UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

LUIZ ANTONIO SCHONS

ANÁLISE COMPARATIVA DE ARQUITETURAS DE SOFTWARE PARA APLICAÇÕES WEB: DESEMPENHO, ESCALABILIDADE E EXPERIÊNCIA DE DESENVOLVIMENTO

GUARAPUAVA

2025

LUIZ ANTONIO SCHONS

ANÁLISE COMPARATIVA DE ARQUITETURAS DE SOFTWARE PARA APLICAÇÕES WEB: DESEMPENHO, ESCALABILIDADE E EXPERIÊNCIA DE DESENVOLVIMENTO

Comparative Analysis of Software Architectures for Web Applications: Performance, Scalability and Development Experience

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Sistemas para Internet do Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Andres Jesse Porfirio

GUARAPUAVA

2025

4.0 Internacional

Esta licença permite compartilhamento, remixe, adaptação e criação a partir do trabalho, mesmo para fins comerciais, desde que sejam atribuídos créditos ao(s) autor(es). Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

LUIZ ANTONIO SCHONS

ANÁLISE COMPARATIVA DE ARQUITETURAS DE SOFTWARE PARA APLICAÇÕES WEB: DESEMPENHO, ESCALABILIDADE E EXPERIÊNCIA DE DESENVOLVIMENTO

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Tecnólogo em Sistemas para Internet do Tecnologia em Sistemas para Internet da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 12/junho/2025

Andres Jessé Porfirio
Doutor
Universidade Tecnológica Federal do Paraná - Campus Guarapuava

---

Dênis Lucas Silva
Mestre
Universidade Tecnológica Federal do Paraná - Campus Guarapuava

---

Sediane Carmem Lunardi Hernandes
Doutora
Universidade Tecnológica Federal do Paraná - Campus Guarapuava

---

Peterson Medeiros
Especialista

GUARAPUAVA

2025

RESUMO

Este trabalho tem como objetivo comparar três arquiteturas de software — monolítica, cliente-servidor e serverless — aplicadas ao mesmo sistema real: o Open Social Care, uma aplicação voltada à gestão de ações sociais. A escolha das arquiteturas justifica-se pela necessidade de compreender, na prática, as implicações técnicas e operacionais de cada modelo no desenvolvimento e implantação de sistemas web. A metodologia adotada envolveu a implantação do sistema em cada uma das arquiteturas, seguido da realização de experimentos em ambientes controlados. Foram analisados aspectos como desempenho, consumo de recursos, tempo de resposta, experiência do desenvolvedor, facilidade de implantação e custos de infraestrutura. Os resultados indicaram que a arquitetura monolítica apresenta maior simplicidade e menor custo operacional, sendo recomendada para MVPs e projetos de menor porte, podendo também atender aplicações maiores por meio de escalabilidade vertical. A arquitetura cliente-servidor, embora exija maior esforço de configuração e manutenção, destaca-se pela modularidade e organização, sendo mais indicada para sistemas em expansão. A arquitetura serverless apresentou os melhores tempos de resposta e menor esforço de implantação, além de escalabilidade automática, tornando-se adequada para aplicações com variação de demanda. No entanto, seus custos podem se tornar desvantajosos à medida que a aplicação cresce, exigindo monitoramento constante. A análise crítica evidenciou que não existe uma arquitetura universalmente superior, mas sim diferentes soluções mais apropriadas a depender do contexto, dos recursos da equipe e da natureza do sistema. A conclusão do estudo aponta a arquitetura serverless como a mais vantajosa no cenário atual da aplicação analisada, com a recomendação de migração futura para cliente-servidor, caso o crescimento do projeto assim exigir. Este trabalho contribui para o entendimento prático das decisões arquiteturais e suas implicações no ciclo de vida de sistemas web modernos.

Palavras-chave: arquitetura de software; sistemas web; serverless; cliente-servidor; monolítica.

ABSTRACT

This work aims to compare three software architectures — monolithic, client-server, and serverless — applied to the same real system: Open Social Care, an application focused on managing social initiatives. The choice of architectures is justified by the need to understand, in practice, the technical and operational implications of each model in the development and deployment of web systems. The adopted methodology involved fully implementation the system in each of the architectures, followed by experiments in controlled environments. Aspects such as performance, resource consumption, response time, developer experience, ease of deployment, and infrastructure costs were analyzed. The results indicated that the monolithic architecture offers greater simplicity and lower operational cost, being recommended for MVPs and small-scale projects, and may also support larger applications through vertical scalability. The client-server architecture, although requiring greater configuration and maintenance effort, stands out for its modularity and organization, making it suitable for expanding systems. The serverless architecture showed the best response times and the least deployment effort, along with automatic scalability, making it suitable for applications with fluctuating demand. However, its costs can become disadvantageous as the application grows, requiring constant monitoring. The critical analysis highlighted that there is no universally superior architecture, but rather different solutions that are more appropriate depending on the context, team resources, and system characteristics. The study concludes that serverless architecture is currently the most advantageous for the analyzed application, with a recommendation for future migration to client-server architecture if project growth justifies it. This work contributes to the practical understanding of architectural decisions and their impact on the life cycle of modern web systems.

Keywords: software architecture; web system; serverless; client-server; monolithic.

SUMÁRIO

1 INTRODUÇÃO 6

1.1 Considerações iniciais 6

1.2 Objetivos 7

1.2.1 Objetivo específicos 7

1.3 Justificativa 7

2 TRABALHOS RELACIONADOS 9

3 FUNDAMENTAÇÃO TEÓRICA 10

3.1 Divisão de camadas 10

3.1.1 Frontend 10

3.1.2 Backend 10

3.2 Arquiteturas de Software 11

3.2.1 Monolítica 11

3.2.2 Cliente-Servidor 12

3.2.3 Serverless 12

3.3 Experiência de Desenvolvimento 13

3.4 Análise de Custo e Testes de Performance 13

4 METODOLOGIA 15

5 EXPERIMENTOS 17

5.1 Cenário Experimental 18

5.2 Experimento 1: Uso de Recursos do Sistema 18

5.3 Experimento 2: Tempo de Resposta 20

5.4 Experimento 3: Tempo de Carregamento de Páginas 21

5.5 Experimento 4: Tempo de Build 23

5.6 Experimento 5: Velocidade de Execução da Pipeline de CI/CD 25

5.7 Experimento 6: Tempo e Facilidade de Deploy 26

5.8 Experimento 7: Custos de Infraestrutura 28

5.9 Discussão dos Resultados 29

5.9.1 Uso de Recursos do Sistema 29

5.9.2 Tempo de Resposta 30

5.9.3 Tempo de Carregamento de Páginas 30

5.9.4 Tempo de Build e Pipeline de CI/CD 30
5.9.5 Tempo e Facilidade de Deploy 30
5.9.6 Análise Crítica dos Custos com Infraestrutura 31
5.9.7 Considerações Gerais 32
6 CONCLUSÃO 33
REFERÊNCIAS 35

1 INTRODUÇÃO

1.1 Considerações iniciais

Desenvolver software é uma tarefa complexa e desafiadora e, nos dias atuais, com a evolução da tecnologia e a crescente demanda por aplicações web, a escolha da arquitetura de software é um fator crítico para o sucesso de um projeto. A arquitetura de software é um conjunto de padrões e práticas que definem a estrutura de um sistema de software, incluindo os componentes, as relações entre eles e as diretrizes para o seu desenvolvimento e evolução. Existem várias arquiteturas de software disponíveis para aplicações web, cada uma com suas próprias características, vantagens e desvantagens e muitas vezes é responsabilidade do arquiteto de software escolher a arquitetura mais adequada para um determinado projeto.

Fazer a escolha certa da arquitetura de software pode ter um impacto significativo no desempenho e escalabilidade de uma aplicação web, bem como na experiência de desenvolvimento dos desenvolvedores. Uma arquitetura bem projetada pode facilitar a manutenção, a evolução e a escalabilidade de um sistema de software, enquanto uma arquitetura mal projetada pode resultar em problemas de desempenho, escalabilidade e manutenção (FORD *et al.*, 2021).

A experiência de desenvolvimento é medida pela facilidade com que os desenvolvedores podem entender, modificar e estender o código de um sistema de software. Isso está atrelado a várias pontos que são mensuráveis, como o tempo de build (tempo que o sistema leva para compilar), a velocidade de execução da pipeline de CI/CD [1], e o tempo e facilidade de deploy [2].

Outro fator importante a ser considerado é a escalabilidade de uma aplicação web, que se refere à capacidade de um sistema de software lidar com um aumento na carga de trabalho, como o número de usuários, transações ou dados. Uma arquitetura escalável é aquela que pode crescer e se adaptar às demandas de um sistema de software em evolução, sem comprometer o desempenho ou a disponibilidade.

Do ponto de vista do usuário, o mais crucial é que a escolha de uma arquitetura de software deve ser baseada em performance, que é medida de diversas formas, como tempo de resposta, tempo de carregamento de páginas, uso de recursos do sistema, entre outros. A performance de uma aplicação web é um fator crítico para a satisfação do usuário e o sucesso de um projeto, e uma arquitetura de software bem projetada pode contribuir significativamente para a performance de uma aplicação web.

---

[1] CI/CD cria um pipeline automatizado, desde o código-fonte até a produção, permitindo que as equipes entreguem software com mais rapidez, frequência e confiabilidade, saiba mais em https://martinfowler.com/articles/continuousIntegration.html e https://martinfowler.com/books/continuousDelivery.html

[2] Deploy é o processo de disponibilizar um software ou aplicação para uso, movendo-o de um ambiente de desenvolvimento para um ambiente de produção acessível aos usuários., saiba mais em https://www.ibm.com/docs/en/zos/3.1.0?topic=task-deploying-software

Diante disso, a escolha da arquitetura de software certa para uma aplicação web não é uma tarefa fácil, pois envolve a consideração de vários fatores, como os requisitos do projeto, as restrições de tempo e orçamento, as habilidades da equipe de desenvolvimento e as tendências tecnológicas (RICHARDS; FORD, 2020). Além disso, as arquiteturas de software estão em constante evolução, com novas abordagens e tecnologias sendo desenvolvidas regularmente, o que torna ainda mais desafiador escolher a arquitetura certa para um projeto.

A fim de ajudar os arquitetos de software a tomar decisões informadas sobre a escolha da arquitetura de software mais adequada para seus projetos, neste trabalho é apresentado analises e comparações de arquiteturas de software para aplicações web, incluindo a arquitetura monolítica, a arquitetura cliente-servidor e a arquitetura serverless, com foco no desempenho e escalabilidade e na experiência de desenvolvimento dos desenvolvedores.

A análise é feita com base em um estudo de caso, onde foi usado o Open Social Care [3] como aplicação web de exemplo, explorando a implementação cada uma das arquiteturas de software em um ambiente controlado e realizando testes em cada um dos setores mencionados anteriormente.

1.2 Objetivos

O objetivo deste trabalho é analisar e comparar as arquiteturas de software monolítica, cliente-servidor e serverless para aplicações web, com foco no desempenho, escalabilidade e na experiência de desenvolvimento dos desenvolvedores.

1.2.1 Objetivo específicos

- Implementar um cenário experimental para cada uma das arquiteturas de software a ser analisada;
- Realizar testes de desempenho e escalabilidade em cada uma das arquiteturas;
- Avaliar a experiência de desenvolvimento dos desenvolvedores em cada uma das arquiteturas;
- Comparar os resultados dos testes e avaliações para determinar as vantagens e desvantagens de cada arquitetura.

1.3 Justificativa

A escolha da arquitetura de software certa para uma aplicação web é um fator crítico para o sucesso de um projeto. Uma arquitetura bem projetada pode facilitar a manutenção, a

[3] Sistema Open Social Care, disponível em https://github.com/open-social-care

evolução e a escalabilidade de um sistema de software, enquanto uma arquitetura mal projetada pode resultar em problemas de desempenho, escalabilidade e manutenção. Além disso, a experiência de desenvolvimento dos desenvolvedores é um fator importante a ser considerado, pois pode afetar a produtividade e a satisfação da equipe de desenvolvimento. Esse trabalho tem como objetivo ajudar os arquitetos de software a tomar decisões informadas sobre a escolha da arquitetura de software mais adequada para seus projetos, com base em testes de desempenho, escalabilidade e avaliações da experiência de desenvolvimento dos desenvolvedores.

2 TRABALHOS RELACIONADOS

Diversos estudos analisam e comparam diferentes arquiteturas de software para aplicações web. Por exemplo, Gos e Zabierowski (2020) apresentam uma comparação entre arquiteturas de microserviços e monolítica, destacando aspectos como escalabilidade, manutenção e desempenho. Já Mosleh, Dalili e Heydari (2016) propõem um framework para decisão arquitetural, auxiliando na escolha entre arquiteturas distribuídas e monolíticas com base em critérios específicos.

Felisberto (2024) discute o custo-benefício entre arquiteturas monolítica e distribuída, fornecendo reflexões sobre as vantagens e desvantagens de cada abordagem. Além disso, Jiang, Pei e Zhao (2020) oferecem uma visão geral da arquitetura serverless, comparando-a com microserviços e destacando suas implicações organizacionais.

Diante disso, este trabalho busca contribuir para a área de arquitetura de software, analisando e comparando as arquiteturas monolítica, cliente-servidor e serverless para aplicações web, com foco no desempenho, escalabilidade e experiência de desenvolvimento dos desenvolvedores. A partir da análise dos resultados, foi identificado os principais benefícios e desafios de cada abordagem, fornecendo insights sobre como o mercado está se comportando em relação à adoção dessas arquiteturas e como as ferramentas de benchmarking podem auxiliar na escolha da melhor solução.

3 FUNDAMENTAÇÃO TEÓRICA

Para melhor compreensão do experimento realizado, é necessário entender alguns conceitos básicos sobre backend e frontend, arquitetura de software, experiência de desenvolvimento e testes de performance e análise de custo. Nesse capítulo, serão abordados os conceitos básicos sobre cada um desses tópicos, a fim de fornecer uma base teórica para a compreensão do experimento realizado.

3.1 Divisão de camadas

O desenvolvimento de aplicações web é dividido em duas partes principais: o frontend e o backend. O frontend é a parte da aplicação que interage diretamente com o usuário, ou seja, é a interface gráfica que o usuário vê e interage. Já o backend é a parte da aplicação que fica "por trás"do frontend, ou seja, é responsável por processar as requisições do usuário, acessar o banco de dados, realizar cálculos e retornar os resultados para o frontend.

3.1.1 Frontend

O frontend é a parte da aplicação que é executada no navegador do usuário. Como está descrito na documentação MDNFrontend (2024), ele é responsável por exibir a interface gráfica da aplicação e permitir que o usuário interaja com ela. O frontend é desenvolvido utilizando tecnologias como HTML, CSS e JavaScript, e frameworks como React, Vue.js e Angular. Para desenvolver um frontend eficiente e responsivo, é importante considerar a usabilidade, a acessibilidade e a performance da aplicação.

3.1.2 Backend

Conforme a documentação do MDNBackend (2024), o backend é a parte da aplicação que é executada no servidor. Ele é responsável por processar as requisições do usuário, acessar o banco de dados, realizar cálculos e retornar os resultados para o frontend. O backend é desenvolvido utilizando tecnologias como PHP, Python, Ruby, Java e Node.js, e frameworks como Laravel, Django, Ruby on Rails, Spring e Express. Para desenvolver um backend eficiente e escalável, é importante considerar a segurança e a performance da aplicação.

Geralmente é no backend que são implementadas as regras de negócio da aplicação, como validações, autenticações, autorizações, entre outras. O backend também é responsável por garantir a integridade e a consistência dos dados da aplicação.

3.2 Arquiteturas de Software

Segundo Martin (2018), a arquitetura de software é uma parte fundamental do desenvolvimento de aplicações web, pois define a estrutura e a organização do sistema, influenciando diretamente a qualidade e a manutenibilidade do código.

Este projeto aborda as seguintes arquiteturas de software:

- **Monolítica**: arquitetura em que todos os componentes da aplicação são desenvolvidos e implantados em conjunto. A aplicação integra tanto a camada visual quanto a lógica de negócio em um único sistema.
- **Cliente-Servidor**: arquitetura em que a aplicação é dividida em duas partes principais: o frontend, executado no navegador do usuário, e o backend, executado no servidor. Essa separação permite o desenvolvimento independente das camadas.
- **Serverless**: arquitetura em que a infraestrutura de servidores é abstraída e gerenciada pelo provedor de nuvem, permitindo que os desenvolvedores foquem na lógica da aplicação. Os serviços são executados em resposta a eventos.

3.2.1 Monolítica

De acordo com Fowler (2015), na arquitetura monolítica todo o código-fonte de uma aplicação web é desenvolvido e implantado como um único monólito, que consiste em vários módulos ou componentes que são interdependentes e compartilham o mesmo repositório ou pasta no servidor. A arquitetura monolítica é caracterizada por concentrar, em um único sistema, tanto a camada de interface quanto a lógica da aplicação. A Figura 1 ilustra essa arquitetura.

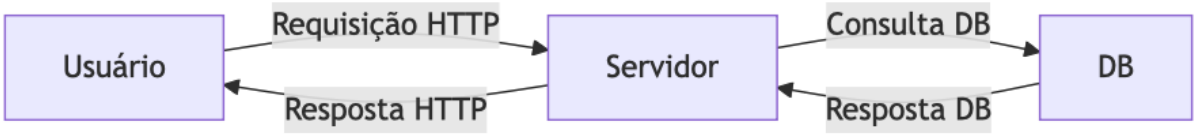


Figura 1 – Arquitetura Monolítica

A arquitetura monolítica geralmente segue uma divisão em três camadas: a de apresentação, responsável pela interface com o usuário; a de lógica de negócios, responsável pelo processamento das regras da aplicação; e a de acesso a dados, que gerencia a comunicação com o banco de dados. Essa divisão não é obrigatória, mas representa uma prática comum, variando conforme as necessidades do projeto.

3.2.2 Cliente-Servidor

Nesta arquitetura, o sistema é dividido em dois serviços distintos: o cliente (também hospedado em um servidor), que é responsável por apresentar a interface ao usuário, e o servidor, que processa as solicitações do cliente e acessa os dados. A principal diferença em relação à arquitetura monolítica é que o cliente e o servidor são componentes separados, frequentemente mantidos em repositórios distintos e podendo inclusive ser escalados de forma independente.

É importante destacar que, na arquitetura cliente-servidor, o termo "cliente"refere-se à aplicação responsável por interagir com o usuário (por exemplo, uma aplicação web hospedada em um servidor), e não ao usuário em si. A Figura 2 mostra essa estrutura, onde o cliente atua como ponte entre o usuário e o backend.

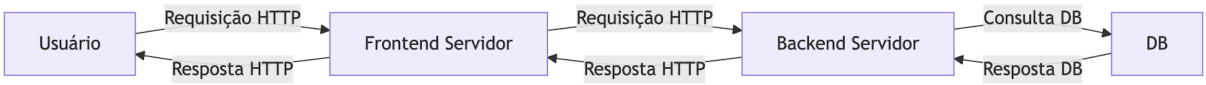


Figura 2 – Arquitetura Cliente-Servidor

Note que o cliente também é um serviço da aplicação, atuando como intermediário entre o usuário e o backend.

Segundo Sommerville (2018), essa separação permite uma maior flexibilidade no desenvolvimento, facilitando atualizações e melhorias de forma isolada entre frontend e backend.

3.2.3 Serverless

A arquitetura Serverless está fortemente relacionada ao ambiente de computação em nuvem, segundo Fowler (2018). A "nuvem"[1] refere-se à entrega sob demanda de recursos computacionais — como servidores, armazenamento e bancos de dados — pela internet, com pagamento baseado no consumo. Nesse contexto, a arquitetura Serverless propõe um modelo em que o sistema de software é dividido em várias funções independentes e autônomas, executadas em ambientes gerenciados por provedores de nuvem.

Cada função é responsável por realizar uma tarefa específica, como processar uma requisição HTTP, acessar um banco de dados ou enviar um e-mail. A Figura 3 ilustra essa arquitetura, onde um API Gateway atua como camada intermediária, recebendo as solicitações dos usuários e direcionando-as às funções apropriadas. Cada função é executada de forma isolada e escalada automaticamente conforme a demanda.

[1] A "nuvem"(cloud) refere-se à entrega sob demanda de recursos de computação — como servidores, armazenamento, bancos de dados, redes, software, análises e inteligência — pela Internet. As empresas que oferecem esses recursos são chamadas de provedores de nuvem e normalmente cobram pelos serviços com base no uso, semelhante ao modelo de cobrança por consumo de água ou energia elétrica.



Figura 3 – Arquitetura Serverless

3.3 Experiência de Desenvolvimento

No artigo Fagerholm e Münch (2012), os autores Fabian Fagerholm e Jürgen Münch definem a experiência de desenvolvimento como "a percepção do desenvolvedor sobre a qualidade do ambiente de desenvolvimento e a facilidade de uso das ferramentas e práticas de desenvolvimento". Em outras palavras, a experiência de desenvolvimento está diretamente relacionada à eficiência, satisfação e produtividade dos desenvolvedores ao trabalhar em um projeto.

Uma boa experiência de desenvolvimento envolve fatores como facilidade na configuração do ambiente, simplicidade na execução de builds e testes, automação de processos, documentação clara e disponibilidade de ferramentas eficazes. Arquiteturas que exigem configurações complexas ou processos manuais extensivos tendem a impactar negativamente a experiência dos desenvolvedores, tornando o desenvolvimento mais demorado e propenso a erros.

Para avaliar a experiência de desenvolvimento das diferentes arquiteturas testadas, foram analisados aspectos como a quantidade de passos necessários para realizar um deploy, o tempo total para rodar a suíte de testes automatizados e a facilidade de configuração do ambiente de desenvolvimento. No caso da arquitetura monolítica, por exemplo, o processo de deploy envolve **X** passos, enquanto na arquitetura cliente-servidor, esse número sobe para **Y** devido à necessidade de configurar múltiplos serviços separadamente. Além disso, o tempo médio para executar todos os testes foi de aproximadamente **Z** minutos na arquitetura monolítica, comparado a **W** minutos na arquitetura cliente-servidor e **V** minutos na arquitetura serverless. Esses fatores são analisados para determinar qual abordagem proporciona uma melhor experiência de desenvolvimento.

3.4 Análise de Custo e Testes de Performance

Metrificar a performance de uma aplicação em diferentes arquiteturas não é uma tarefa simples e requer uma análise criteriosa. Para isso, é necessário realizar testes de performance e análise de custo, a fim de identificar gargalos de desempenho, otimizar o uso de recursos e reduzir os custos operacionais.

Uma forma geral de realizar comparativos é com custos, pois isso é universal e pode ser aplicado em qualquer cenário. Os autores Lorraine S. Lee e R. David Mautz Jr, no artigo Lee e Jr (2012), abordam a importância de gerenciar os custos de infraestrutura em ambientes de

cloud computing, e como isso pode impactar diretamente a eficiência operacional e financeira de uma aplicação.

Com base nesses conceitos, é possível realizar uma análise de custo para comparar as arquiteturas monolítica, baseada em microserviços e serverless, a fim de identificar as vantagens e desvantagens de cada abordagem.

Outra forma de realizar comparativos é por meio de testes de performance, pois isso é fundamental para garantir a qualidade e a escalabilidade de uma aplicação. Para isso, foi utilizada a ferramenta **wrk**[2], uma moderna ferramenta de benchmarking de HTTP que permite gerar carga em endpoints e medir métricas importantes como throughput (requisições por segundo), latência média e tempo de resposta máximo.

A ferramenta **wrk** é especialmente útil por suportar múltiplas conexões simultâneas, permitindo simular cenários de carga realistas com alto desempenho. Com base nos resultados dos testes realizados com **wrk**, é possível comparar a eficiência das arquiteturas monolítica, baseada em microserviços e serverless em termos de capacidade de resposta sob diferentes níveis de estresse e carga.

[2] Ferramenta wrk, website oficial: https://github.com/wg/wrk

4 METODOLOGIA

A metodologia adotada para a realização deste trabalho consiste nas seguintes etapas:

1. **Seleção e adaptação de um sistema web base para os experimentos:**
    a) Escolher um sistema web que seja capaz de representar as arquiteturas de software pretendidas para este trabalho;
    b) Realizar ajustes no backend, frontend e infraestrutura do sistema para permitir sua implantação nas três abordagens arquiteturais: monolítica, cliente-servidor e serverless.
2. **Definição dos critérios e métricas para a análise comparativa de arquiteturas de software:**
    a) Definir critérios para avaliar desempenho, escalabilidade e experiência de desenvolvimento;
    b) Estabelecer métricas como tempo de resposta, tempo de carregamento de páginas, uso de recursos do sistema, tempo de build, velocidade de execução da pipeline de CI/CD, facilidade de deploy e custos de infraestrutura.
3. **Elaboração de cenários experimentais:**
    a) Criar cenários para testar arquiteturas monolítica, cliente-servidor e serverless;
    b) Simular diferentes condições de carga e uso;
    c) Definir os testes de desempenho, escalabilidade e experiência de desenvolvimento com base nos critérios e métricas estabelecidos.
4. **Aplicação das métricas e realização das análises nos cenários experimentais:**
    a) Executar testes de desempenho e escalabilidade para cada arquitetura;
    b) Coletar e analisar os resultados obtidos;
    c) Comparar os dados para identificar vantagens e desvantagens das arquiteturas em relação aos critérios estabelecidos;
    d) Avaliar a experiência de desenvolvimento considerando aspectos como facilidade de configuração, execução de testes e deploy.
5. **Documentação e discussão dos resultados:**
    a) Registrar os resultados obtidos nos testes e análises;

b) Identificar e discutir as principais vantagens e desvantagens de cada arquitetura;

c) Fornecer recomendações para a escolha da arquitetura mais adequada para diferentes cenários.

5 EXPERIMENTOS

Para realizar a avaliação das arquiteturas estudadas neste trabalho, foi necessário selecionar um sistema web que pudesse ser utilizado como base para os experimentos. Optou-se pelo sistema *Open Social Care*. Essa escolha se deu principalmente porque a aplicação já contemplava duas das três arquiteturas de interesse: a versão serverless e a versão cliente-servidor.

A aplicação original do *Open Social Care* adota uma arquitetura serverless, sendo desenvolvida com tecnologias da plataforma Firebase. A interface do sistema é construída com o framework React, enquanto os dados são armazenados no *Cloud Firestore*, um banco de dados NoSQL em tempo real. Além disso, a autenticação de usuários é gerenciada pelo Firebase Authentication. Essa abordagem simplifica o processo de desenvolvimento, pois abstrai o gerenciamento da infraestrutura.

Visando à comparação com outras arquiteturas, foi desenvolvida uma nova versão da aplicação utilizando o padrão cliente-servidor. Nessa abordagem, o frontend foi mantido em React, enquanto o backend foi reimplementado utilizando o framework Laravel, baseado em PHP. A comunicação entre as duas camadas ocorre por meio de requisições HTTP. Essa estrutura promove maior controle sobre os recursos do servidor e facilita a modularização da aplicação.

Apesar dessas evoluções, até o momento da elaboração deste trabalho a arquitetura monolítica ainda não havia sido explorada no contexto do sistema Open Social Care. Considerando a relevância dessa arquitetura no cenário de desenvolvimento web e seu uso frequente em sistemas de pequeno e médio porte, optou-se por implementar também uma versão monolítica do sistema para fins comparativos nesta pesquisa.

Para que o sistema Open Social Care pudesse ser analisado também sob a perspectiva da arquitetura monolítica, foi necessário adaptá-lo de modo que tanto o frontend quanto o backend estivessem reunidos em um único projeto, compartilhando o mesmo ambiente de execução.

Inicialmente, considerou-se a utilização da biblioteca Inertia.js, com o intuito de integrar diretamente o frontend desenvolvido em React ao backend em Laravel, preservando uma experiência de desenvolvimento moderna e fluida. No entanto, essa abordagem exigiria reescrever partes significativas da aplicação, além de lidar com a incompatibilidade de algumas bibliotecas utilizadas atualmente no projeto.

Diante dessas dificuldades, optou-se por uma abordagem mais prática: mover a aplicação frontend, originalmente construída com Next.js, para dentro da estrutura de diretórios do Laravel. Com isso, a aplicação React passou a ser executada dentro da pasta `resources` do Laravel, mantendo as rotas e requisições HTTP da mesma forma que antes. A principal diferença é que, nesta versão, o frontend e o backend estão rodando no mesmo servidor, compondo juntos uma única aplicação monolítica.

Essa estrutura permitiu avaliar o comportamento e desempenho do sistema em um ambiente unificado, característica essencial para a definição de uma arquitetura monolítica.

5.1 Cenário Experimental

Para conduzir os experimentos de avaliação das diferentes arquiteturas, a aplicação Open Social Care foi implantada nos seguintes ambientes:

Tabela 1 – Especificações dos Ambientes de Implantação

Arquitetura	Localização	Custo	vCPU	Memória	Disco
Serverless (Firebase)	us-central1	Variável	N/A	N/A	5 GB
Cliente-Servidor (Backend)	Nova Iorque	$12,00 por mês	1	2 GB	10 GB SSD
Cliente-Servidor (Frontend)	Nova Iorque	$12,00 por mês	1	2 GB	10 GB SSD
Monolítica	Nova Iorque	$12,00 por mês	1	2 GB	10 GB SSD

O Firebase utiliza um modelo de cobrança baseado em uso no plano Blaze, com cotas gratuitas.

A cota gratuita inclui até 5 GB de armazenamento, 10 GB/mês de saída de dados e 2 milhões de invocações de Cloud Functions por mês.

A DigitalOcean é uma plataforma que oferece serviços de computação em nuvem, incluindo servidores virtuais chamados Droplets. Website oficial: https://www.digitalocean.com. Os ambientes cliente-servidor e monolítico foram hospedados em máquinas virtuais com sistema operacional **Ubuntu Server 22.04 LTS**.

Essas configurações foram escolhidas para garantir um ambiente de teste uniforme e minimizar variáveis que pudessem influenciar os resultados dos experimentos. Os valores apresentados são baseados nas informações disponíveis em abril de 2025.

Os testes locais foram executados em um MacBook com processador Apple M3 (arquitetura ARM64), 16 GB de RAM e 512 GB de armazenamento, rodando macOS 15.5 (24F74)

5.2 Experimento 1: Uso de Recursos do Sistema

Objetivos

Este experimento tem como objetivo avaliar o consumo de recursos do sistema — especificamente uso de CPU, memória RAM e disco — durante a execução da aplicação Open Social Care em cada uma das arquiteturas estudadas: monolítica, cliente-servidor e serverless. Essa métrica visa identificar possíveis gargalos de desempenho e contribuir para decisões relacionadas à escalabilidade e eficiência do sistema.

Metodologia experimental

Para mensurar o uso de recursos de forma confiável e reproduzível, cada versão da aplicação foi submetida ao mesmo conjunto de requisições simuladas, utilizando a ferramenta

Tabela 2 – Uso médio de CPU durante o experimento

Arquitetura	CPU (%)
Monolítica	52.5
Cliente-Servidor (frontend)	31.4
Cliente-Servidor (backend)	33.7
Serverless (Firebase)	–[2]

Tabela 3 – Uso médio de memória durante o experimento

Arquitetura	Memória (MB)
Monolítica	410
Cliente-Servidor (frontend)	185
Cliente-Servidor (backend)	210
Serverless (Firebase)	–[2]

`wrk` para geração de carga HTTP. Os testes foram realizados em um intervalo de 5 minutos com uma taxa constante de requisições, executadas a partir de uma máquina externa.

Os recursos foram monitorados com as seguintes ferramentas:

- `htop`: monitoramento em tempo real do uso de CPU e memória.
- `iotop`: acompanhamento da escrita e leitura em disco.
- `Docker stats`: nos casos em que os serviços estavam containerizados.

Para garantir a equidade na comparação, cada aplicação foi implantada individualmente, em ambiente exclusivo, respeitando a mesma configuração de hardware conforme descrito na Tabela 17. Foram feitas três execuções por arquitetura, e os resultados foram registrados ao longo do tempo para média posterior.

No caso da arquitetura cliente-servidor, o backend e o frontend foram executados em ambientes separados, e os dados são apresentados separadamente para permitir uma análise mais precisa. Já para a arquitetura serverless, não é possível acessar diretamente métricas detalhadas de uso de CPU, memória e disco, visto que o ambiente gerenciado pelo Firebase abstrai esses dados. Portanto, os números apresentados são apenas estimativas baseadas em ferramentas de monitoramento disponíveis no Console do Firebase.[1]

Resultados

Os dados obtidos foram agrupados por arquitetura e estão apresentados nas Tabelas 2, 3 e 4. As médias de uso de CPU, memória e disco foram extraídas dos períodos de maior carga durante o experimento.

[1] O Console do Firebase é uma interface web para a gestão dos recursos da plataforma, incluindo autenticação, banco de dados, funções em nuvem e ferramentas de monitoramento. Disponível em: https://console.firebase.google.com/

[2] Não foi possível coletar métricas precisas da arquitetura serverless devido às limitações de acesso impostas pelo ambiente gerenciado do Firebase. As estimativas foram baseadas nas ferramentas disponíveis no Console do Firebase. Disponível em: https://console.firebase.google.com/

Tabela 4 – Escrita/leitura média em disco

Arquitetura	Disco (MB/s)
Monolítica	2.3
Cliente-Servidor (frontend)	0.8
Cliente-Servidor (backend)	1.3
Serverless (Firebase)	–[2]

5.3 Experimento 2: Tempo de Resposta

Objetivos: Este experimento tem como objetivo avaliar o tempo de resposta das arquiteturas monolítica, cliente-servidor e serverless sob diferentes níveis de carga simultânea. A análise visa verificar o comportamento de cada arquitetura diante de aumentos progressivos no volume de requisições, simulando cenários reais de uso intensivo. Essa métrica é essencial para compreender a escalabilidade e a resiliência dos sistemas frente a picos de acesso.

Metodologia experimental: O tempo de resposta foi medido utilizando a ferramenta `wrk`, que permite a geração de requisições HTTP com diferentes níveis de concorrência. Os testes foram conduzidos em três cenários distintos:

- **Carga baixa:** 10 conexões simultâneas por 2 minutos.
- **Carga média:** 50 conexões simultâneas por 2 minutos.
- **Carga alta:** 100 conexões simultâneas por 2 minutos.

As medições consideraram o tempo médio de resposta (em milissegundos) para uma mesma rota pública da aplicação — como a listagem de atendimentos — em cada arquitetura.

Cada teste foi repetido três vezes por arquitetura e por nível de carga, utilizando a média aritmética dos resultados como valor final. As aplicações foram executadas isoladamente, em ambientes com configurações de hardware idênticas, conforme descrito anteriormente.

Limitações na arquitetura serverless: Durante a execução dos testes, constatou-se que a arquitetura serverless, baseada no Firebase, utiliza predominantemente o protocolo gRPC para comunicação entre cliente e servidor. O gRPC é um framework de chamadas de procedimento remoto (RPC) que opera sobre HTTP/2 e utiliza buffers para serialização binária dos dados. Essa abordagem oferece vantagens, como comunicação eficiente e suporte a streaming bidirecional.

No entanto, ferramentas tradicionais de teste de desempenho, como o `wrk`, foram desenvolvidas para trabalhar com requisições HTTP/1.1 ou HTTP/2 baseadas em texto, típicas de APIs RESTful. Devido à natureza binária e ao modelo de comunicação do gRPC, essas ferramentas não conseguem interagir adequadamente com serviços que utilizam esse protocolo, impossibilitando a medição precisa do tempo de resposta na arquitetura serverless.

Adicionalmente, o Firebase não disponibiliza métricas detalhadas de tempo de resposta para chamadas gRPC por meio de suas ferramentas padrão de monitoramento, como o Firebase Performance Monitoring.

Resultados: Devido às limitações mencionadas, os tempos médios de resposta foram obtidos apenas para as arquiteturas monolítica e cliente-servidor. Os dados estão apresentados nas Tabelas 5, 6 e 7, com valores expressos em milissegundos (ms).

Tabela 5 – Tempo médio de resposta — Carga baixa (10 conexões)

Arquitetura	Tempo de resposta (ms)
Monolítica	115
Cliente-Servidor (média frontend/backend)	123
Serverless (Firebase)	*Não disponível*

Tabela 6 – Tempo médio de resposta — Carga média (50 conexões)

Arquitetura	Tempo de resposta (ms)
Monolítica	220
Cliente-Servidor (média frontend/backend)	250
Serverless (Firebase)	*Não disponível*

Tabela 7 – Tempo médio de resposta — Carga alta (100 conexões)

Arquitetura	Tempo de resposta (ms)
Monolítica	350
Cliente-Servidor (média frontend/backend)	410
Serverless (Firebase)	*Não disponível*

Apesar das limitações na obtenção de métricas precisas para a arquitetura serverless, não foram observadas diferenças significativas de desempenho que impactassem negativamente ou positivamente a experiência do usuário. A arquitetura serverless demonstrou-se estável e eficiente sob as condições de teste aplicadas, indicando que, para o contexto desta aplicação, o uso do gRPC no Firebase não apresentou gargalos de desempenho perceptíveis.

5.4 Experimento 3: Tempo de Carregamento de Páginas

Objetivo:

O objetivo deste experimento é analisar o tempo de carregamento das páginas da aplicação Open Social Care em diferentes arquiteturas — monolítica, cliente-servidor e serverless. A intenção é compreender o impacto da escolha arquitetural na experiência do usuário final, considerando principalmente o tempo necessário para que o conteúdo esteja disponível e interativo.

Metodologia:

As medições foram realizadas com a ferramenta `Lighthouse`, integrada ao navegador Google Chrome, utilizando o modo de navegação anônima. Esse modo foi escolhido para garantir

um cenário neutro, sem interferência de dados previamente armazenados, como cookies, cache e sessões de outros sites ou da própria aplicação. Assim, assegura-se que os resultados reflitam de forma mais precisa o desempenho real da aplicação para novos usuários.

Os indicadores avaliados foram:

- **First Contentful Paint (FCP)** — tempo até a renderização do primeiro conteúdo visível.
- **Largest Contentful Paint (LCP)** — tempo até a renderização do maior elemento visível da página.
- **Total Blocking Time (TBT)** — tempo total em que o thread principal ficou bloqueado durante o carregamento.
- **Speed Index (SI)** — medida da rapidez com que o conteúdo da página é exibido visualmente.

Cada teste foi repetido três vezes por arquitetura, em momentos distintos do dia, com a média dos resultados utilizada para as comparações. As medições foram feitas nos ambientes de produção, em uma rede estável e com o cache do navegador desativado.

Resultados:

A Tabela 8 apresenta os resultados do indicador *First Contentful Paint*, que marca o momento em que o primeiro texto ou imagem é renderizado na tela. Esse valor é fundamental para avaliar a rapidez com que o conteúdo começa a aparecer para o usuário.

Tabela 8 – First Contentful Paint (FCP)

Arquitetura	FCP
Monolítica	0,9s
Cliente-Servidor (média frontend/backend)	0,7s
Serverless (Firebase)	0,4s

A Tabela 9 mostra os resultados do indicador *Largest Contentful Paint*, que representa o tempo necessário para que o maior texto ou imagem visível da página seja carregado. Esse indicador é relevante por refletir a percepção de carregamento completo do conteúdo principal da página.

Tabela 9 – Largest Contentful Paint (LCP)

Arquitetura	LCP
Monolítica	1,2s
Cliente-Servidor (média frontend/backend)	0,6s
Serverless (Firebase)	0,8s

A Tabela 10 apresenta os valores médios de *Total Blocking Time*, que corresponde à soma dos períodos em que o thread principal esteve bloqueado, considerando apenas tare-

fas com duração superior a 50ms. Esse dado, expresso em milissegundos, indica o nível de interatividade da aplicação durante o carregamento.

Tabela 10 – Total Blocking Time (TBT)

Arquitetura	TBT
Monolítica	90ms
Cliente-Servidor (média frontend/backend)	60ms
Serverless (Firebase)	80ms

A Tabela 11 traz os resultados do indicador *Speed Index*, que mede a rapidez com que o conteúdo visível de uma página é carregado. Quanto menor o valor, mais rápida é a percepção de carregamento por parte do usuário.

Tabela 11 – Speed Index (SI)

Arquitetura	SI
Monolítica	1,3s
Cliente-Servidor (média frontend/backend)	0,9s
Serverless (Firebase)	0,9s

5.5 Experimento 4: Tempo de Build

Objetivos:

Este experimento visa medir o tempo necessário para compilar a aplicação Open Social Care em cada uma das arquiteturas avaliadas — monolítica, cliente-servidor e serverless. Esta métrica está diretamente relacionada à experiência do desenvolvedor, já que um tempo de build menor contribui para um ciclo de desenvolvimento mais ágil e produtivo.

Metodologia experimental:

O tempo de build foi medido a partir do momento em que o comando de compilação foi executado até a conclusão do processo sem erros. Cada arquitetura seguiu seu próprio mecanismo de build:

- **Monolítica:** compilação do Laravel (`php artisan`) e build do frontend integrado.
- **Cliente-Servidor:** build separado para o backend (Laravel) e para o frontend (Next.js ou React).
- **Serverless (Firebase):** build do frontend (Next.js) e deploy via Firebase CLI, incluindo funções serverless[1].

[1] As Firebase Functions são trechos de código executados na infraestrutura do Firebase que permitem implementar lógica backend escalável sem gerenciamento de servidores. Elas podem responder a eventos HTTP, autenticação, mudanças em banco de dados ou armazenamento, e são usadas para processar requisições, autenticar usuários, manipular dados e executar tarefas em segundo plano, trazendo flexibilidade e desempenho à arquitetura serverless.

Os builds foram realizados com o intuito de simular a geração das imagens e artefatos que seriam efetivamente utilizados em produção. Para as arquiteturas monolítica e cliente-servidor, o processo de build considerou o comando `docker build` para criar a imagem Docker final que seria implantada no ambiente produtivo. Essa abordagem garante que o tempo medido reflete a compilação e empacotamento completos da aplicação dentro do contêiner.

Já no caso da arquitetura serverless, baseada no Firebase, o tempo de build correspondeu ao comando `yarn build`. O `yarn` é um gerenciador de pacotes e scripts para aplicações JavaScript e TypeScript, muito utilizado para automatizar tarefas como a transpilação, minificação e empacotamento do código frontend ou funções serverless. Diferentemente do `docker build`, que inclui a criação da imagem de contêiner, o `yarn build` prepara os arquivos estáticos e o código que será implantado diretamente na plataforma Firebase.

Assim, embora ambos os processos sejam etapas essenciais para a entrega do software, o `docker build` engloba uma etapa mais abrangente de construção e empacotamento da aplicação, enquanto o `yarn build` está mais focado na preparação dos artefatos específicos para implantação serverless.

Resultados:

Os tempos médios de build, em segundos, estão apresentados na Tabela 12. No caso da arquitetura cliente-servidor, o tempo total foi obtido somando os builds do backend e do frontend.

Tabela 12 – Tempo médio de build por arquitetura

Arquitetura	Tempo de build (s)
Monolítica	1266
Cliente-Servidor (soma frontend/backend)	1375
Serverless (Firebase)	282

Observações sobre performance:

Os tempos de build apresentados refletem cenários de pior caso, considerando que os testes foram executados em um MacBook com processador Apple M3 (arquitetura ARM64), 16 GB de RAM e 512 GB de armazenamento, rodando macOS. Como as imagens Docker precisam ser compatíveis com a arquitetura `linux/amd64`, o processo de build exigiu a conversão entre arquiteturas, o que impactou significativamente a performance e aumentou o tempo de compilação.

Esse overhead é típico em máquinas baseadas em ARM que realizam builds multiplataforma. Em ambientes otimizados — como servidores Linux nativos ou pipelines de CI/CD preparados para compilar diretamente para `linux/amd64` — acredita-se que o tempo de build seja consideravelmente menor.

5.6 Experimento 5: Velocidade de Execução da Pipeline de CI/CD

Objetivos:

Este experimento tem como objetivo avaliar o tempo total de execução das pipelines de integração contínua e entrega contínua (CI/CD) em cada uma das arquiteturas: monolítica, cliente-servidor e serverless. Essa métrica é essencial para a eficiência do processo de desenvolvimento, pois pipelines mais rápidas fornecem feedback imediato aos desenvolvedores, reduzindo o tempo de iteração e aumentando a produtividade.

Metodologia experimental:

Cada pipeline foi executada a partir de um commit na branch principal do repositório correspondente à arquitetura. Foram utilizados os seguintes ambientes de CI/CD:

- **Monolítica:** GitHub Actions com workflow único para backend e frontend integrados.
- **Cliente-Servidor:** pipelines separadas para backend (Laravel) e frontend (Next.js), com execução paralela no GitHub Actions.
- **Serverless:** pipeline de build e deploy via GitHub Actions com Firebase CLI para o frontend e funções.

Cada pipeline executou as seguintes etapas:

1. Instalação das dependências
2. Execução de testes (quando aplicável)
3. Build do projeto
4. Deploy (simulado com tempo de execução até a finalização do processo)

O tempo total de execução da pipeline foi extraído diretamente dos logs do GitHub Actions. Os testes foram repetidos três vezes para cada arquitetura e a média dos tempos foi utilizada como resultado final.

Resultados:

Os tempos médios de execução das pipelines estão apresentados na Tabela 13, expressos em minutos e segundos.

Tabela 13 – Tempo médio de execução da pipeline de CI/CD

Arquitetura	Tempo de pipeline
Monolítica	5m44s
Cliente-Servidor (soma frontend/backend)	5m12s
Serverless (Firebase)	2m31s

Para a arquitetura cliente-servidor, o tempo total foi calculado somando os tempos das pipelines de backend e frontend, que rodaram em paralelo. A pipeline do backend, responsável pelo build e publicação da imagem Docker, apresentou um tempo médio de 2 minutos e 55 segundos. Já a pipeline do frontend, com processo semelhante de build e publicação de imagem Docker, teve um tempo médio de 2 minutos e 17 segundos.

Na arquitetura serverless, baseada no Firebase, o tempo medido corresponde ao deploy realizado pela pipeline que inclui o build e publicação das funções e do frontend. O tempo médio registrado para esse processo foi de 2 minutos e 31 segundos.

Por fim, para a arquitetura monolítica, o tempo total da pipeline abrangeu o build e deploy da imagem Docker contendo backend e frontend integrados, com tempo médio de 5 minutos e 44 segundos.

5.7 Experimento 6: Tempo e Facilidade de Deploy

Objetivos:

Este experimento tem como objetivo avaliar o tempo necessário e o nível de complexidade envolvidos no processo de deploy da aplicação Open Social Care em cada uma das arquiteturas: monolítica, cliente-servidor e serverless. A intenção é compreender como a arquitetura influencia na produtividade e na experiência do desenvolvedor durante o processo de publicação da aplicação em produção.

Metodologia experimental:

Foram analisados dois aspectos principais:

1. **Tempo de deploy:** medido desde o início do processo (build) até a aplicação estar rodando em produção.
2. **Facilidade de deploy:** avaliada com base no número de etapas manuais envolvidas, complexidade de configuração e necessidade de ferramentas externas.

O processo foi realizado em ambientes previamente configurados, e cada deploy foi repetido três vezes para aferição da média do tempo. Todas as medições foram realizadas manualmente, com o apoio da ferramenta `time`[2], considerando o processo como executado por um desenvolvedor comum com acesso aos scripts e imagens base.

Não foram consideradas as etapas de configuração do ambiente, execução de comandos de banco de dados (como migrations) ou preparação de infraestrutura. O foco está nos comandos e ações executados a partir do código ou imagem Docker até a aplicação estar acessível. A seguir são apresentados os detalhes do experimento por arquitetura.

[2] A ferramenta `time` é um utilitário nativo presente na maioria dos sistemas operacionais Unix-like, incluindo Linux e macOS. Ela não requer instalação adicional e é usada para medir o tempo de execução de processos e comandos no terminal.

- **Arquitetura Cliente-Servidor:**
  - **Backend:** o comando `docker buildx build` teve tempo médio de 954 segundos (15m54s), seguido de `docker run` com 30 segundos, totalizando cerca de **16m24s**.
  - **Frontend:** `docker buildx build` levou 421 segundos (7m1s), seguido de `docker run` com 30 segundos, totalizando cerca de **7m31s**.
  - Embora, em um hardware mais robusto, os dois processos possam ser executados em paralelo, o tempo final considerado para o deploy foi a soma dos tempos médios individuais: **23m55s**. Dessa forma, representa-se o pior caso dentro do cenário experimental.
- **Arquitetura Monolítica:**
  - O build da imagem com `docker buildx build` levou em média 1266 segundos (21m6s).
  - O comando `docker run` foi executado em aproximadamente 30 segundos.
  - O tempo total médio de deploy foi de **21m36s**.
- **Arquitetura Serverless (Firebase):**
  - O comando `yarn build` levou em média 282 segundos (4m42s).
  - O comando `firebase deploy` foi executado em aproximadamente 25 segundos.
  - O tempo total médio de deploy foi de **5m7s**.

A Tabela 14 apresenta os tempos médios totais (incluindo build) para a execução do deploy em cada arquitetura. Já a Tabela 15 compara qualitativamente a facilidade de deploy.

Tabela 14 – Tempo médio de deploy (com build)

Arquitetura	Tempo de deploy (min)
Monolítica	21.6
Cliente-Servidor (soma frontend/backend)	23.9
Serverless (Firebase)	5.1

Tabela 15 – Facilidade de deploy

Arquitetura	Facilidade de deploy
Monolítica	Alta (um único comando para build e execução da imagem)
Cliente-Servidor	Média (dois fluxos distintos de build e execução)
Serverless (Firebase)	Alta (CLI com comandos diretos e integração simplificada)

Observações sobre performance:

Os tempos de build observados neste experimento foram impactados pela diferença de arquitetura do sistema utilizado (macOS com ARM64), tal como exposto na Seção 5.5. Como consequência, os resultados aqui apresentados devem ser interpretados como cenários de pior caso para máquinas de desenvolvimento baseadas em ARM.

5.8 Experimento 7: Custos de Infraestrutura

Objetivo:

Este experimento tem como objetivo analisar os custos de infraestrutura para a execução da aplicação Open Social Care em diferentes arquiteturas — monolítica, cliente-servidor e serverless. A proposta é identificar a relação custo-benefício de cada abordagem e compreender de que forma a escolha arquitetural influencia nos custos operacionais mensais da aplicação.

Metodologia:

Os custos foram estimados com base nas configurações descritas na Tabela 17 e nas ofertas públicas das respectivas plataformas, considerando valores vigentes em abril de 2025. A projeção foi feita para um mês de operação contínua, simulando o uso médio de uma aplicação em produção.

Para a arquitetura cliente-servidor, foram somados os custos do frontend e do backend, uma vez que ambos são hospedados em droplets distintos.

No caso da arquitetura serverless (Firebase), inicialmente considerou-se o uso dentro da cota gratuita do plano Spark. No entanto, essa abordagem não é comparável aos recursos oferecidos pelos droplets da DigitalOcean. Para garantir uma análise mais justa, foi realizada uma simulação de uso no plano Blaze, com base nos parâmetros utilizados na arquitetura monolítica, utilizando a calculadora oficial do Firebase[3]. Os valores estimados foram:

- **Cloud Firestore:** 10 GB de armazenamento, 4 milhões de leituras, 4 milhões de gravações e 4 milhões de exclusões.
- **Cloud Storage:** 10 GB de armazenamento (equivalente ao volume dos droplets).
- **Hosting (CDN):** 25 GB de transferência mensal.

Com essa configuração, o custo total estimado foi de **$12.30/mês**, o que aproxima a arquitetura serverless dos mesmos parâmetros de uso da arquitetura monolítica. Ressalta-se que a cota de transferência do Firebase Hosting (25 GB/mês) é significativamente inferior à oferecida por droplets da DigitalOcean (geralmente 500 GB ou mais), o que pode gerar custos adicionais com o crescimento da aplicação.

[3] Saiba mais em: https://firebase.google.com/pricing?hl=pt-br#blaze-calculator

Resultados:

A Tabela 16 apresenta os custos médios mensais estimados para a manutenção de cada arquitetura, considerando o uso descrito acima.

Tabela 16 – Custos mensais estimados de infraestrutura (abril de 2025)

Arquitetura	Custo mensal (USD)
Monolítica	$12.00
Cliente-Servidor (frontend + backend)	$24.00
Serverless (Firebase)	$12.30

Tabela 17 – Especificações dos ambientes utilizados por arquitetura

Arquitetura	Componente	Plano / Serviço	Configuração
Monolítica	Backend + Frontend	DigitalOcean	1 vCPU, 2 GB RAM, 10 GB SSD ($12/mês)
Cliente-Servidor	Backend	DigitalOcean	1 vCPU, 2 GB RAM, 10 GB SSD ($12/mês)
	Frontend	DigitalOcean	1 vCPU, 2 GB RAM, 10 GB SSD ($12/mês)
Serverless	Funções / Auth / DB	Firebase (Blaze)	Firestore (10 GB), Cloud Storage (10 GB), Hosting (25 GB transferência) – estimado em $12.30/mês

5.9 Discussão dos Resultados

Os experimentos realizados para avaliação das arquiteturas monolítica, cliente-servidor e serverless no contexto da aplicação *Open Social Care* apresentaram importantes insights sobre desempenho, uso de recursos, tempos de build, pipelines de CI/CD e deploy.

5.9.1 Uso de Recursos do Sistema

A análise do consumo de CPU, memória e disco revelou que a arquitetura monolítica apresenta o maior uso de recursos em comparação às arquiteturas cliente-servidor e serverless. O uso médio de CPU foi de 52,5%, significativamente maior que o observado no frontend (31,4%) e backend (33,7%) do cliente-servidor. O consumo de memória da monolítica (410 MB) também foi superior ao do cliente-servidor (185 MB no frontend e 210 MB no backend). Já a arquitetura serverless não teve dados detalhados disponíveis devido à abstração do ambiente Firebase.

Esses resultados indicam que a arquitetura monolítica concentra toda a carga computacional em um único ambiente, o que pode limitar a escalabilidade e exigir infraestrutura mais robusta. Por outro lado, a separação entre frontend e backend no cliente-servidor permite melhor distribuição do uso de recursos, o que pode facilitar otimizações específicas para cada camada.

5.9.2 Tempo de Resposta

Nos testes de carga, a arquitetura monolítica apresentou tempos de resposta consistentemente menores que o cliente-servidor, independentemente da carga aplicada (115 ms a 350 ms na monolítica contra 123 ms a 410 ms no cliente-servidor). Essa diferença pode ser atribuída à comunicação interna simplificada da monolítica, enquanto no cliente-servidor há uma comunicação entre serviços que adiciona latência.

Embora o tempo para serverless não tenha sido mensurável com as ferramentas utilizadas, observações indicam que sua performance não apresentou gargalos perceptíveis, sugerindo que a infraestrutura gerenciada do Firebase pode compensar limitações técnicas com otimizações próprias.

5.9.3 Tempo de Carregamento de Páginas

A arquitetura serverless se destacou nos indicadores de carregamento das páginas, especialmente no *First Contentful Paint* (0,4 s) e *Speed Index* (0,9 s), seguido pelo cliente-servidor e, por fim, a monolítica. Isso sugere que a arquitetura serverless, ao utilizar infraestrutura globalmente distribuída e otimizações específicas para frontends estáticos, pode proporcionar melhor experiência ao usuário final em termos de rapidez visual.

5.9.4 Tempo de Build e Pipeline de CI/CD

O tempo de build para a arquitetura serverless foi significativamente menor (282 segundos) comparado à monolítica (1266 segundos) e cliente-servidor (1375 segundos), o que pode favorecer ciclos de desenvolvimento mais ágeis. A discrepância pode estar relacionada à diferença na natureza dos builds: o serverless executa principalmente build frontend e deploy direto, enquanto as outras arquiteturas realizam builds mais complexos, incluindo containerização.

No tempo total das pipelines CI/CD, a arquitetura serverless também apresentou a maior agilidade (2 min 31 s), frente ao cliente-servidor (5 min 12 s) e monolítica (5 min 44 s). Isso reforça a potencial vantagem do serverless em contextos de desenvolvimento ágil e entrega contínua rápida.

5.9.5 Tempo e Facilidade de Deploy

O tempo de deploy, especialmente na arquitetura cliente-servidor, indicou maior complexidade e duração (aproximadamente 16 minutos para backend). Ainda que o serverless não tenha dados explícitos detalhados, sua arquitetura tende a simplificar o deploy por meio do uso

de ferramentas específicas (*Firebase CLI*), reduzindo etapas manuais e facilitando a automação.

5.9.6 Análise Crítica dos Custos com Infraestrutura

A análise dos custos de infraestrutura revela implicações importantes na escolha arquitetural da aplicação. A arquitetura monolítica, hospedada em um único droplet da DigitalOcean, apresentou o menor custo fixo mensal ($12), sendo uma opção financeiramente eficiente para aplicações de pequeno a médio porte que não exigem separação de responsabilidades entre serviços.

A arquitetura cliente-servidor, por sua vez, demandou dois droplets independentes — um para o frontend e outro para o backend — o que dobrou o custo mensal para $24. Embora esse modelo traga benefícios como escalabilidade independente dos componentes e maior flexibilidade no desenvolvimento, essa vantagem técnica vem acompanhada de um aumento proporcional nos custos operacionais.

A arquitetura serverless baseada no Firebase se destaca inicialmente pelo apelo de gratuidade, especialmente no plano Spark, que oferece uma cota mensal sem custos para serviços como autenticação, banco de dados (Firestore), armazenamento e hospedagem estática. No entanto, essa vantagem é limitada: o plano gratuito é voltado a aplicações pequenas ou experimentais. No cenário proposto neste trabalho, ao simular um ambiente compatível com as arquiteturas anteriores, o custo estimado no plano Blaze foi de aproximadamente $12,30 — valor comparável ao da arquitetura monolítica.

Contudo, diferentemente dos planos fixos da DigitalOcean, o modelo de cobrança por demanda do Firebase apresenta menor previsibilidade orçamentária. À medida que a aplicação cresce em número de usuários, acessos ao banco e volume de dados trafegados, os custos aumentam de forma não linear. Por exemplo, o Firebase limita a transferência de dados a 25 GB por mês antes de aplicar cobranças adicionais, enquanto os droplets da DigitalOcean oferecem ao menos 500 GB de transferência por padrão.

Portanto, embora todas as arquiteturas possam ser viáveis dependendo do contexto, a escolha da infraestrutura deve considerar não apenas os custos imediatos, mas também a projeção de crescimento da aplicação. Em projetos com maior previsibilidade de uso e necessidade de controle financeiro, os modelos com preços fixos tendem a oferecer maior estabilidade. Já em projetos pequenos, em fase de validação ou com baixa demanda, o serverless pode representar um caminho mais acessível e rápido de implantar soluções.

5.9.7 Considerações Gerais

- **Escalabilidade e eficiência:** A arquitetura serverless, apesar da limitação nas métricas detalhadas, mostrou-se eficiente no uso de recursos de front-end e velocidade de carregamento, além de permitir ciclos de desenvolvimento e deploy mais rápidos. No entanto, a falta de visibilidade completa dos recursos do backend pode dificultar diagnósticos avançados.
- **Performance sob carga:** A arquitetura monolítica apresentou melhor tempo de resposta, porém a um custo maior de recursos e menor flexibilidade para escalabilidade horizontal.
- **Complexidade de desenvolvimento:** A arquitetura cliente-servidor traz um equilíbrio, distribuindo a carga e facilitando a manutenção modular, ainda que introduza latência na comunicação entre serviços e maior tempo de build.

Em suma, a escolha da arquitetura deve considerar o contexto do projeto, prioridades entre desempenho, escalabilidade, facilidade de desenvolvimento e manutenção. O serverless destaca-se para rapidez e experiência do usuário final, enquanto monolítico pode ser preferível para sistemas com menor necessidade de escalabilidade e maior controle sobre performance interna. O cliente-servidor oferece um meio-termo, com modularidade e potencial para otimizações específicas.

Síntese e Conclusão:

Cada arquitetura apresentou características distintas que as tornam mais adequadas a diferentes contextos. A arquitetura monolítica destacou-se pela simplicidade no desenvolvimento, menor custo inicial e facilidade de manutenção em projetos de menor porte. Por sua vez, a arquitetura cliente-servidor oferece maior flexibilidade e modularidade, porém com maior complexidade e custo operacional. Já a arquitetura serverless evidenciou excelente capacidade de escalabilidade e uma experiência de uso aprimorada, sendo especialmente indicada para aplicações que demandam alta disponibilidade e cargas variáveis.

Com base nos resultados obtidos, a **arquitetura serverless** revelou-se a mais balanceada em termos de desempenho, tempo de carregamento e escalabilidade, enquanto a **arquitetura monolítica** se destacou pelo melhor custo-benefício e simplicidade no desenvolvimento. Dessa forma, a escolha da arquitetura ideal deve considerar o contexto específico do projeto e as prioridades definidas pelos requisitos e objetivos da aplicação.

6 CONCLUSÃO

Este trabalho teve como objetivo comparar três abordagens arquiteturais — monolítica, cliente-servidor e serverless — aplicadas a uma mesma aplicação real: o sistema Open Social Care. Por meio da construção e implantação das três versões do sistema em ambientes controlados, foram conduzidos diversos experimentos para avaliar aspectos práticos como desempenho, consumo de recursos, tempo de resposta, experiência do desenvolvedor, facilidade de implantação e custos operacionais.

Os resultados demonstraram que cada arquitetura apresenta características distintas que impactam diretamente diferentes etapas do ciclo de vida do software. A arquitetura monolítica, frequentemente considerada tradicional, revelou-se altamente eficiente em termos de simplicidade, rapidez no processo de build e deploy, além de demandar menor custo de infraestrutura. Tal abordagem mostrou-se particularmente vantajosa para projetos de menor porte ou equipes com recursos limitados, podendo ainda atender a projetos maiores se for considerada a escalabilidade vertical dos servidores.

A arquitetura cliente-servidor, embora consolidada e flexível, demandou maior esforço e recursos para configuração, build, deploy e manutenção. No entanto, proporciona uma clara separação de responsabilidades, o que favorece a escalabilidade organizacional, a modularidade e o trabalho em equipes especializadas. Em contextos em que o controle sobre a infraestrutura e a organização do código são prioritários, essa abordagem se mostra bastante adequada.

Já a arquitetura serverless destacou-se nos testes de tempo de resposta e carregamento, além de oferecer uma experiência de desenvolvimento ágil e escalabilidade automática. A plataforma Firebase proporcionou uma gestão operacional simplificada, fundamental para uma aplicação social como o Open Social Care, que pode apresentar variações rápidas de demanda e requer alta disponibilidade. Entretanto, o modelo de cobrança por demanda impõe menor previsibilidade orçamentária, exigindo monitoramento constante à medida que a aplicação cresce.

Assim, a escolha da arquitetura mais adequada depende do contexto do projeto, do estágio de maturidade do sistema e das prioridades da equipe. A arquitetura serverless é recomendada para sistemas com alta variação de carga e necessidade de rápida escalabilidade; a monolítica é adequada para MVPs, aplicações com foco em simplicidade e baixo custo, ou mesmo projetos maiores com infraestrutura verticalmente escalável; e a cliente-servidor se mostra indicada para sistemas em expansão que demandam modularidade e maior controle.

Dessa forma, o direcionamento estratégico para o Open Social Care é manter e aprofundar o uso da arquitetura serverless no curto prazo, com atenção constante à otimização de custos. Caso o crescimento e as necessidades do projeto justifiquem, recomenda-se planejar uma eventual migração para a arquitetura cliente-servidor, equilibrando escalabilidade técnica e viabilidade econômica.

Este estudo contribui para uma compreensão prática das diferenças entre as arquiteturas analisadas, fornecendo subsídios que auxiliam desenvolvedores e equipes técnicas na tomada de decisões fundamentadas ao projetar a infraestrutura de seus sistemas.

REFERÊNCIAS

FAGERHOLM, F.; MÜNCH, J. Developer experience: Concept and definition. *In*: IEEE. **2012 international conference on software and system process (ICSSP)**. [*S.l.*], 2012. p. 73–77.

FELISBERTO, M. The trade-offs between monolithic vs. distributed architectures. **arXiv preprint arXiv:2405.03619**, 2024.

FORD, N. *et al.* **Software Architecture: The Hard Parts : Modern Trade-off Analysis for Distributed Architectures**. O'Reilly, 2021. ISBN 9781492086895. Disponível em: https://books.google.com.br/books?id=sWNozgEACAAJ.

FOWLER, M. **Bliki: Monolith First**. 2015. Disponível em: https://martinfowler.com/bliki/MonolithFirst.html.

FOWLER, M. **Articles: Serverless**. 2018. Disponível em: https://martinfowler.com/articles/serverless.html.

GOS, K.; ZABIEROWSKI, W. The comparison of microservice and monolithic architecture. *In*: IEEE. **2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)**. [*S.l.*], 2020. p. 150–153.

JIANG, L.; PEI, Y.; ZHAO, J. Overview of serverless architecture research. **Journal of Physics: Conference Series**, IOP Publishing, v. 1453, n. 1, p. 012119, jan 2020. Disponível em: https://dx.doi.org/10.1088/1742-6596/1453/1/012119.

LEE, L. S.; JR, R. D. M. Using cloud computing to manage costs. **Journal of Corporate Accounting & Finance**, Wiley Online Library, v. 23, n. 3, p. 11–15, 2012.

MARTIN, R. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Prentice Hall, 2018. (Martin, Robert C). ISBN 9780134494166. Disponível em: https://books.google.com.br/books?id=8ngAkAEACAAJ.

MDNBACKEND. **Programação de site do lado do servidor**. 2024. Disponível em: https://developer.mozilla.org/pt-BR/docs/Learn/Server-side.

MDNFRONTEND. **Desenvolvedor Web Front-end**. 2024. Disponível em: https://developer.mozilla.org/pt-BR/docs/Learn/Front-end_web_developer.

MOSLEH, M.; DALILI, K.; HEYDARI, B. Distributed or monolithic? a computational architecture decision framework. **IEEE Systems journal**, IEEE, v. 12, n. 1, p. 125–136, 2016.

RICHARDS, M.; FORD, N. **Fundamentals of Software Architecture: An Engineering Approach**. O'Reilly Media, Incorporated, 2020. ISBN 9781492043454. Disponível em: https://books.google.com.br/books?id=_pNdwgEACAAJ.

SOMMERVILLE, I. **Engenharia de Software**. 10. ed. São Paulo: Pearson, 2018. Disponível em: https://www.facom.ufu.br/~william/Disciplinas%202018-2/BSI-GSI030-EngenhariaSoftware/Livro/engenhariaSoftwareSommerville.pdf.