dos autores, isso resulta em uma experiência mais fluida e dinâmica para o usuário. Além disso, ao reduzir a carga sobre o servidor, a arquitetura *Client-Side* pode ser mais escalável em cenários de grande volume de acessos.

Por outro lado, essa abordagem exige que o navegador do cliente tenha capacidade de processar a lógica de negócios e renderizar a interface de forma

eficiente (NEVES et al., 2023). Além disso, a segurança se torna um desafio, uma vez que o código é enviado para o cliente e pode ser acessado e manipulado, o que requer cuidados especiais no desenvolvimento para evitar vulnerabilidades.

2.2.3 Comparativo entre server-side e client-side

Ao comparar as arquiteturas *Server-Side Rendering* (SSR) e *Client-Side Rendering* (CSR), é possível identificar diferenças cruciais em termos de desempenho, segurança, escalabilidade e complexidade de desenvolvimento. Ambas as abordagens têm suas vantagens e limitações, e a escolha entre elas depende dos requisitos específicos do projeto.

A arquitetura *Server-Side* tende a ser mais segura, já que o código e os dados sensíveis permanecem no servidor, minimizando os riscos de exposição ao usuário. Conforme Iskandar et al. (2020), o servidor processa as requisições do usuário, executa a lógica de negócio e envia ao cliente uma página completamente renderizada, o que contribui para a proteção dos dados e da lógica interna da aplicação. No entanto, essa abordagem pode ser mais lenta em termos de desempenho, pois cada interação do usuário requer uma nova requisição ao servidor, o que pode resultar em maior latência e tempo de resposta. Além disso, seguindo a ideia dos autores, a escalabilidade em sistemas *Server-Side* depende fortemente da infraestrutura do servidor.

Em contrapartida, a arquitetura *Client-Side* se destaca pela rapidez nas interações, uma vez que o processamento é feito no próprio navegador. Isso pode melhorar a experiência do usuário, especialmente em aplicações interativas e dinâmicas. No entanto, a segurança e a compatibilidade entre navegadores tornam-se desafios a serem superados (NEVES et al., 2023). Além disso, a complexidade do desenvolvimento pode aumentar, pois o desenvolvedor precisa garantir que o código seja eficiente e funcione corretamente em diferentes dispositivos e navegadores.

A escolha entre uma arquitetura *Server-Side* ou *Client-Side* deve ser feita com base nas necessidades do projeto, levando em consideração o tipo de aplicação, os requisitos de desempenho, a segurança e a escalabilidade desejada. Esta análise comparativa é essencial para compreender as decisões de arquitetura que influenciam

o desenvolvimento web moderno, especialmente em projetos que utilizam Blazor WebAssembly e JavaScript tradicional, por exemplo.

2.3 WebAssembly

Com o avanço das arquiteturas web e a popularização do modelo Client-Side, surge o WebAssembly (WASM) como uma tecnologia inovadora que amplia as possibilidades do desenvolvimento web ao permitir a execução de código compilado com alta performance em ambiente de navegador. Nesse contexto, com o objetivo de superar as limitações do JavaScript tradicional em aplicações mais complexas, o WebAssembly busca oferecer uma alternativa mais eficiente para cenários que demandam maior desempenho. Além disso, essa tecnologia conta com suporte nativo nos principais navegadores modernos, como Google Chrome, Microsoft Edge, Mozilla Firefox e Apple Safari, o que reforça sua viabilidade no ecossistema *web* atual.

2.3.1 Conceito e funcionamento do WebAssembly

Segundo a documentação oficial do WebAssembly (WEBASSEMBLY, 2025), o WebAssembly (WASM) é composto a partir de um formato binário portátil, projetado para execução segura e eficiente em navegadores modernos. Seu nome advém da junção dos termos *Web*, em referência ao ambiente de execução, como os navegadores, e *Assembly*, remetendo à programação de baixo nível próxima ao hardware. O objetivo do WebAssembly é viabilizar a execução de código compilado com desempenho próximo ao nativo diretamente no ambiente web, superando as limitações de performance do JavaScript tradicional (ROSSBERG, 2022).

Ao contrário do JavaScript, que é interpretado ou compilado dinamicamente (*just-in-time*), o WebAssembly é pré-compilado em *bytecode* antes da execução, o que reduz significativamente o tempo de carregamento e o uso de recursos durante o processamento (WANG et al., 2021). Esse *bytecode* é executado em uma *sandbox* chamada *WebAssembly Runtime*, que proporciona isolamento, segurança e portabilidade, o que possibilita sua utilização em diversas plataformas, inclusive fora do navegador, como em servidores, dispositivos embarcados e aplicações móveis (HOFFMAN, 2019, apud HEINRICH et al., 2021).

O WebAssembly suporta a compilação de linguagens como C, C++, Rust, Go e até C#, por meio de frameworks como o Blazor WebAssembly. Isso permite que aplicações escritas em linguagens de alto nível sejam executadas na web com alta performance, sem abrir mão da produtividade e das ferramentas de desenvolvimento modernas (MICROSOFTa, 2024; WEBASSEMBLY, 2025).

Além disso, a tecnologia mantém interoperabilidade total com o JavaScript, o que permite que ambas sejam utilizadas de maneira complementar, somando a flexibilidade do JavaScript à eficiência do WASM (AMMAR et al., 2024).

Para ampliar ainda mais suas capacidades, foi desenvolvida a *WebAssembly System Interface (WASI)*, uma API que define a interação entre aplicações WebAssembly e o sistema operacional. Sendo assim, essa interface fornece acesso a recursos como sistema de arquivos, rede e entrada/saída, ampliando as possibilidades de uso do WebAssembly para além do navegador (HEINRICH et al., 2021).

2.3.2 Vantagens e limitações do WebAssembly

O WebAssembly (WASM) apresenta uma série de vantagens que o destacam no cenário do desenvolvimento web moderno. Uma das principais virtudes dessa tecnologia está no desempenho: por ser um formato binário de baixo nível e pré-compilado, o WASM possibilita a execução de código com performance próxima à de aplicações nativas, representando um avanço significativo em relação ao JavaScript tradicional, especialmente em tarefas que demandam processamento intensivo (WANG et al., 2021).

Ademais, outra vantagem relevante é a possibilidade de reutilização de código escrito em linguagens consolidadas, como C, C++ e C#. Isso permite o uso de bibliotecas desenvolvidas e testadas, acelerando o desenvolvimento e contribuindo para a performance das aplicações (PREMACHANDRA, 2024). No contexto de plataformas como o Blazor WebAssembly, essa característica possibilita a construção de aplicações client-side com a produtividade do ecossistema .NET, sem abandonar práticas tradicionais de desenvolvimento (MICROSOFTa, 2024).

Além disso, o WebAssembly possui interoperabilidade total com o JavaScript, permitindo que ambas as tecnologias coexistam de forma sinérgica. Essa integração

oferece flexibilidade ao desenvolvedor, que pode delegar tarefas computacionalmente custosas ao WASM, enquanto mantém a lógica de interface no JavaScript (AMMAR et al., 2024).

Por outro lado, apesar das vantagens, o WebAssembly apresenta limitações que merecem atenção. Uma das principais vantagens está no seu modelo de execução restrito, que, por razões de segurança, limita o acesso direto à DOM (Document Object Model), exigindo a manipulação da interface via JavaScript. Essa restrição aumenta a complexidade de projetos com alta interação com o usuário (FERREIRA et al., 2023.

Além das limitações de acesso à DOM, o ecossistema de ferramentas para desenvolvimento com WASM ainda está em amadurecimento. Ferramentas de depuração, em especial, enfrentam desafios devido à natureza binária do código, que ainda carecem de sofisticação comparável às de outras plataformas (GILBERT et al., 2025). Essa situação dificulta a identificação e correção de erros em aplicações complexas.

Outra limitação importante é o tamanho dos arquivos gerados. Módulos WASM tendem a ser maiores que scripts JavaScript otimizados, impactando o tempo de carregamento inicial (payload), especialmente em conexões mais lentas. No entanto, técnicas de compactação, como GZIP, podem atenuar esse efeito, melhorando o tempo percebido de carregamento (SILVA; SILVA, 2021).

Segundo Cohen (2020, apud SILVA; SILVA, 2021), o WebAssembly não proporciona ganhos significativos de desempenho em aplicações que fazem uso esporádico da tecnologia ou que não exigem alto processamento, uma vez que o JavaScript moderno já apresenta elevada eficiência nessas situações.

Portanto, embora o WebAssembly represente uma inovação importante no desenvolvimento web, seu uso deve ser avaliado conforme as necessidades específicas do projeto, equilibrando ganhos de desempenho e escalabilidade com os desafios de sua implementação e manutenção.

2.4 Blazor

Como abordado anteriormente, o WebAssembly abriu novas possibilidades para a execução de aplicações no navegador, proporcionando performance nativa

sem depender exclusivamente de JavaScript. Nesse novo cenário de execução *client-side*, uma das tecnologias que mais se destaca é o Blazor, tecnologia desenvolvida pela Microsoft que se apoia diretamente no modelo de desenvolvimento de aplicações modernas integralmente em C#. Com o Blazor, torna-se possível implementar interfaces web interativas sem a necessidade de recorrer ao JavaScript, explorando o poder do WebAssembly e a robustez do ecossistema .NET (MEDAVARAPU, 2021).

Segundo a documentação oficial da Microsoft (2024a), o Blazor surge para superar uma limitação histórica enfrentada pelos desenvolvedores .NET: a dissociação entre o desenvolvimento do *back-end*, geralmente escrito em C#, e o *front-end*, tradicionalmente implementado em JavaScript. Ao unificar essas camadas sob uma única linguagem e conjunto de ferramentas, o Blazor contribui significativamente para a redução da complexidade arquitetural, além de promover a melhoria da produtividade no desenvolvimento a partir do reaproveitamento de código entre diferentes partes da aplicação.

Além de seu suporte ao WebAssembly, que será aprofundado nos próximos tópicos, o Blazor oferece modelos alternativos de hospedagem, como a execução via servidor ou em ambientes híbridos. Essa flexibilidade permite sua aplicação em uma ampla gama de cenários, desde soluções *web* tradicionais até aplicações *desktop* e *mobile* integradas, sem abrir mão do ecossistema .NET (MICROSOFT, 2024a). A seguir, será apresentada a estrutura conceitual do Blazor e suas principais variantes de execução.

2.4.1 Conceito e arquitetura geral do Blazor

Blazor é uma tecnologia que permite a construção de interfaces *web* interativas utilizando C# e HTML, baseada na *engine* Razor, uma sintaxe para criar *templates* de páginas *web* que combina código C# com HTML de forma eficiente e legível. O nome “Blazor” é uma junção das palavras “Browser” e “Razor”, indicando o propósito da tecnologia: levar o poder da *engine* Razor, tradicionalmente utilizada no lado servidor, para ser executada diretamente no navegador. O framework é projetado para funcionar com base em componentes, que são unidades modulares compostas por marcação (HTML), lógica (C#) e estilos, facilitando o desenvolvimento organizado e a sua reutilização (MICROSOFT, 2024a).

No Blazor, esses componentes são as peças fundamentais da interface, funcionando como blocos modulares que combinam estilos visuais e comportamento. Cada componente gerencia seu próprio estado e reage de forma única a eventos do usuário, como cliques ou entradas de dados. Para otimizar a atualização da interface, o Blazor mantém uma árvore virtual de renderização, implementada em C#, que atualiza somente os campos da página que sofreram alterações, garantindo uma experiência integrada ao ecossistema .NET (MICROSOFT, 2025a).

Além disso, essa arquitetura flexível do Blazor suporta diferentes modelos de hospedagem, que definem como e onde a aplicação é executada, seja no servidor, no navegador ou em um ambiente híbrido, oferecendo alternativas para diferentes cenários e necessidades de desenvolvimento (MICROSOFT, 2024b).

2.4.2 Modelos de hospedagem do Blazor

O Blazor foi desenvolvido com a finalidade de oferecer uma arquitetura flexível e adaptável a diferentes cenários de execução. Atualmente, a Microsoft disponibiliza três modelos principais de hospedagem: Blazor Server, Blazor WebAssembly e Blazor Hybrid. Sendo assim, embora compartilhem a mesma base de componentes e recursos do framework, cada modelo possui características arquiteturais distintas, que impactam diretamente na forma como a aplicação é executada e utilizada pelo usuário final (MICROSOFT, 2024b).

2.4.2.1 Blazor Server

O Blazor Server foi o primeiro modelo lançado e tem como principal característica o fato de que toda a lógica da aplicação é executada no servidor. A interface exibida no navegador do usuário é renderizada por meio de uma conexão em tempo real baseada no SignalR, uma biblioteca da Microsoft que permite comunicação bidirecional entre cliente e servidor, geralmente baseada em WebSockets. Dessa forma, a cada interação, como um clique ou digitação do usuário, um evento é enviado ao servidor, processado, e as atualizações visuais são enviadas de volta ao navegador (MICROSOFT, 2024b).

Esse modelo tem como principal vantagem o carregamento inicial (*payload*) extremamente leve, já que o navegador carrega apenas instância cliente da biblioteca SignalR e os arquivos básicos da aplicação. No entanto, ele depende de uma conexão persistente com o servidor, o que pode ser um ponto crítico em ambientes com latência elevada ou instabilidade de rede.

2.4.2.2 Blazor WebAssembly

Em contraste, o Blazor WebAssembly transfere a execução da aplicação para o cliente. Neste modelo, o navegador do usuário carrega o *runtime* do .NET compilado em WebAssembly, juntamente com os *assemblies* da aplicação e suas dependências. Isso significa que toda a lógica é processada localmente, no próprio navegador, dispensando a necessidade de comunicação constante com o servidor (MICROSOFT, 2024b).

Todavia, apesar da vantagem de não depender da conexão com o servidor após o carregamento, o Blazor WebAssembly apresenta tempos iniciais de carregamento maiores, especialmente em conexões lentas, devido ao tamanho dos arquivos transferidos. Ainda assim, é uma abordagem bastante promissora para Aplicações de Página Única (*Single Page Applications* - SPA) especialmente com foco em performance do lado do cliente (MEDAVARAPU, 2021).

2.4.2.3 Blazor Hybrid

Por fim, o Blazor Hybrid representa um modelo inovador, voltado para cenários em que a aplicação Blazor é incorporada dentro de um contêiner nativo. Nesse caso, a interface é construída com componentes Blazor renderizados em um WebView, mas a lógica da aplicação é executada nativamente, com acesso completo às APIs da plataforma (MICROSOFT, 2025b). Essa abordagem é ideal para aplicações desktop e mobile que desejam reaproveitar o mesmo código utilizado na web, combinando a familiaridade do desenvolvimento web com os recursos e desempenho de aplicações nativas (MICROSOFT, 2024b).

2.4.2.4 Considerações sobre escolha do modelo

Segundo Medavarapu (2021), a escolha do modelo de hospedagem mais adequado deve ser analisada conforme o cenário de uso, como requisitos de conectividade, volume de usuários simultâneos, suporte a dispositivos e até mesmo o tempo de carregamento inicial. Contudo, para os fins deste estudo, a análise será direcionada ao Blazor WebAssembly, por representar a abordagem mais inovadora do framework, ao permitir a execução de aplicações .NET diretamente no navegador, sem intermediação do servidor.

2.5 JavaScript no desenvolvimento Web

Apesar do surgimento de alternativas como o Blazor WebAssembly, que busca proporcionar alta performance com linguagens compiladas no navegador, o JavaScript permanece como a base do desenvolvimento *web client-side*. Essa permanência se deve à sua ampla adoção, compatibilidade universal com navegadores e à maturidade de seu ecossistema. Sua evolução o tornou não apenas uma linguagem de script para interações básicas, mas um pilar completo para o desenvolvimento de aplicações ricas e responsivas. A seguir, serão abordadas sua trajetória histórica, o ecossistema de ferramentas, arquitetura geral, modelos de execução e seus principais pontos fortes e limitações.

2.5.1 História e evolução da linguagem

‘ O JavaScript surgiu em 1995, criado por Brendan Eich na empresa Netscape, com a proposta inicial de viabilizar páginas web interativas através da manipulação do HTML diretamente no navegador. A linguagem foi primeiramente chamada de LiveScript, mas recebeu o nome definitivo de JavaScript como estratégia de marketing, visando aproveitar o prestígio da linguagem Java na época (FLANAGAN, 2020; MORORÓ, 2024).

A padronização oficial da linguagem ocorreu em 1997, com o lançamento da especificação ECMAScript pela Ecma International, definindo regras formais que permitiram sua adoção ampla pelos navegadores. A linguagem passou por diversas

atualizações, sendo a versão ECMAScript 6 (ES6), lançada em 2015, especialmente relevante, que trouxe melhorias significativas como o uso de classes, *arrow functions*, módulos e o escopo de variáveis com *let* e *const*, aproximando-se de paradigmas da programação orientada a objetos e funcional (FLANAGAN, 2020).

Mais recentemente, com o surgimento da plataforma Node.js, o JavaScript passou a ser utilizado também no *back-end*, possibilitando sua adoção em aplicações *full-stack*. Sendo assim, uma mesma linguagem empregada do lado do cliente e do servidor, promovendo economia de contexto e reutilização de lógica de negócios (RIALLAND; NARDIN, 2022).

2.5.2 Conceito e Arquitetura geral

O JavaScript é uma linguagem interpretada, executado diretamente no navegador por um interpretador em tempo de execução, utilizando o motor de interpretação (*engine*) de cada *browser*, como o V8 (Chrome), SpiderMonkey (Firefox) ou JavaScriptCore (Safari). Além disso, no contexto *client-side*, o JavaScript interage com a árvore DOM (*Document Object Model*), permitindo modificar dinamicamente a estrutura e o conteúdo das páginas HTML (CONTRIBUTORS, 2021).

Dessa forma, com a ascensão dos frameworks como React, Angular e Vue.js, o JavaScript passou a ser estruturado de forma mais robusta, utilizando o conceito de componentes, estados e rotas *client-side* para construção de SPAs (*Single Page Applications*), ampliando significativamente sua escalabilidade.

2.5.3 Pontos fortes do JavaScript

O JavaScript se destaca como a principal linguagem para desenvolvimento *web client-side* devido à sua ubiquidade, sendo suportado nativamente por todos os navegadores modernos, sem necessidade de instalação adicional (CONTRIBUTORS, 2021). Seu ecossistema é amplo e em constante evolução, com milhares de bibliotecas e frameworks acessíveis via NPM e YARN, o que impulsiona a produtividade e acelera o desenvolvimento. Ademais, por ser interpretado, a linguagem permite testes e ajustes rápidos, reduzindo o ciclo de desenvolvimento (RIALLAND; NARDIN, 2022).

A linguagem também conta com uma comunidade ativa, vasta documentação e recursos de aprendizado acessíveis, beneficiando tanto iniciantes quanto desenvolvedores experientes. Sua integração fluida com tecnologias como HTML, CSS, Web APIs e módulos WebAssembly reforça sua versatilidade. Além disso, o suporte à programação assíncrona, por meio de *callbacks, promises e async/await*, garante eficiência em aplicações que demandam paralelismo e comunicação com servidores (MORORÓ, 2024).

2.5.4 Limitações do JavaScript

Apesar de sua ampla adoção e versatilidade, o JavaScript também apresenta limitações importantes que devem ser consideradas no desenvolvimento de aplicações (FLANAGAN, 2020). Por ser uma linguagem interpretada e de tipagem dinâmica, tende a apresentar desempenho inferior em comparação com linguagens compiladas, sobretudo em tarefas computacionalmente intensas, como processamento de imagens ou grandes volumes de dados em tempo real.

Esse desempenho é ainda impactado pelo modelo de execução *single-threaded*, no qual apenas uma tarefa é processada por vez na thread principal. Embora recursos como Web Workers permitam paralelismo, sua implementação é complexa e nem sempre viável. Dessa maneira, em projetos maiores, a ausência de tipagem estática compromete a legibilidade e a manutenção do código, aumentando a incidência de erros difíceis de rastrear e exigindo práticas rigorosas de teste, além da adoção de ferramentas como o TypeScript para mitigar esses riscos (CONTRIBUTORS, 2021).

Por fim, apesar dos avanços na padronização da linguagem, ainda persistem diferenças entre os motores JavaScript dos navegadores, o que pode ocasionar comportamentos inconsistentes e demandar testes adicionais para garantir uma experiência uniforme entre plataformas.

2.5.5 Considerações sobre a escolha do modelo tradicional

Segundo Mororó (2024), a seleção da linguagem ou modelo de desenvolvimento deve considerar fatores como compatibilidade, curva de aprendizado,

requisitos de desempenho e complexidade do projeto. Neste estudo, opta-se pela análise do JavaScript tradicional como base de comparação por sua ampla adoção, execução direta no navegador e papel consolidado no desenvolvimento web. Essa escolha permite avaliar, de forma clara, os contrastes com o modelo Blazor WebAssembly, sobretudo no que diz respeito às vantagens e desvantagens de ambas tecnologias, e ao paradigma de performance e execução *client-side.*

3. RESULTADOS ESPERADOS

A partir da realização dos testes comparativos entre as aplicações desenvolvidas em JavaScript e Blazor WebAssembly, espera-se obter resultados que evidenciem as principais diferenças entre essas tecnologias no desenvolvimento *web client-side*. Os experimentos deverão fornecer dados objetivos sobre desempenho, consumo de recursos e tempo de resposta, permitindo identificar os pontos fortes e limitações de cada abordagem.

É esperado que o JavaScript apresente melhor desempenho no tempo de carregamento inicial, em virtude do tamanho reduzido dos arquivos transferidos e da ausência de um *runtime* adicional, como propõe o Blazor. Essa característica pode favorecer a experiência do usuário em dispositivos com conexões mais lentas ou menor capacidade de processamento.

Por outro lado, o Blazor WebAssembly tende a se destacar na execução de tarefas mais complexas, como filtros dinâmicos, ordenações e exclusões em massa, devido à sua execução compilada em WebAssembly. Espera-se que isso resulte em tempos de resposta mais consistentes e maior eficiência, especialmente em contextos com grandes volumes de informação.

Além disso, é esperado que o Blazor WebAssembly demande maior uso de memória e apresente um *payload* inicial mais elevado, o que pode impactar a performance em ambientes com recursos limitados. Ainda assim, espera-se que a tecnologia demonstre vantagem em projetos que já utilizam o ecossistema .NET e buscam unificação entre *back-end* e *front-end*.

Por fim, espera-se que os resultados contribuam para identificar os cenários mais adequados para o uso de cada tecnologia, fornecendo subsídios técnicos que auxiliem desenvolvedores na tomada de decisões mais alinhadas às necessidades específicas de cada projeto web.

REFERÊNCIAS

ALMEIDA, Pedro H. Marques; PAULA, Werik Gonçalves de. **Análise comparativa das características de performance dos JavaScript frameworks React e Vue**. Belo Horizonte: Instituto de Ciências Exatas e Informática – PUC Minas, 2021. Disponível em: http://bib.pucminas.br:8080/pergamumweb/vinculos/000004/00000433.pdf. Acesso em: 20 maio 2025.

AMMAR, Mohammed; HUSSAIN, Faraz; MAFRA, João; MAKHLOUF, Walid. **WebAssembly and security: A review**. Journal of Web Semantics, v. 84, p. 100776, 2024. Disponível em: https://www.sciencedirect.com/science/article/abs/pii/S157401372500005X. Acesso em: 18 maio 2025.

CONTRIBUTORS, M. Javascript. MDN Web Docs, 2021. Acessado em 20 de maio de 2025. Disponível em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript

DAMKE, Gabriel Trevisan; GREGORINI, Daniel Mahl. **Comparação de Desempenho em Páginas Web: Uma Análise de Client-Side Rendering e Server-Side Rendering**. In: Congresso Latino-Americano de Software Livre e Tecnologias Abertas (Latinoware), 2022. Sociedade Brasileira de Computação. Disponível em: https://sol.sbc.org.br/index.php/latinoware/article/view/31567. Acesso em: 15 maio 2025.

DRAPER, C. **The Rise of Web 3.0: Blockchain, Decentralization, and the Future of the Internet.** Springer, 2020. Disponível em: https://link.springer.com/article/10.1007/s10586-021-03301-8. Acesso em: 14 maio 2025.

DRAPER, C. **The Internet's Evolution: From Web 1.0 to 2.0: The Good, The Bad and The Ugly.** Journal of Digital History, 2022. Disponível em: https://www.researchgate.net/publication/360638266_Evolution_of_the_Internet_from_Web_10_to_Metaverse_The_Good_The_Bad_and_The_Ugly. Acesso em: 14 maio 2025.

FERREIRA, João; SILVA, Maria; PEREIRA, Carlos. **WebAssembly and Security: a review**. 2023. Disponível em: https://www.researchgate.net/publication/382332042_WebAssembly_and_Security_a_review. Acesso em: 19 maio 2025.

MEDAVARAPU, Sai Vaibhav, **Server-Side Rendering vs. Client-Side Rendering in Blazor**, Journal of Scientific and Engineering Research, 2021. Disponível em: https://jsaer.com/download/vol-8-iss-12-2021/JSAER2021-8-12-311-317.pdf. Acesso em: 19 maio 2025.

FLANAGAN, David. **JavaScript: o guia definitivo**. 7. ed. Sebastopol: O'Reilly Media, 2020.

GAN, Wensheng; YE, Zhenqiang; WAN, Shicheng; YU, Philip S. **Web 3.0: The Future of Internet.** In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, 2022. Disponível em: https://arxiv.org/pdf/2304.06032. Acesso em: 8 maio 2025.

GILBERT, Elizabeth et al. **Debugging WebAssembly? Put some Whamm on it!** arXiv preprint arXiv:2504.20192, 2025. Disponível em: https://arxiv.org/abs/2504.20192. Acesso em: 19 maio 2025.

HEINRICH, Tiago; WILL, Newton C.; OBELHEIRO, Rafael R.; MAZIERO, Carlos A. **Uso de chamadas WASI para a identificação de ameaças em aplicações WebAssembly.** In: SIMPÓSIO BRASILEIRO EM SEGURANÇA DA INFORMAÇÃO E DE SISTEMAS COMPUTACIONAIS (SBSeg), 2021. Disponível em: https://sol.sbc.org.br/index.php/sbseg/article/view/27210/27026. Acesso em: 19 maio 2025.

ISKANDAR, Taufan Fadhilah et al. **Comparison between client-side and server-side rendering in the web development.** IOP Conference Series: Materials Science and Engineering, v. 801, p. 012136, 2020. Disponível em: https://iopscience.iop.org/article/10.1088/1757-899X/801/1/012136/pdf. Acesso em: 15 maio 2025

MICROSOFT. **ASP.NET Core Blazor Hybrid**, 2025b. Disponível em: https://learn.microsoft.com/pt-br/aspnet/core/blazor/hybrid/?view=aspnetcore-9.0 Acesso em: 19 maio 2025.

MICROSOFT. **Blazor WebAssembly**, 2024a. Disponível em: https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-7.0. Acesso em: 19 maio 2025.

MICROSOFT. **Conceitos básicos do ASP.NET Core Blazor**, 2025a. Disponível em: https://learn.microsoft.com/pt-br/aspnet/core/blazor/fundamentals/?view=aspnetcore-9.0 Acesso em: 19 maio 2025.

MICROSOFT. **Modelos de hospedagem do Blazor ASP.NET Core**, 2024b. Disponível em: https://learn.microsoft.com/pt-br/aspnet/core/blazor/hosting-models?view=aspnetcore-9.0 Acesso em: 19 maio 2025.

MORORÓ, Jadson Faustino. ***Um estudo comparativo entre JavaScript e TypeScript.*** 2024. 154 f. Trabalho de Conclusão de Curso (Graduação em Engenharia de Computação) – Universidade Federal do Ceará, Campus de Sobral, Sobral, 2024. Disponível em: https://repositorio.ufc.br/bitstream/riufc/78817/1/2024_tcc_jfmororo.pdf

NEVES, Lucas Rocha; CARVALHO JUNIOR, Átila; HORA, Henrique Monteiro Rego da. **Aplicações web client-side baseadas em JavaScript: uma análise bibliométrica**. 2020. Disponível em: https://www.researchgate.net/publication/347949078_APLICACOES_WEB_CLIENT-SIDE_BASEADAS_EM_JAVASCRIPT_UMA_ANALISE_BIBLIOMETETRICA. Acesso em: 18 maio 2025.

OLIVEIRA, Flávio Donizeti de. **Desenvolvimento de sistemas web modernos**. Revista Gestão em Foco, ano 2024, p. 215. Disponível em: http://www.unifia.edu.br. Acesso em: 9 maio 2025.

PREMACHANDRA, Bakh. **WebAssembly in modern web technology: Analysis of benefits vs challenges.** Department of Information Technology, General Sir John Kotelawala Defence University (KDU), Sri Lanka, 2024. Disponível em: https://www.researchgate.net/publication/378901628_WebAssembly_in_modern_web_technology_Analysis_of_benefits_vs_challenges. Acesso em: 19 maio 2025.

RIALLAND, J.; NARDIN, F**. Fullstack JavaScript: do cliente ao servidor com Node.js e React.** Novatec, 2022.

ROSSBERG, A. **WebAssembly specification**. [S.l.]: WebAssembly Community Group, 2022. Disponível em: https://webassembly.github.io/spec/core/index.html. Acesso em: 19 maio 2025.

SCOTT, Adam D. **JavaScript Cookbook: Programming the Web**. 3. ed. Sebastopol: O'Reilly Media, 2021.

SHAIKH, Aftab; SIDDIKI, Shahin; PATHAK, Rashmi. **The Rise of Web 3.0: Decentralization and Its Impact on Web Development.** International Journal of Scientific Research and Engineering Development, v. 8, n. 2, p. 2214–2220, 2025. Disponível em: https://ijsred.com/volume8/issue2/IJSRED-V8I2P351.pdf. Acesso em: 14 maio 2025.

SILVA, D. R.; SILVA, H. L. da L**. Avaliação do desempenho do WebAssembly: um estudo de caso utilizando reconhecimento de dígitos manuscritos**. 2021. Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) – Instituto Federal de Ciência e Tecnologia de Pernambuco, Recife, 2021. Disponível em: https://repositorio.ifpe.edu.br/xmlui/bitstream/handle/123456789/981/Avaliação%20do%20desempenho%20do%20Webassembly%20.pdf. Acesso em: 19 maio 2025.

Smith, J**. Web Evolution to Revolution: Navigating the Future of Web Application Development**, 2024. Disponível em: https://www.researchgate.net/publication/378893182_Web_Evolution_to_Revolution_Navigating_the_Future_of_Web_Application_Development. Acesso em: 14 maio 2025.

VOGEL, Lucas; SPRINGER, Thomas; WÄHLISCH, Matthias. **Files to Streams: Revisiting Web History and Exploring Potentials for Future Prospects**. TU Dresden, 2022. Disponível em: https://arxiv.org/abs/2303.14184. Acesso em: 8 maio 2025.

WEBASSEMBLY. **What is WebAssembly?** Disponível em: https://webassembly.org/. Acesso em: 18 maio 2025.

YAN, Yutian; TU, Tengfei; ZHAO, Lijian; ZHOU, Yuchen; WANG, Weihang. **Understanding the Performance of WebAssembly Applications**. Proceedings of the ACM Internet Measurement Conference, 2021. Disponível em: https://weihang-wang.github.io/papers/imc21.pdf. Acesso em: 18 maio 2025.