UNIVERSIDADE FEDERAL DO CEARÁ

CURSO DE GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

PAULO THIAGO PEREIRA DO VALE

COMPARAÇÃO ENTRE ARQUITETURAS DE SOFTWARE MONOLÍTICA E DE MICROSSERVIÇOS

SOBRAL

2025

PAULO THIAGO PEREIRA DO VALE

COMPARAÇÃO ENTRE ARQUITETURAS DE SOFTWARE MONOLÍTICA E DE MICROSSERVIÇOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de em Engenharia da Computação.

Orientador: Prof. Dr. Thiago Iachiley Araújo de Souza

SOBRAL

2025

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

V243c Vale, Paulo Thiago Pereira do.
COMPARAÇÃO ENTRE ARQUITETURAS DE SOFTWARE MONOLÍTICA E DE MICROSSERVIÇOS / Paulo Thiago Pereira do Vale. – 2025.
52 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Sobral, Curso de Engenharia da Computação, Sobral, 2025.
Orientação: Prof. Dr. Thiago Iachiley Araújo de Souza.

1. Arquitetura de Software. 2. Engenharia de Software. 3. Monólito. 4. Microsserviços. I. Título.
CDD 621.39

PAULO THIAGO PEREIRA DO VALE

COMPARAÇÃO ENTRE ARQUITETURAS DE SOFTWARE MONOLÍTICA E DE MICROSSERVIÇOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de em Engenharia da Computação.

Aprovada em: 08 de Agosto de 2025

BANCA EXAMINADORA

Prof. Dr. Thiago Iachiley Araújo de Souza (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Evilásio Costa Júnior
Universidade Federal do Ceará (UFC)

Prof. Dr. Diego Caitano de Pinho
Universidade da Integração Internacional da Lusofonia Afro-Brasileira (UNILAB)

À ciência.

AGRADECIMENTOS

Aos meus pais Jean Parcelli Costa do Vale e Marliete Pereira do Vale.

Ao meu irmão Jean Matheus Pereira do Vale.

À minha namorada Elma Luce Silveira Pessoa e sua (minha nova) família.

Ao professor Dr. Thiago Iachiley, pela orientação para esse trabalho.

À UFC como um todo por ser a instituição que me abriu as portas para o conhecimento.

“A persistência é o caminho do êxito.”

(Charles Chaplin)

RESUMO

Este estudo investiga o persistente debate entre as arquiteturas de software monolítica e de microsserviços, focando na transição da análise teórica para a validação empírica. O objetivo principal é quantificar os trade-offs associados a cada abordagem em um ambiente experimental controlado. Para tal, foi conduzida uma pesquisa que consistiu na implementação de protótipos funcionalmente idênticos de um sistema CRUD, um em arquitetura monolítica e outro em microsserviços, utilizando o ecossistema WEB: JavaScript com Node.js, Express e Docker. A análise comparativa foi realizada a partir de um conjunto de métricas objetivas, incluindo tempo de resposta de API, uso de CPU e memória, e a complexidade do processo de implantação. Os dados coletados revelam que a arquitetura monolítica se destaca pela menor latência e pela simplicidade de infraestrutura. Em contrapartida, os custos da arquitetura de microsserviços foram quantificados através de uma latência 147% superior na operação de orquestração, um consumo de memória em repouso 271% maior e uma complexidade que exigiu mais de três vezes o número de artefatos de configuração. Como contribuição acadêmica, este trabalho apresenta um estudo de caso prático e reprodutível que serve como ferramenta de apoio à decisão para arquitetos de software e equipes de desenvolvimento, fundamentando uma escolha estratégica com evidências quantitativas e qualitativas sobre os custos inerentes a cada abordagem.

Palavras-chave: Monólito. Microsserviços. Arquitetura de Software. Engenharia de Software.

ABSTRACT

This study investigates the persistent debate between monolithic and microservices software architectures, focusing on the transition from theoretical analysis to empirical validation. The main objective is to quantify the trade-offs associated with each approach in a controlled experimental environment. To this end, an experimental study was conducted, consisting of the implementation of functionally identical prototypes of a CRUD system—one with a monolithic architecture and the other with microservices—using the WEB JavaScript ecosystem with Node.js, Express, and Docker. The comparative analysis was performed based on a set of objective metrics, including API response time, CPU and memory usage, and the complexity of the deployment process. The collected data reveal that the monolithic architecture excels in lower latency and infrastructure simplicity. Conversely, the costs of the microservices architecture were quantified through a 147% higher latency in the orchestration operation, a 271% greater memory consumption at rest, and a complexity that required more than three times the number of configuration artifacts. As an academic contribution, this work presents a practical and reproducible case study that serves as a decision-support tool for software architects and development teams, grounding a strategic choice in quantitative and qualitative evidence regarding the inherent costs of each approach.

Keywords: Monolith. Microservices. Software Architecture. Software Engineering.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A gestão da complexidade em sistemas de software é um desafio central da engenharia desde suas primeiras formulações teóricas (BROOKS, 1972). A disciplina de Arquitetura de Software consolidou-se como resposta a este desafio, estabelecendo que as decisões estruturais de um sistema são determinantes para atributos não-funcionais críticos como manutenibilidade e escalabilidade (PERRY; WOLF, 1992). No contexto contemporâneo, este desafio se manifesta no dilema entre duas abordagens arquiteturais predominantes: a monolítica e a de microsserviços.

A arquitetura monolítica, o padrão tradicional de desenvolvimento, oferece uma simplicidade inicial inegável e permanece como uma escolha estratégica para um vasto espectro de projetos, especialmente no desenvolvimento de Produtos Mínimos Viáveis (MVPs), em startups e em sistemas com escopo bem definido, onde a velocidade de entrega e o baixo custo operacional inicial são críticos. Contudo, a arquitetura de microsserviços emergiu como uma alternativa proeminente, formalizada e popularizada por trabalhos seminais como o de (LEWIS, 2014). Esta abordagem, que propõe a decomposição de um sistema em um conjunto de serviços independentes, foi adotada em larga escala por empresas de hiperescala (como Netflix e Amazon) como uma solução para gerenciar a complexidade extrema de seus domínios de negócio e a organização de centenas de equipes de desenvolvimento. A promessa de benefícios como escalabilidade granular, resiliência a falhas e autonomia para equipes, conforme extensivamente documentado por (NEWMAN, 2019), tornou-a o padrão de fato para sistemas distribuídos de grande porte.

Entretanto, o sucesso da adoção da arquitetura de microsserviços por essas gigantes da tecnologia frequentemente mascara o imenso investimento em engenharia de plataforma e maturidade operacional que tal migração exige. A flexibilidade da distribuição é obtida ao custo de uma complexidade intrínseca significativamente maior. Questões como a comunicação inter-serviços, a garantia de consistência de dados distribuídos e a necessidade de uma infraestrutura robusta para orquestração, monitoramento e descoberta de serviços são barreiras práticas, documentadas por autores como (RICHARDSON, 2018). Ele, entre outros, mapeou um ecossistema complexo de padrões de design necessários para mitigar os problemas inerentes a um sistema distribuído. Esta elevada barreira de entrada técnica é o que torna a escolha arquitetural um dilema tão crítico e o foco central deste estudo.

1.1 Problematização

Apesar de a literatura teórica descrever de forma robusta estes trade-offs, a decisão de qual arquitetura adotar na prática industrial ainda carece, em muitos casos, de uma base empírica sólida conforme mencionado por trabalhos como (Di Francesco *et al.*, 2019; GHOFRANI; LüBKE, 2018). A escolha é frequentemente influenciada por tendências de mercado ou por casos de sucesso de grandes corporações, sem uma análise quantitativa que se aplique ao contexto específico de um novo projeto. É precisamente nesta lacuna — a ausência de estudos experimentais controlados que quantifiquem o custo de desempenho, de recursos e de complexidade de implantação — que o presente trabalho se insere.

Este estudo tem como objetivo principal preencher parte dessa lacuna, realizando uma análise comparativa, fundamentada em um protocolo experimental, entre as arquiteturas monolítica e de microsserviços. Através do desenvolvimento e da mensuração de protótipos funcionalmente idênticos, busca-se gerar evidências quantitativas e qualitativas que auxiliem na tomada de decisões arquiteturais mais informadas e alinhadas aos requisitos específicos de cada projeto.

1.2 Questões de Pesquisa

A análise da literatura permite consolidar os principais trade-offs entre as arquiteturas em três eixos centrais: desempenho, custo de recursos e complexidade. A teoria postula que a arquitetura de microsserviços, ao introduzir a comunicação via rede e a distribuição de processos, paga um custo em latência e consumo de recursos em troca de benefícios de escalabilidade e manutenibilidade.

Com base nesta fundamentação teórica, o presente estudo experimental foi delineado para investigar e quantificar estes custos. As questões que norteiam a investigação empírica são:

- **Questão 1 (Desempenho):** A latência de resposta da arquitetura monolítica será inferior à da arquitetura de microsserviços, sendo esta diferença mais pronunciada em operações que exijam comunicação inter-serviços.
- **Questão 2 (Recursos):** O consumo de recursos em repouso (RAM) da arquitetura de microsserviços será superior ao do monólito, devido à sobrecarga da execução de múltiplos processos e contêineres.
- **Questão 3 (Complexidade):** A complexidade de implantação da arquitetura de microsser-

viços, medida pelo número de artefatos de configuração, será objetivamente maior que a da arquitetura monolítica.

1.3 Justificativa

A relevância deste estudo é fundamentada na necessidade premente de substituir a tomada de decisão arquitetural, frequentemente baseada em intuição ou tendências, por uma abordagem alicerçada em evidências empíricas. A escolha entre uma arquitetura monolítica e uma de microsserviços acarreta consequências técnicas e financeiras significativas para o ciclo de vida de um projeto de software. Decisões equivocadas, tomadas sem uma análise quantitativa, podem levar a custos operacionais imprevistos, complexidade acidental e dificuldades de manutenção que comprometem o sucesso do projeto. Portanto, a justificativa para esta pesquisa reside em três contribuições centrais:

- Contribuição Científica: O trabalho se propõe a preencher uma lacuna metodológica identificada na literatura, fornecendo um estudo experimental com um protocolo claro e reprodutível que pode servir de base para futuras investigações na área.
- Contribuição Prática: Oferece subsídios concretos — dados de desempenho, de consumo de recursos e de complexidade — que capacitam arquitetos de software e equipes de desenvolvimento a fundamentar suas escolhas, mitigando riscos e alinhando a arquitetura aos recursos e objetivos reais do negócio.
- Contribuição Acadêmica: Ao validar empiricamente os trade-offs teóricos, este estudo fortalece o corpo de conhecimento da Engenharia de Software, provendo um caso de análise detalhado que pode ser utilizado como material didático e referência para novos pesquisadores.

1.4 Trabalhos Relacionados

A necessidade de validar empiricamente os trade-offs entre as arquiteturas monolítica e de microsserviços tem motivado diversos estudos experimentais na comunidade científica. Estes trabalhos buscam, em sua maioria, quantificar as diferenças de desempenho e custo que a literatura teórica descreve.

Um estudo relevante é o de Freddy et al. (TAPIA *et al.*, 2020), que comparou o desempenho de uma aplicação web em uma estrutura monolítica operando em um servidor

virtual contra a mesma aplicação em uma arquitetura de microsserviços executada em contêineres. Através de testes de estresse e da aplicação de um modelo matemático de regressão, o estudo buscou quantificar as diferenças de performance, confirmando o interesse da indústria em obter métricas precisas para guiar a decisão arquitetural.

De forma similar, Grzegorz et al. (BLINOWSKI *et al.*, 2022) investigaram a performance e a escalabilidade das duas arquiteturas, mas com uma abordagem multi-tecnologia (Java e C# .NET) e em múltiplos ambientes de implantação, incluindo a nuvem Azure. Um dos achados centrais do estudo foi que, em um ambiente de máquina única, a arquitetura monolítica apresenta um desempenho superior ao seu equivalente em microsserviços. O trabalho questiona se os benefícios de migração, frequentemente citados por grandes corporações, se aplicam a sistemas de menor escala, um ponto que ressoa diretamente com a análise do presente TCC.

No contexto nacional, o trabalho de Nery (PEREIRA, 2022) adota uma metodologia análoga à deste estudo, combinando uma revisão bibliográfica com uma comparação prática entre dois servidores idênticos, implementados nas diferentes arquiteturas. A execução de casos de teste para coletar evidências e medir o desempenho permitiu ao autor emitir uma conclusão sobre a melhor utilização de cada arquitetura dentro do escopo definido.

Estes trabalhos, em conjunto, corroboram a relevância da comparação empírica e a validade da abordagem experimental. O presente estudo se alinha a essa linha de pesquisa, contribuindo com uma análise focada no ecossistema JavaScript com Node.js e Docker, e validando achados como os de (BLINOWSKI *et al.*, 2022) sobre a eficiência superior do monólito em cenários de máquina única, ao mesmo tempo que quantifica os custos de latência, recursos e complexidade de implantação.

1.5 Objetivos

1.5.1 Objetivo Geral

Realizar uma análise comparativa, fundamentada em um estudo experimental, entre as arquiteturas de software para a WEB monolítica e de microsserviços, a fim de quantificar e avaliar os trade-offs inerentes a cada abordagem em termos de desempenho, consumo de recursos e complexidade de implantação em um ambiente controlado.

1.5.2 Objetivos Específicos

Para alcançar o objetivo geral proposto, foram definidos os seguintes objetivos específicos:

- Desenvolver dois protótipos de software para a WEB funcionalmente idênticos, um implementando a arquitetura monolítica e outro a arquitetura de microsserviços, para servirem como base controlada para a comparação experimental.
- Analisar comparativamente a latência (tempo de resposta) de cada arquitetura, medindo o desempenho em operações de CRUD simples e em uma operação de escrita complexa que exija orquestração entre domínios.
- Medir e comparar o consumo de recursos computacionais (CPU e Memória RAM) de cada arquitetura em estado de repouso, a fim de avaliar o custo operacional base de cada abordagem.
- Avaliar qualitativamente a complexidade de implantação, através da análise e comparação dos artefatos de configuração e dos processos necessários para executar cada um dos ambientes.

Para atingir tais objetivos, este documento está estruturado da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica sobre as arquiteturas em estudo. O Capítulo 3 detalha a metodologia experimental empregada. O Capítulo 4 apresenta e discute os resultados obtidos. Finalmente, o Capítulo 5 conclui o trabalho, sintetizando os achados e sugerindo direções para pesquisas futuras.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a fundamentação teórica necessária para embasar a análise e discussão dos tópicos abordados neste trabalho. Serão explorados conceitos essenciais sobre arquitetura de software, com foco nas abordagens monolítica e de microsserviços, incluindo suas características, vantagens, desvantagens e casos de uso. Além disso, serão discutidos critérios e métricas relevantes para a comparação entre essas arquiteturas, bem como ferramentas e tecnologias associadas, fornecendo contexto para a compreensão do trabalho desenvolvido ao longo dos próximos capítulos.

2.1 Arquitetura de Software

A arquitetura de um sistema de software é o conjunto de decisões estruturais fundamentais sobre como o software é composto. Formalmente, ela pode ser definida como a estrutura ou conjunto de estruturas de um sistema, que compreende seus componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles (BASS *et al.*, 2013). O objetivo desta disciplina é planejar e organizar o sistema de forma a satisfazer tanto os requisitos funcionais quanto os não-funcionais. Para compreender uma arquitetura, é essencial analisar seus elementos primários:

- Componentes: São as unidades de computação ou de dados que compõem o sistema. Um componente pode variar em granularidade, desde uma classe individual, um módulo, até um serviço independente ou uma camada completa da aplicação (e.g., a camada de interface com o usuário ou a camada de acesso a dados).
- Conectores: São os elementos que mediam a comunicação e a cooperação entre os componentes. Os conectores podem ser simples, como chamadas de procedimento, ou complexos, como protocolos de rede (e.g., HTTP) ou barramentos de mensagens.

A forma como componentes e conectores são arranjados é ditada pelos princípios e atributos de qualidade que o sistema almeja alcançar. Estes atributos, também conhecidos como requisitos não-funcionais, incluem características como desempenho, escalabilidade, manutenibilidade, segurança e confiabilidade. Invariavelmente, otimizar para um atributo pode impactar negativamente outro, criando um cenário de trade-offs (pontos de compromisso). Por exemplo, uma arquitetura projetada para máxima segurança pode sacrificar parte de seu desempenho. A principal responsabilidade do arquiteto de software é, portanto, entender e

gerenciar esses trade-offs para atender às prioridades do projeto (RICHARDS; FORD, 2020).

É neste ponto que a discussão sobre a natureza do software se torna crucial. Diferente de outras engenharias, sistemas de software são abstratos e não são regidos por leis da física, o que permite que alcancem níveis de complexidade impraticáveis em outras áreas (SOMMERVILLE, 2011). Sem uma estrutura deliberada, essa complexidade não gerenciada resulta em sistemas frágeis, onde pequenas mudanças podem ter efeitos imprevisíveis. A arquitetura de software, portanto, desempenha o papel central de impor uma estrutura e um conjunto de regras que controlam essa complexidade, permitindo que os sistemas sejam desenvolvidos, mantidos e evoluídos de forma sustentável (SOMMERVILLE, 2011).

2.2 Arquitetura Monolítica

A arquitetura monolítica refere-se a um modelo de software no qual todos os componentes da aplicação são estruturados e implantados como uma única e indivisível unidade (RICHARDSON, 2018). Essa abordagem é caracterizada por sua simplicidade conceitual e operacional nos estágios iniciais de um projeto. Uma vez que todo o código-fonte, a lógica de negócio e as funcionalidades estão contidos em um único processo, atividades como o desenvolvimento, a depuração e a implantação inicial são significativamente simplificadas (NEWMAN, 2019). Esta simplicidade é tão reconhecida que estratégias como a "Monolith-First" são defendidas por autores como Fowler, que argumentam ser um ponto de partida pragmático para muitos sistemas antes de considerar uma eventual transição para modelos mais distribuídos.

No entanto, a principal crítica a este modelo emerge com o crescimento da aplicação. A manutenção de sistemas monolíticos em larga escala torna-se progressivamente complexa, um fenômeno frequentemente descrito como "dores do monólito" (monolith pains) por (NEWMAN, 2021). Problemas como o acoplamento excessivo entre componentes, onde uma alteração em uma parte do sistema pode ter efeitos colaterais inesperados em outra, tornam a evolução do software lenta e arriscada (HOSSAIN *et al.*, 2023). Adicionalmente, a escalabilidade horizontal é inerentemente ineficiente: como o sistema é uma unidade única, é necessário replicar toda a aplicação, mesmo que o gargalo de desempenho esteja concentrado em apenas uma pequena parte de suas funcionalidades (RICHARDSON, 2018).

Esses fatores podem limitar severamente a capacidade de evolução do sistema, especialmente em cenários onde a agilidade e a escalabilidade granular são requisitos críticos de negócio. Apesar dessas limitações, a arquitetura monolítica continua sendo uma escolha

arquitetural válida e estratégica para projetos de menor escopo, protótipos ou em contextos onde a simplicidade e o tempo de lançamento no mercado (time-to-market) são as principais prioridades (NEWMAN, 2019).

2.3 Arquitetura de Microsserviços

A arquitetura de microsserviços representa uma abordagem para o desenvolvimento de uma única aplicação como um conjunto de pequenos serviços, cada um executando em seu próprio processo e comunicando-se através de mecanismos leves, frequentemente uma API de recursos HTTP (LEWIS, 2014). Diferente da abordagem monolítica, onde os componentes são integrados em um único bloco, os microsserviços são organizados em torno de capacidades de negócio e são implantáveis de forma independente por meio de processos de automação totalmente consolidados (NEWMAN, 2019).

Essa modularidade granular gera benefícios significativos. A implantação independente de cada serviço reduz o escopo e o risco de cada deploy, permitindo ciclos de lançamento mais rápidos e seguros. Adicionalmente, a arquitetura promove a diversidade tecnológica, permitindo que cada serviço seja implementado com a pilha tecnológica mais adequada para sua função específica, um conceito conhecido como persistência poligota (polyglot persistence) quando aplicado a bancos de dados (RICHARDSON, 2018). A principal vantagem, no entanto, é a escalabilidade granular: cada serviço pode ser dimensionado horizontalmente de forma independente, alocando recursos computacionais precisamente onde são necessários, em vez de replicar todo o sistema (NEWMAN, 2021).

Contudo, a adoção de microsserviços introduz a complexidade inerente aos sistemas distribuídos. Conforme argumentado por (NEWMAN, 2021), a complexidade não desaparece, ela apenas se desloca do código para a infraestrutura e a comunicação. A gestão da comunicação entre os serviços, a garantia da consistência eventual dos dados entre diferentes bases de dados e o tratamento de falhas parciais em um ambiente distribuído são desafios não triviais (HOSSAIN *et al.*, 2023). Além disso, a complexidade operacional aumenta consideravelmente, exigindo uma infraestrutura robusta e uma cultura de DevOps madura, com ferramentas para automação de implantação (CI/CD), monitoramento distribuído e descoberta de serviços (service discovery) (RICHARDSON, 2018). Apesar desses desafios, a arquitetura de microsserviços é amplamente adotada em sistemas complexos e de grande escala, onde a flexibilidade, a escalabilidade e a capacidade de evolução contínua são prioridades estratégicas.

2.4 Comparação Conceitual entre Monolito e Microsserviços

Qualquer estilo arquitetural apresenta trade-offs: pontos fortes e fracos que devem ser avaliados de acordo com o contexto em que são utilizados, um princípio fundamental na aplicação de padrões arquiteturais, conforme detalhado por (FOWLER *et al.*, 2006). Este é precisamente o caso das arquiteturas monolítica e de microsserviços. A escolha entre elas não se baseia em uma suposta superioridade de um modelo sobre o outro, mas em uma análise criteriosa das suas características fundamentalmente opostas.

A atratividade da arquitetura de microsserviços reside nos benefícios que ela oferece a sistemas complexos e de grande escala. Conforme discutido por (FOWLER, 2015), os principais pontos fortes derivam diretamente de sua natureza distribuída: o reforço das fronteiras modulares facilita a organização do código e a divisão de responsabilidades entre equipes; a implantação independente de cada serviço reduz o risco e aumenta a frequência dos deploys; e a diversidade tecnológica permite a otimização da pilha de ferramentas para cada funcionalidade específica.

No entanto, estes mesmos benefícios vêm com custos intrínsecos e significativos, como aponta o mesmo autor. A natureza distribuída que permite a implantação independente também introduz uma complexidade de programação elevada, pois chamadas de rede são inerentemente mais lentas e menos confiáveis que chamadas de processo locais. O isolamento que permite a diversidade tecnológica também gera o desafio da consistência eventual, já que manter uma consistência forte entre bases de dados distintas é extremamente difícil. Por fim, a autonomia dos serviços exige uma complexidade operacional crítica, demandando uma equipe de operações madura e ferramentas robustas para gerenciar múltiplos serviços que são reimplantados com frequência (FOWLER, 2015).

Em contrapartida, a simplicidade inerente da arquitetura monolítica permanece uma vantagem estratégica precisamente por evitar esta sobrecarga distribuída. Ao manter a aplicação como uma única unidade, ela garante a consistência forte dos dados de forma trivial e elimina a complexidade da comunicação em rede e da gestão operacional de múltiplos serviços. Esta abordagem torna-se, portanto, uma opção pragmática para projetos em estágios iniciais ou com escopo bem definido, onde a velocidade de desenvolvimento e a baixa complexidade são prioritárias. A decisão arquitetural, portanto, se resume a um exercício de ponderação entre a simplicidade e a eficiência de um sistema coeso contra a flexibilidade e o custo de um sistema distribuído.

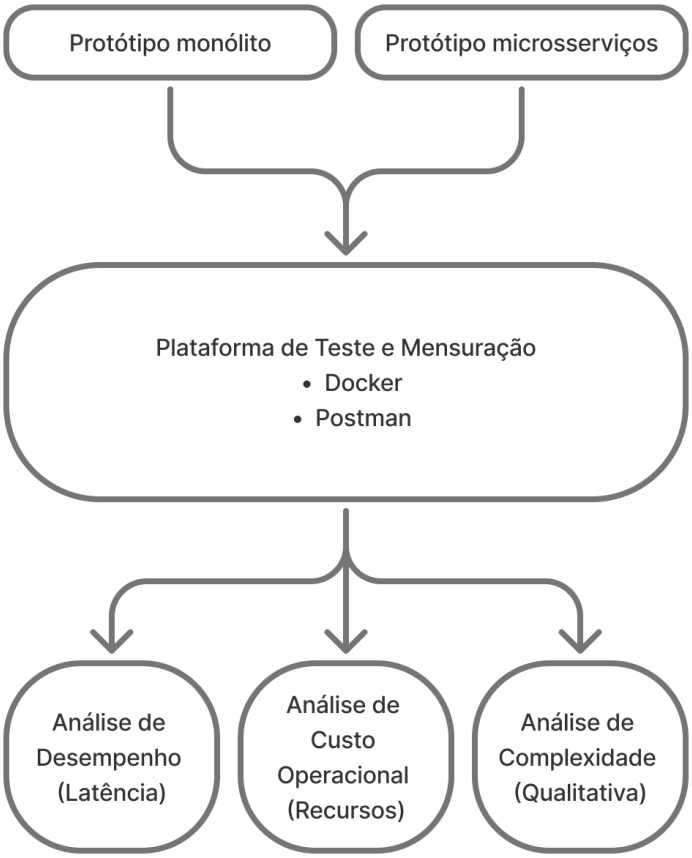
Os capítulos subsequentes detalharão a metodologia empregada para testar as ques-

tões de pesquisa determinadas no Capítulo 1 e apresentarão os resultados obtidos.

3 METODOLOGIA

Este capítulo detalha o rigor metodológico empregado para a condução da análise comparativa entre as arquiteturas de software monolítica e de microsserviços. A validade e a reprodutibilidade dos resultados apresentados nos capítulos subsequentes dependem fundamentalmente da precisão dos procedimentos aqui descritos. A Figura 1 ilustra como se deu o fluxo do experimento.

Figura 1 – Fluxo do experimento



Fonte: Elaborado pelo autor

3.1 Delineamento da Pesquisa

A presente pesquisa caracteriza-se como um estudo quantitativo de natureza experimental. A abordagem é comparativa, visando analisar o desempenho e o custo operacional de duas implementações arquiteturais distintas, porém funcionalmente idênticas, submetidas a um conjunto controlado de testes.

A escolha por um delineamento experimental justifica-se pela necessidade de transcender a análise puramente teórica, que domina grande parte da literatura sobre o tema. O objetivo é gerar evidências empíricas e quantificáveis que possam validar ou refutar as afirmações teóricas sobre os trade-offs de cada arquitetura, oferecendo subsídios concretos para a tomada de decisão técnica.

3.2 Ambiente Experimental

Todos os testes foram executados em um único ambiente de hardware e software para garantir a consistência e eliminar variáveis externas. A configuração do ambiente, que constitui a base para a reprodutibilidade deste estudo, consiste em uma máquina Apple MacBook Air M2 na versão com 16GB de RAM e 256GB de armazenamento SSD. A configuração de rede foi: internet via cabo Ethernet com velocidade 500 Mbps. Os softwares e ferramentas utilizados estarão descritos na subseção a seguir.

3.2.1 Ferramentas e Tecnologias

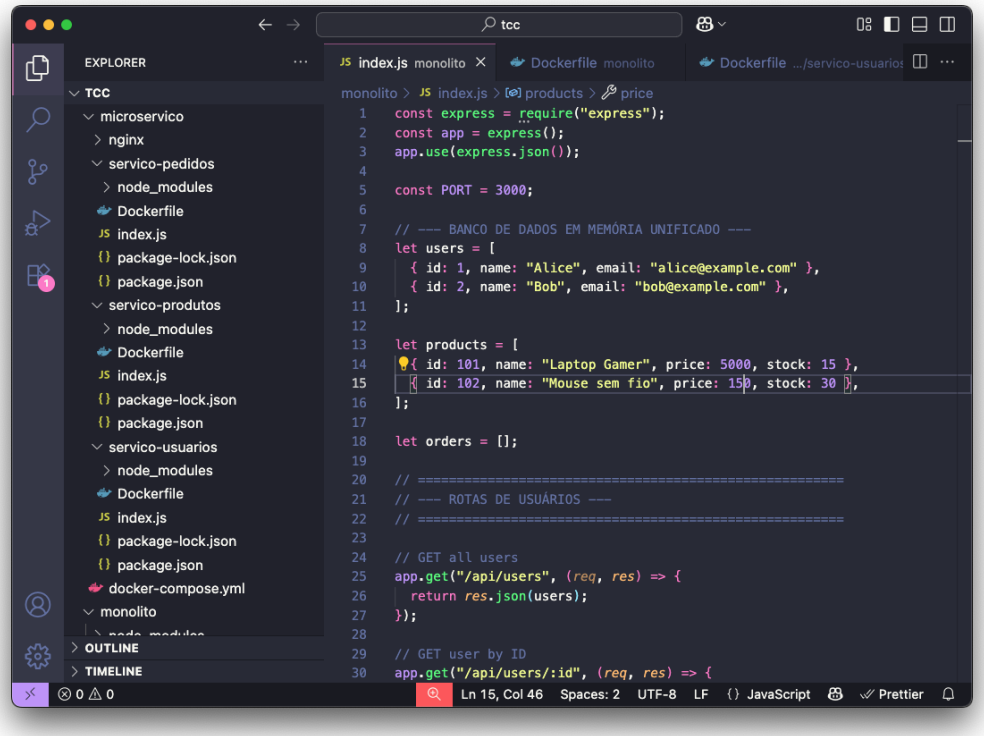
A implementação dos protótipos e a condução dos testes foram realizadas com o auxílio de um conjunto de tecnologias de software padrão da indústria, escolhidas por suas características específicas que garantem a validade e a reprodutibilidade do experimento. A pilha tecnológica é detalhada a seguir.

3.2.1.1 Plataforma de Backend: Node.js e Express.js

A lógica de negócio de ambos os protótipos foi desenvolvida em JavaScript e executada sobre o runtime Node.js (versão 22.14.0). Para a estruturação das APIs, o framework Express.js (versão 5.1.0) foi utilizado para gerenciar as rotas e os middlewares de forma concisa e padronizada. A comunicação HTTP entre os microsserviços foi implementada através da

biblioteca Axios. A Figura 2 ilustra um trecho do código-fonte desenvolvido para uma das APIs, demonstrando a sintaxe e a estrutura empregadas.

Figura 2 – Código fonte do backend



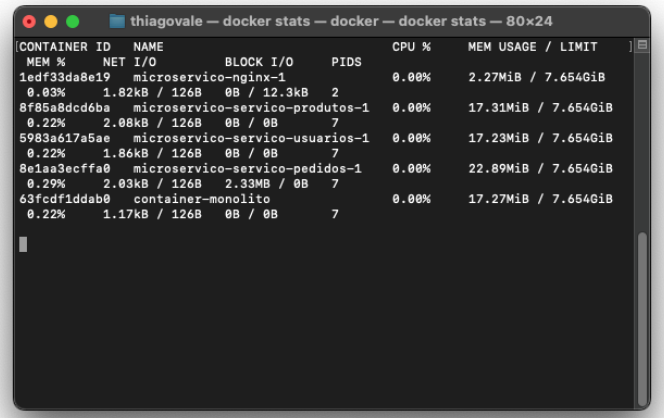
Fonte: Elaborado pelo autor

3.2.1.2 Plataforma de Containerização e Rede: Docker e Nginx

Para garantir o isolamento total entre os ambientes de teste e a reprodutibilidade dos resultados, a plataforma de containerização Docker (versão 4.43.1) foi empregada. Cada protótipo (o monólito e o conjunto de microsserviços) foi executado dentro de contêineres Docker, permitindo um controle preciso e a medição confiável do consumo de recursos (CPU e RAM) através do comando docker stats. A Figura 3 exibe a saída deste comando no terminal, utilizado para a coleta em tempo real dos dados de consumo dos contêineres.

Na arquitetura de microsserviços, um contêiner adicional executando o servidor web Nginx (versão stable-alpine) foi configurado para atuar como um Proxy Reverso (API Gateway). Sua função foi receber todas as requisições HTTP externas e roteá-las para o microsserviço interno correspondente, simulando um ponto de entrada único.

Figura 3 – Comando docker stats em execução no terminal

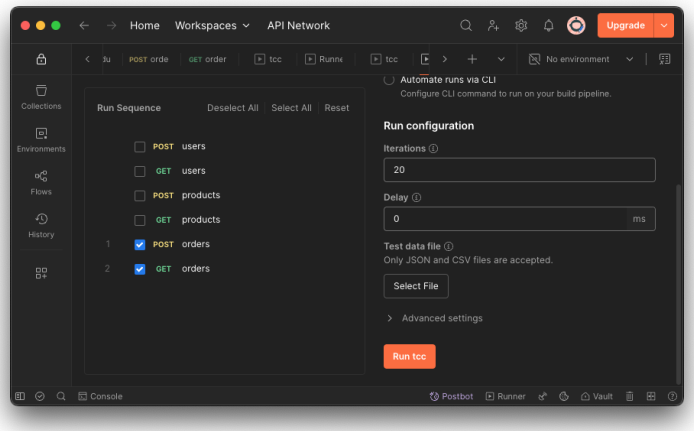


Fonte: Elaborado pelo autor

3.2.1.3 Ferramenta de Teste e Mensuração: Postman

A execução dos testes de carga e a coleta da métrica de latência foram automatizadas com o uso da ferramenta Postman (versão 11.55.1). Através de sua funcionalidade "Collection Runner", foi possível disparar sequências de 20 requisições para cada endpoint, garantindo consistência nos testes e registrando com precisão o tempo de resposta em milissegundos para cada chamada individual. A interface do "Collection Runner" do Postman, utilizada para a automação, é apresentada na Figura 4.

Figura 4 – Programa Postman sendo utilizado



Fonte: Elaborado pelo autor

3.3 Desenvolvimento dos Protótipos

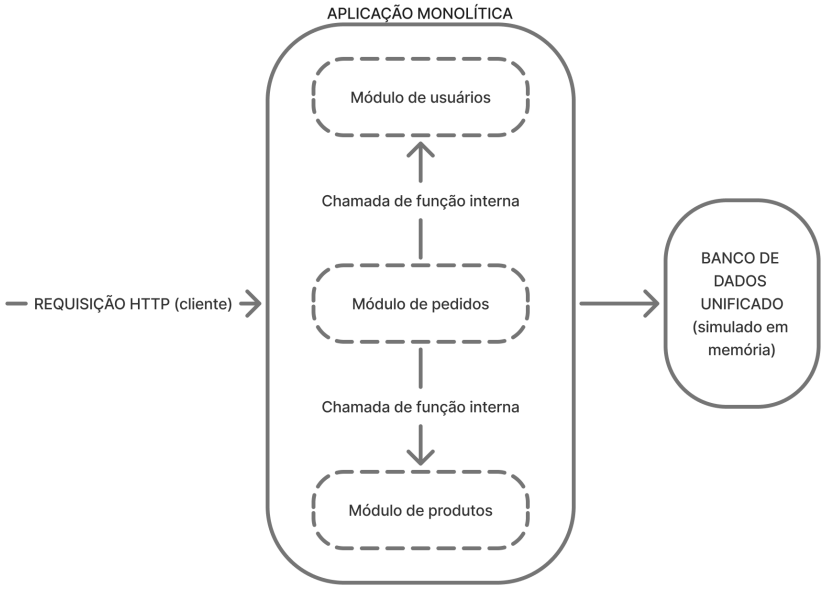
Para a comparação, foram desenvolvidos dois protótipos de software que implementam a mesma API para um sistema simplificado de e-commerce. A premissa de identidade funcional é um controle experimental mandatório para que as diferenças observadas possam ser atribuídas exclusivamente às variações arquiteturais.

Ambos os protótipos expõem endpoints para operações CRUD em três domínios: Usuários, Produtos e Pedidos. A complexidade da lógica reside na operação de criação de um pedido, que exige a orquestração e validação de dados dos outros dois domínios. Os trechos de código mais relevantes que ilustram a diferença de implementação da lógica de criação de pedidos entre as duas arquiteturas podem ser consultados no Apêndice A.

3.3.1 Monólito

A implementação monolítica consiste em um único processo Node.js/Express. Todos os domínios coexistem na mesma base de código e executam no mesmo espaço de memória. A comunicação entre os domínios (ex: a lógica de Pedidos validando um Usuário) é realizada através de chamadas de função diretas, síncronas e internas ao processo. O sistema foi containerizado em uma única imagem Docker. A Figura 5 ilustra o funcionamento dos componentes e conectores do protótipo monolítico.

Figura 5 – Diagrama de Arquitetura Monolítica do Protótipo

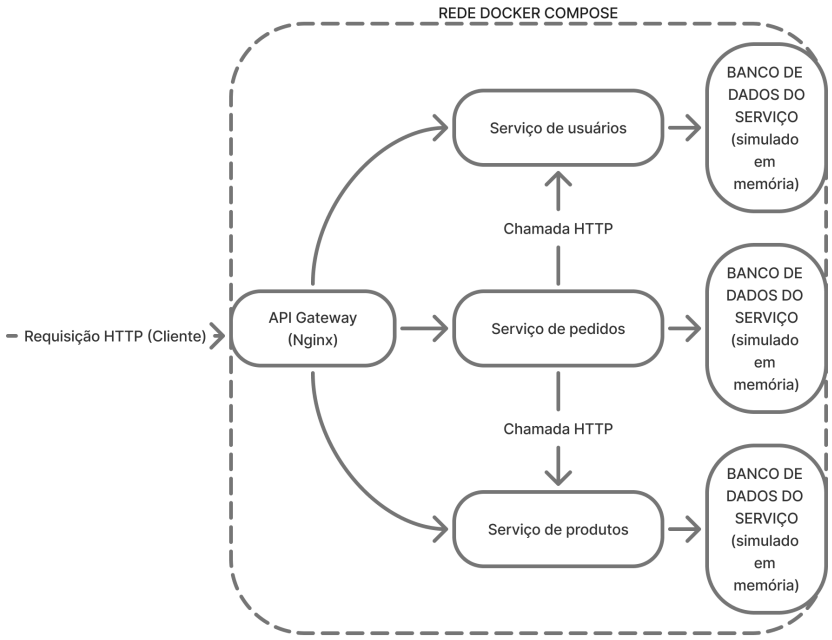


Fonte: Elaborado pelo autor

3.3.2 Microsserviços

A implementação de microsserviços decompõe o sistema em três serviços independentes, cada um em seu próprio processo Node.js/Express e containerizado em sua própria imagem Docker: servico-usuarios, servico-produtos e servico-pedidos. Um quarto contêiner, executando o Nginx, atua como um API Gateway, roteando as requisições externas para o serviço apropriado. A comunicação entre os serviços (ex: servico-pedidos validando um Usuário) é realizada através de chamadas de rede HTTP assíncronas, utilizando a biblioteca axios. A orquestração dos contêineres é gerenciada por um arquivo docker-compose.yml. A Figura 6 ilustra o funcionamento dos componentes e conectores do protótipo de microsserviços.

Figura 6 – Diagrama de Arquitetura de Microsserviços do Protótipo



Fonte: Elaborado pelo autor

3.4 Métricas de Avaliação

A comparação entre os protótipos foi fundamentada em um conjunto de métricas deliberadamente escolhidas para fornecer uma análise multidimensional dos trade-offs entre as arquiteturas, abordando diretamente as questões formuladas na Introdução. As métricas e suas respectivas justificativas são definidas a seguir:

- **Latência (Desempenho):** Definida como o tempo total de resposta de uma requisição HTTP, medido em milissegundos (ms). Esta métrica foi selecionada por ser o principal indicador de desempenho sob a perspectiva do usuário final. De forma crucial, ela permite quantificar empiricamente o impacto da **sobrecarga de rede (network overhead)** e da **comunicação inter-serviços**, testando diretamente a Questão 1, que postula uma desvantagem de desempenho para a arquitetura de microsserviços em operações de orquestração.
- **Consumo de Recursos (Custo Operacional):** Medido através do uso de CPU (em porcentagem) e de Memória RAM (em Megabytes - MB). A escolha desta métrica visa avaliar o custo de infraestrutura de cada arquitetura. A medição da RAM em repouso

permite quantificar a **sobrecarga de processo (process overhead)** de manter múltiplos contêineres ativos, validando a Questão 2. O consumo de CPU sob carga, por sua vez, revela a eficiência computacional de cada modelo ao processar requisições. Estes dados traduzem-se diretamente em custos de hospedagem e provisionamento de hardware.

- **Complexidade de Implantação (Custo de Desenvolvimento):** Avaliada qualitativamente através da contagem e análise dos artefatos de configuração (e.g., `Dockerfile`, `docker-compose.yml`). Enquanto as métricas anteriores medem o custo de *execução*, esta métrica avalia o custo de *implementação*. Ela foi escolhida para testar a Questão 3, buscando quantificar de forma objetiva a **carga cognitiva** e o esforço de configuração inicial que cada abordagem impõe à equipe de desenvolvimento.

3.5 Protocolo de Coleta de Dados

Para garantir a consistência, um protocolo rigoroso foi seguido para a coleta de dados de cada uma das 20 execuções por endpoint e por arquitetura.

1. Isolamento do Ambiente: Antes de cada bateria de testes, todos os contêineres Docker da execução anterior eram parados e removidos (docker-compose down ou docker stop/rm) para garantir um estado inicial limpo.
2. Execução da Arquitetura: O sistema a ser testado era iniciado (docker-compose up ou docker run).
3. Período de Estabilização: Aguardava-se um período de 60 segundos após a inicialização para que todos os processos se estabilizassem e o consumo de recursos em repouso pudesse ser medido.
4. Coleta em Repouso: Os valores de CPU e RAM em repouso (idle) eram registrados utilizando o comando docker stats.
5. Execução dos Testes: As 20 requisições para o endpoint em teste eram disparadas sequencialmente através do Postman Runner, sem intervalo entre cada requisição para simular uma sobrecarga artificial no servidor.
6. Registro dos Dados: A latência de cada uma das 20 requisições era registrada automaticamente pelo Postman. O pico de consumo de CPU e RAM durante a execução era observado no docker stats e registrado manualmente.
7. Finalização: Ao final da bateria, o ambiente era desligado. O processo era então repetido para a próximo arquitetura.

3.6 Análise dos Dados

Os dados brutos de latência coletados (20 amostras por cenário) foram tabulados em uma planilha eletrônica. Para cada conjunto de amostras, foram calculadas as seguintes medidas de estatística descritiva: média, mediana, desvio padrão, mínimo e máximo. Estes dados consolidados foram utilizados para gerar as tabelas e os gráficos de barras comparativas apresentados no Capítulo 4.

4 RESULTADOS

Este capítulo apresenta os dados brutos e consolidados coletados a partir da execução do protocolo experimental descrito no Capítulo 3. Os resultados são apresentados de forma objetiva e segmentada por cada uma das métricas de avaliação definidas: desempenho (latência), consumo de recursos e complexidade de implantação. A análise e interpretação destes resultados serão realizadas no capítulo subsequente.

4.1 Resultados de Desempenho (Latência)

A latência, definida como o tempo de resposta total de uma requisição HTTP, foi a principal métrica quantitativa para avaliar o desempenho de cada arquitetura. Para cada um dos cenários de teste, foram executadas 20 requisições, e os dados coletados foram consolidados através de estatística descritiva. A Tabela 1 consolida os dados de desempenho coletados no experimento.

Tabela 1 – Consolidação Estatística dos Dados de Latência (ms)

Arquitetura	Endpoint	Média	Mediana	D. Padr.	Mín.	Máx.
Monólito	GET /api/users	2.80	3.0	0.75	2	5
Monólito	POST /api/users	3.35	3.0	0.74	2	4
Microsserviços	GET /api/users	3.00	2.0	1.25	1	5
Microsserviços	POST /api/users	3.50	4.0	1.16	2	4
Monólito	GET /api/products	2.45	2.0	0.74	2	3
Monólito	POST /api/products	2.75	3.0	0.50	2	3
Microsserviços	GET /api/products	3.45	3.5	0.83	2	5
Microsserviços	POST /api/products	3.95	4.0	0.80	2	5
Monólito	GET /api/orders	2.60	2.0	0.84	2	4
Monólito	POST /api/orders	3.25	3.0	1.18	2	5
Microsserviços	GET /api/orders	2.55	3.0	0.95	2	3
Microsserviços	POST /api/orders	8.05	7.5	1.36	6	11

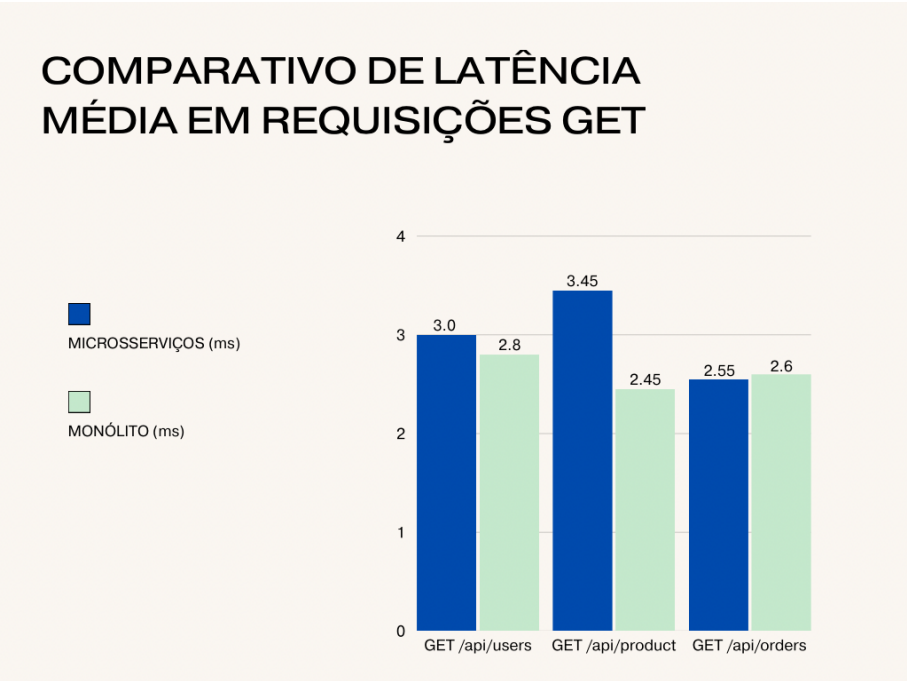
Fonte: Elaborado pelo autor

Essa tabela foi sintetizada a partir da aplicação de cálculos de estatística descritiva sobre as 20 amostras de latência registradas para cada um dos cenários de teste, conforme o protocolo definido na Metodologia. A tabela apresenta a média, mediana, desvio padrão, e os valores mínimo e máximo em milissegundos (ms), permitindo uma análise comparativa direta entre as arquiteturas.

Dentre os resultados apresentados, o achado de maior destaque é a disparidade de desempenho na operação de escrita complexa (POST /api/orders). Neste cenário, a latência

média da arquitetura de microsserviços (8.05 ms) foi 147% superior à da arquitetura monolítica (3.25 ms). Este valor evidencia um custo de desempenho substancial para a orquestração entre serviços. Para as demais operações de CRUD simples, os dados indicam uma diferença de performance marginal, embora consistentemente favorável à arquitetura monolítica. A Figura 7 mostra o gráfico de comparação para a latência média para as requisições do tipo GET.

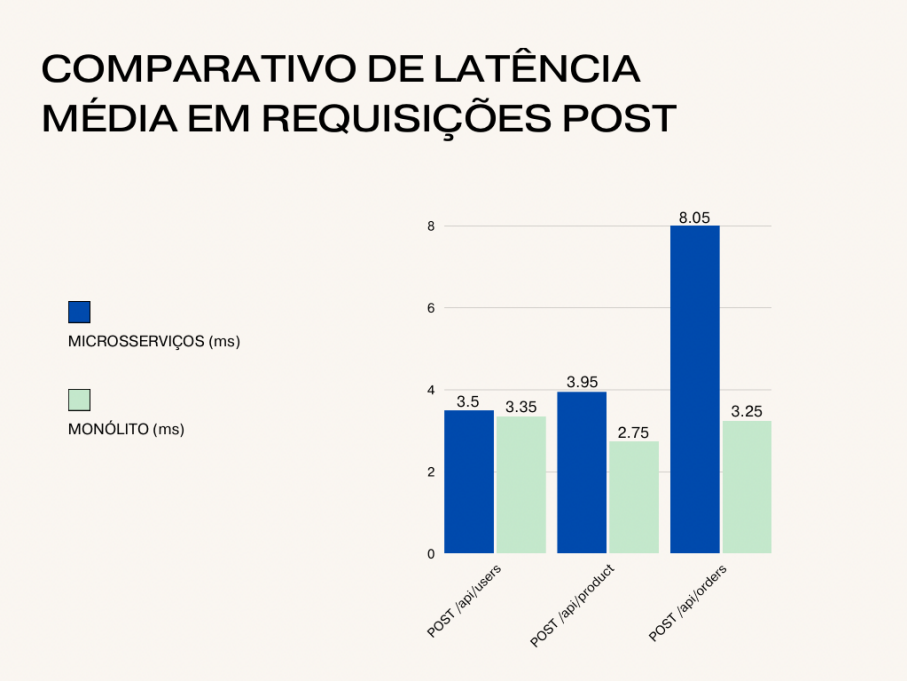
Figura 7 – Comparativo entre arquiteturas para requisições GET



Fonte: Elaborado pelo autor

Observa-se que, para as operações de leitura de dados, ambas as arquiteturas apresentaram tempos de resposta similares, com uma pequena vantagem para a arquitetura monolítica nos endpoints de users e products. A Figura 8, por sua vez, compara a latência média para as requisições do tipo POST.

Figura 8 – Comparativo entre arquiteturas para requisições POST



Fonte: Elaborado pelo autor

Neste cenário, a diferença de desempenho entre as arquiteturas torna-se mais pronunciada. Para as operações de escrita simples (users e products), a vantagem do monólito se mantém pequena. No entanto, para a operação de escrita complexa (POST /api/orders), que exige orquestração entre domínios, a latência média da arquitetura de microsserviços foi 147% superior à da arquitetura monolítica (8.05 ms contra 3.25 ms).

4.1.1 Análise dos Resultados de Desempenho

Uma inspeção detalhada da Tabela 1 revela padrões consistentes. A média de latência da arquitetura monolítica foi inferior em quase todos os cenários. Este fato, por si só, confirma a Questão 1. No entanto, a análise da mediana oferece uma visão mais apurada da distribuição dos tempos de resposta. Para a operação crítica POST /api/orders, a mediana da arquitetura de microsserviços (7.5 ms) é notavelmente inferior à sua média (8.05 ms), indicando que um pequeno número de requisições mais lentas (outliers) elevou o valor médio geral. Em contraste, na arquitetura monolítica, a média (3.25 ms) e a mediana (3.0 ms) para a mesma operação são extremamente próximas, o que sugere um comportamento de desempenho mais homogêneo e previsível.

O desvio padrão quantifica essa previsibilidade. Em quase todos os testes, o desvio padrão do monólito foi menor, indicando que seus tempos de resposta variaram menos entre

as execuções. Essa consistência é um atributo de qualidade valioso. Os dados, portanto, não mostram apenas que o monólito foi, em média, mais rápido; eles provam que seu desempenho foi também mais estável. Com estes padrões estatísticos estabelecidos, a discussão agora se volta para a interpretação arquitetural destes fenômenos.

Para operações de CRUD simples (GET e POST em /users e /products), os resultados demonstram uma diferença de latência mínima, porém consistente, conforme observado na análise estatística. Este acréscimo de latência na arquitetura de microsserviços é atribuível à sobrecarga de rede (network overhead). Conforme argumentado por (LEWIS, 2014), a introdução de chamadas de processo remotas, em oposição às chamadas de função intra-processo do monólito, é um custo fundamental dos sistemas distribuídos. No nosso experimento, a rota da requisição (Cliente $\rightarrow$ API Gateway $\rightarrow$ Serviço) é inerentemente mais longa. A utilização de um API Gateway, um padrão essencial para fornecer um ponto de entrada único e desacoplar os clientes dos serviços internos, conforme descrito por (RICHARDSON, 2018), adiciona um "salto" de rede (hop) que, embora de baixa latência em uma rede Docker otimizada, é mensurável e contribui para o tempo de resposta total.

O resultado mais expressivo do estudo foi a disparidade de 147% na latência da operação POST /api/orders. Este fenômeno fornece evidência empírica para o custo da consistência distribuída. A criação de um pedido no protótipo de microsserviços exigiu uma orquestração explícita, envolvendo múltiplas chamadas de rede sequenciais para validar e atualizar o estado em serviços distintos.

Este cenário materializa o desafio central do padrão "Database per Service", defendido por (NEWMAN, 2019). Uma vez que cada serviço detém a sua própria base de dados, uma transação de negócio que abrange múltiplos domínios não pode mais ser gerida por uma transação ACID local. Em vez disso, ela requer padrões de comunicação inter-serviços. O nosso protótipo implementou a forma mais simples de orquestração, mas o custo de cada chamada HTTP remota se acumula, explicando não apenas a média superior, mas também a maior variabilidade (desvio padrão) observada nos dados. A arquitetura monolítica, ao executar a mesma lógica através de chamadas de função na memória, evita completamente esta penalidade. O dado de 8.05 ms vs 3.25 ms não é apenas um número; é a quantificação do "imposto de rede" pago pela distribuição da lógica de negócio.

4.2 Resultados de Consumo de Recursos

O custo operacional foi avaliado medindo-se o consumo de Memória RAM e CPU dos contêineres em dois estados: em repouso (idle), 60 segundos após a inicialização; e os picos registrados durante a execução das requisições (load). A Tabela 2 apresenta os dados de consumo de recursos.

Tabela 2 – Comparativo de Consumo de Recursos por Arquitetura

Arquitetura	Estado	CPU (Total)	RAM (Total)
Monólito	Repouso (Idle)	0%	17.37 MB
Monólito	Carga (Load)	16.00%	19.00 MB
Microsserviços	Repouso (Idle)	0%	64.46 MB
Microsserviços	Carga (Load)	24.88%	77.10 MB

Fonte: Elaborado pelo autor.

Os dados indicam que a arquitetura de microsserviços, em estado de repouso, consumiu aproximadamente 271% mais Memória RAM (soma de todos os serviços) em comparação com a arquitetura monolítica. O consumo de CPU para ambas as arquiteturas em repouso foi desprezível.

Quando os sistemas foram submetidos às requisições, o consumo de memória aumentou ligeiramente para cada uma das arquiteturas, enquanto o consumo de CPU apresentou uma diferença significativa quando comparado consumo do monólito (16%) e soma dos consumos dos microsserviços (24,88%).

4.2.1 Análise dos Resultados de Consumo de Recursos

Os dados da Tabela 2, que mostram um consumo de RAM em repouso quase quatro vezes maior na arquitetura de microsserviços, são um reflexo direto da sobrecarga de processo e de containerização. Este "imposto de memória" é um fenômeno bem documentado. Cada contêiner executa não apenas a aplicação, mas uma instância do runtime Node.js e uma base do sistema operativo, resultando numa pegada de memória fixa que é multiplicada pelo número de serviços.

Sob carga, o maior consumo de CPU (24.88% vs 16.00%) é igualmente explicável. A carga computacional dos microsserviços inclui não apenas a lógica de negócio, mas também a serialização/desserialização de dados para as chamadas HTTP, a gestão das conexões de rede pelo axios e o processamento de roteamento no Nginx. Estes são ciclos de CPU que a arquitetura

monolítica simplesmente não executa. A simplicidade do monólito traduz-se diretamente em eficiência de recursos, um ponto que (NEWMAN, 2019) adverte ser um fator crítico para startups e projetos com restrições de infraestrutura.

4.3 Resultados da Complexidade de Implantação (Qualitativa)

Além das métricas quantitativas de desempenho e consumo de recursos, este estudo avaliou a complexidade de implantação de cada arquitetura. Esta análise qualitativa se baseia na decomposição do processo de configuração e execução de cada protótipo em seus componentes e passos fundamentais. Os arquivos de configuração completos, como os Dockerfiles, o docker-compose.yml e a configuração do Nginx, estão disponíveis para consulta no Apêndice B. O objetivo é quantificar os artefatos de software e a carga operacional necessários para tornar cada sistema funcional. A Tabela 3 detalha os artefatos de configuração e os comandos de alto nível necessários para a implantação de cada ambiente a partir do código-fonte.

Tabela 3 – Comparativo de Artefatos de Implantação por Arquitetura

Critério de Comparação	Monólito	Microsserviços
Projetos npm distintos	1	3
Bases de código (index.js)	1	3
Arquivos Dockerfile	1	3
Arquivos docker-compose.yml	0	1
Arquivos de config. de Proxy (Nginx)	0	1
Total de Artefatos de Configuração	**3**	**11**

Fonte: Elaborado pelo autor

A tabela dos artefatos revela que a arquitetura de microsserviços exigiu um número substancialmente maior de arquivos para sua configuração e implantação. Enquanto o ambiente monolítico foi definido por 3 artefatos principais, a arquitetura de microsserviços demandou um total de 11 arquivos distintos para atingir o mesmo estado funcional.

4.3.1 Análise dos Resultados de Complexidade de Implantação

Além da contagem de arquivos, a análise do processo revela uma diferença na carga cognitiva e na complexidade conceitual. A implantação do monólito é um processo linear e direto. Em contraste, a arquitetura de microsserviços introduz conceitos adicionais que não existem na abordagem monolítica, notadamente:

1. Orquestração de Contêineres: A necessidade de um arquivo docker-compose.yml para

gerenciar o ciclo de vida e a rede de múltiplos contêineres.

2. Roteamento de Rede (API Gateway): A necessidade de configurar um proxy reverso (Nginx) para atuar como ponto de entrada único e rotear o tráfego para os serviços internos corretos.
3. Comunicação Inter-serviços: A complexidade intrínseca de gerenciar chamadas de rede entre os serviços, que deve ser tratada no código da aplicação.

Em síntese, a análise qualitativa demonstra que a arquitetura de microsserviços exige um investimento inicial maior em configuração e conhecimento técnico. A necessidade de gerenciar os conceitos de orquestração, roteamento e comunicação inter-serviços constitui um custo de desenvolvimento e manutenção que não está presente na abordagem monolítica, mais direta. Esta complexidade adicional é o preço pago para habilitar os benefícios operacionais, como a implantação independente, que caracterizam a abordagem distribuída. (RICHARDSON, 2018) refere-se a esta complexidade como a "taxonomia dos microsserviços", onde cada padrão (como o API Gateway) resolve um problema, mas adiciona uma nova peça móvel ao sistema. O nosso estudo demonstra que, mesmo no cenário mais simples possível, essa taxonomia tem um custo de implementação prático e imediato.

4.4 Síntese e Implicações

Os resultados deste estudo, quando interpretados através da literatura, permitem a formulação de uma conclusão clara: a arquitetura de microsserviços impõe um "imposto" mensurável sobre o desempenho, o consumo de recursos e a complexidade de implantação. Em troca deste imposto, a arquitetura oferece benefícios teóricos (não medidos neste trabalho) como escalabilidade granular, resiliência e autonomia de equipes.

O presente trabalho contribui para a área ao quantificar o valor deste "imposto" num ambiente controlado. Os dados sugerem que, para sistemas de baixa complexidade ou em fases iniciais de desenvolvimento, onde a eficiência de recursos e a velocidade de entrega são primordiais, a escolha pela arquitetura monolítica é fortemente suportada por evidências empíricas.

5 CONCLUSÃO

O presente trabalho originou-se da constatação de uma lacuna na literatura técnica. Enquanto os trabalhos seminais da área descrevem de forma robusta os trade-offs teóricos entre as arquiteturas monolítica e de microsserviços (LEWIS, 2014; NEWMAN, 2019), revisões sistemáticas da literatura apontam que a validação empírica e a quantificação do desempenho dessas arquiteturas em cenários controlados ainda constituem um campo de pesquisa ativo e com demanda por mais estudos experimentais (Di Francesco *et al.*, 2019; GHOFRANI; LüBKE, 2018).

5.1 Síntese dos Resultados e Conclusões

A execução do experimento, detalhada na Metodologia e cujos dados foram apresentados no Capítulo 4, permitiu a formulação de conclusões diretas, validadas pela análise realizada no mesmo capítulo.

As conclusões desta pesquisa são válidas e devem ser interpretadas exclusivamente no escopo deste estudo. Dentro destes limites experimentais, a arquitetura monolítica provou ser objetivamente mais eficiente em todas as métricas de performance e custo operacional local. A latência de suas respostas foi consistentemente inferior, e seu consumo de recursos (CPU e RAM) foi significativamente menor em comparação com a soma dos componentes da arquitetura de microsserviços.

O achado mais significativo foi a quantificação do custo da comunicação inter-serviços. A operação de escrita complexa (POST /api/orders), que exigiu orquestração, revelou uma penalidade de desempenho de 147% para a arquitetura de microsserviços. Este dado transforma o conceito teórico de network overhead em uma métrica concreta, provando que a distribuição da lógica de negócio tem um custo de latência imediato e substancial.

Adicionalmente, a análise qualitativa da complexidade de implantação concluiu que a arquitetura de microsserviços impõe uma carga cognitiva e operacional superior, exigindo um número maior de artefatos de configuração e o domínio de conceitos de sistemas distribuídos que são abstraídos na abordagem monolítica.

Em suma, este trabalho conclui que a arquitetura de microsserviços impõe um "imposto" de complexidade e de recursos em troca de seus benefícios teóricos de escalabilidade e resiliência — benefícios estes que não foram o foco de medição deste estudo, mas cujos custos

foram claramente evidenciados.

5.2 Contribuições da Pesquisa

As contribuições deste trabalho para a área de Engenharia de Software podem ser categorizadas em três domínios:

- Contribuição Empírica: Fornece um conjunto de dados quantitativos que substancia as discussões teóricas sobre os trade-offs arquiteturais, servindo como um ponto de referência para futuras análises.
- Contribuição Metodológica: Apresenta um protocolo experimental claro e reprodutível para a comparação de arquiteturas de software, que pode ser adaptado e estendido por outros pesquisadores.
- Contribuição Prática: Oferece subsídios concretos para que arquitetos de software e equipas de desenvolvimento possam tomar decisões mais informadas na fase inicial de um projeto, ponderando os custos imediatos de uma arquitetura de microsserviços contra seus benefícios a longo prazo.

5.3 Limitações e Trabalhos Futuros

A validade destas conclusões está intrinsecamente ligada às limitações do estudo, que, por sua vez, abrem caminhos para futuras investigações. Reconhece-se que o ambiente de rede idealizado e a simplicidade da aplicação são fatores limitantes. Com base nisso, sugerem-se as seguintes direções para trabalhos futuros:

- Simulação de Ambientes de Rede Realistas: Repetir o experimento introduzindo latência e instabilidade artificial na rede entre os contêineres para simular com maior fidelidade um ambiente de produção em nuvem e medir o impacto na resiliência de cada arquitetura.
- Análise de Escalabilidade sob Carga Concorrente: Realizar testes de carga com múltiplos utilizadores simultâneos para avaliar o ponto de saturação de cada arquitetura e a eficácia da escalabilidade horizontal dos microsserviços em comparação com a escalabilidade vertical do monólito.
- Aumento da Complexidade da Lógica de Negócio: Implementar transações de negócio mais complexas, que exijam padrões de comunicação mais sofisticados entre os serviços (como coreografia baseada em eventos), para analisar o impacto na consistência de dados

e na complexidade do código.

- Análise de Métricas de Manutenibilidade: Investigar o impacto de cada arquitetura no ciclo de vida do software, medindo métricas como tempo para adicionar uma nova funcionalidade, complexidade ciclomática do código e facilidade de realização de testes de regressão.

REFERÊNCIAS

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software architecture in practice**. [S.l.]: Addison-Wesley, 2013.

BLINOWSKI, G.; OJDOWSKA, A.; PRZYBYŁEK, A. Monolithic vs. microservice architecture: A performance and scalability evaluation. **IEEE Access**, v. 10, p. 20357–20374, 2022.

BROOKS, J. F. **The Mythical Man-Month**. Reading, MA: Addison-Wesley, 1972.

Di Francesco, P.; LAGO, P.; MALAVOLTA, I. Architecting with microservices: A systematic mapping study. **Journal of Systems and Software**, v. 150, p. 77–97, 2019. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121219300019>.

FOWLER, M. **Microservice Trade-Offs**. 2015. Disponível em: <https://martinfowler.com/articles/microservice-trade-offs.html>. Acesso em: 7 mar. 2025.

FOWLER, M.; FERNANDES, A.; RICE, D.; FOEMMEL, M.; HIEATT, E.; MEE, R.; STAFFORD, R. **Padroes de Arquitetura de aplicacoes Corporativas**. [S.l.]: Bookman, 2006.

GHOFRANI, J.; LüBKE, D. Challenges of microservices architecture: A survey on the state of the practice. 05 2018.

HOSSAIN, M. D.; SULTANA, T.; AKHTER, S.; HOSSAIN, M. I.; THU, N. T.; HUYNH, L. N.; LEE, G.-W.; HUH, E.-N. The role of microservice approach in edge computing: Opportunities, challenges, and research directions. **ICT Express**, v. 9, n. 6, p. 1162–1182, 2023. ISSN 2405-9595.

LEWIS, M. F. J. **Microservices**. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 7 mar. 2025.

NEWMAN, S. **Monolith to Microservices**. [S.l.]: O'Reilly Media, 2019.

NEWMAN, S. **Building microservices: Designing fine-grained systems**. [S.l.]: O'Reilly, 2021.

PEREIRA, L. G. N. Comparativo entre arquitetura monolítica e arquitetura de microsserviços. 12 2022.

PERRY, D.; WOLF, A. Foundations for the study of software architecture. **SIGSOFT Softw. Eng. Notes**, Association for Computing Machinery, v. 17, n. 4, p. 40–52, 1992.

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>.

RICHARDSON, C. **Microservices Patterns: With examples in Java**. Manning, 2018. ISBN 9781617294549. Disponível em: <https://books.google.com.br/books?id=UeK1swEACAAJ>.

SOMMERVILLE, I. **Software engineering**. Boston: Pearson, 2011.

TAPIA, F.; MORA, M. ; FUERTES, W.; AULES, H.; FLORES, E.; TOULKERIDIS, T. From monolithic systems to microservices: A comparative study of performance. **Applied Sciences**, v. 10, n. 17, 2020. ISSN 2076-3417. Disponível em: <https://www.mdpi.com/2076-3417/10/17/5797>.

APÊNDICE A – TRECHOS DE CÓDIGO

TRECHO DE CÓDIGO DO PROTÓTIPO MONOLÍTICO

```
// POST new order
app.post("/api/orders", (req, res) => {
  const { userId, productId, quantity } = req.body;

  // No MONOLITO, a validação é uma busca direta no array em memória
  // (extremamente rápida).
  const user = users.find((u) => u.id === userId);
  const product = products.find((p) => p.id === productId);

  if (!user) {
    return res.status(404).json({ message: "Usuário não encontrado." });
  }
  if (!product) {
    return res.status(404).json({ message: "Produto não encontrado." });
  }

  if (product.stock < quantity) {
    return res.status(400).json({ message: "Estoque insuficiente." });
  }

  // No MONOLITO, a atualização do estoque é uma
  // modificação direta no objeto
  // (extremamente rápida).
  product.stock -= quantity;

  const newId =
    orders.length > 0 ?
    Math.max(...orders.map((o) => o.id)) + 1 : 1;
  const newOrder = {
```

```
    id: newId,
    user: { id: user.id, name: user.name },
    product: { id: product.id, name: product.name },
    quantity,
    totalPrice: product.price * quantity,
    status: "CRIADO",
  };
  orders.push(newOrder);

  console.log(`[MONOLITO] Pedido ${newId} criado com sucesso.`);
  return res.status(201).json(newOrder);
});
```

TRECHO DE CÓDIGO DO PROTÓTIPO DE MICROSSERVIÇOS

```
// POST new order
app.post("/", async (req, res) => {
  const { userId, productId, quantity } = req.body;

  // Usamos um bloco try...catch para lidar com erros de rede
  try {
    // 1. Validar o usuário
    console.log(`[PEDIDOS] Verificando usuário: ${userId}`);
    const userResponse = await axios.get(`${USER_API_URL}/${userId}`);
    const user = userResponse.data;

    // 2. Validar o produto e o estoque
    console.log(`[PEDIDOS] Verificando produto: ${productId}`);
    const productResponse =
        await axios.get(`${PRODUCT_API_URL}/${productId}`);
    const product = productResponse.data;

    if (product.stock < quantity) {
```

```
      return res.status(400).json(
      { message: "Estoque insuficiente." }
      );
    }

    // 3. Abater o estoque (fazendo uma chamada PATCH
    // para o serviço de produtos)
    console.log(
    `[PEDIDOS] Atualizando estoque do produto ${productId}`
    );
    await axios.patch(`${PRODUCT_API_URL}/${productId}`, {
      stock: product.stock - quantity,
    });

    // 4. Se tudo deu certo, criar o pedido
    const newId =
      orders.length > 0 ?
        Math.max(...orders.map((o) => o.id)) + 1 : 1;
    const newOrder = {
      id: newId,
      user: { id: user.id, name: user.name },
      product: { id: product.id, name: product.name },
      quantity,
      totalPrice: product.price * quantity,
      status: "CRIADO",
    };
    orders.push(newOrder);

    console.log(`[PEDIDOS] Pedido ${newId} criado com sucesso.`);
    return res.status(201).json(newOrder);
  } catch (error) {
    // Lida com erros (ex: usuário ou produto não encontrado)
```

```
    if (error.response && error.response.status === 404) {
      console.error(
        "[PEDIDOS] Erro: Usuário ou Produto não encontrado
            na chamada interna."
      );
      return res.status(404).json(
      { message: "Usuário ou Produto inválido." }
      );
    }
    // Outros erros de comunicação
    console.error(
      "[PEDIDOS] Erro de comunicação entre serviços:",
      error.message
    );
    return res
      .status(500)
      .json({ message: "Erro interno no serviço de pedidos." });
  }
});
```

APÊNDICE B – ARTEFATOS DE CONFIGURAÇÃO DOS PROTÓTIPOS

Dockerfile do protótipo monolítico

```
# Dockerfile (Monolito)

# 1. Usar uma imagem base oficial do Node.js
FROM node:18-alpine

# 2. Definir o diretório de trabalho dentro do contêiner
WORKDIR /usr/src/app

# 3. Copiar os arquivos de dependência
COPY package*.json ./

# 4. Instalar as dependências
RUN npm install

# 5. Copiar o restante do código da aplicação
COPY . .

# 6. Expor a porta que a aplicação usa
EXPOSE 3000

# 7. O comando para iniciar a aplicação
CMD [ "node", "index.js" ]
```

docker-compose.yml da arquitetura de microsserviços

```
# docker-compose.yml
version: "3.8"

services:
  servico-usuarios:
    build: ./servico-usuarios
```

```
    ports:
      - "3001:3001"

  servico-produtos:
    build: ./servico-produtos
    ports:
      - "3002:3002"

  servico-pedidos:
    build: ./servico-pedidos
    ports:
      - "3003:3003"

  nginx:
    image: nginx:stable-alpine
    ports:
      - "8080:80" # Porta de entrada para o mundo exterior
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - servico-usuarios
      - servico-produtos
      - servico-pedidos
```

Configuração do servidor Nginx (API Gateway)

```
events {}
http {
    server {
        listen 80;

        location /api/users/ {
            proxy_pass http://servico-usuarios:3001/;
        }
```

```
        location /api/products/ {
            proxy_pass http://servico-produtos:3002/;
        }

        location /api/orders/ {
            proxy_pass http://servico-pedidos:3003/;
        }
    }
}
```

Dockerfile do microsserviço de pedidos

```
# Dockerfile (Pedidos)

# 1. Usar uma imagem base oficial do Node.js
FROM node:18-alpine

# 2. Definir o diretório de trabalho dentro do contêiner
WORKDIR /usr/src/app

# 3. Copiar os arquivos de dependência
COPY package*.json ./

# 4. Instalar as dependências
RUN npm install

# 5. Copiar o restante do código da aplicação
COPY . .

# 6. Expor a porta que a aplicação usa
EXPOSE 3003

# 7. O comando para iniciar a aplicação
```

```
CMD [ "node", "index.js" ]
```

Dockerfile do microsserviço de produtos

```
# Dockerfile (Produtos)

# 1. Usar uma imagem base oficial do Node.js
FROM node:18-alpine

# 2. Definir o diretório de trabalho dentro do contêiner
WORKDIR /usr/src/app

# 3. Copiar os arquivos de dependência
COPY package*.json ./

# 4. Instalar as dependências
RUN npm install

# 5. Copiar o restante do código da aplicação
COPY . .

# 6. Expor a porta que a aplicação usa
EXPOSE 3002

# 7. O comando para iniciar a aplicação
CMD [ "node", "index.js" ]
```

Dockerfile do microsserviço de usuários

```
# Dockerfile (Usuarios)

# 1. Usar uma imagem base oficial do Node.js
FROM node:18-alpine

# 2. Definir o diretório de trabalho dentro do contêiner
```

```
WORKDIR /usr/src/app

# 3. Copiar os arquivos de dependência
COPY package*.json ./

# 4. Instalar as dependências
RUN npm install

# 5. Copiar o restante do código da aplicação
COPY . .

# 6. Expor a porta que a aplicação usa
EXPOSE 3001

# 7. O comando para iniciar a aplicação
CMD [ "node", "index.js" ]
```