# Arquitetura de Software Aplicada ao Desenvolvimento de APIs: Uma Análise de Desempenho

**Aluno:** Davi Souza Resende - daviresendedev@gmail.com
**Orientador:** Prof. Me. Wander Inácio de Souza - wanderinacio@uniformg.edu.br

## Resumo

Os sistemas modernos dependem cada vez mais da comunicação entre diferentes aplicações, e essa integração é viabilizada principalmente por meio de *interfaces* que permitem a troca estruturada de informações. Entretanto, o modo como essas *interfaces* são projetadas pode afetar significativamente o desempenho, a escalabilidade e a facilidade de manutenção das soluções desenvolvidas. Este trabalho investiga como diferentes formas de organização e estruturação de sistemas influenciam seu comportamento sob alta demanda. Para isso, foram criadas duas versões de uma mesma aplicação: uma construída de maneira simples, sem padrões de arquitetura definidos, e outra desenvolvida com práticas consolidadas de engenharia de *software*. Também foram analisados recursos adicionais, como o uso de mecanismos de *cache* e a separação entre comandos e consultas, com o objetivo de compreender seu impacto no tempo de resposta e na estabilidade do serviço resultante. A pesquisa utilizou testes de carga automatizados para avaliar o desempenho das aplicações, com monitoramento contínuo e análise dos resultados em gráficos e métricas de uso. Com base nos experimentos realizados, o estudo busca demonstrar de forma prática como um bom planejamento arquitetural contribui para a eficiência, a confiabilidade e a evolução de sistemas, fornecendo subsídios úteis tanto para profissionais quanto para pesquisadores da área.

**Palavras-chave:** Integração de sistemas, Arquitetura de *software*, Desempenho, Escalabilidade, Engenharia de *software*.

## Abstract

*Modern systems increasingly rely on communication between various applications, and it's integration is primarily achieved through interfaces to enable structured exchange of information. However, the way these interfaces are designed can significantly affect the performance, scalability, and maintainability of the developed solutions. This study investigates how different forms of system organization and architectural structure influence performance under high-demand conditions. To achieve this, two versions of the same application were developed: one built in a simple way, without defined architectural patterns, and another using consolidated software engineering practices. Additional architectural features, such as caching mechanisms and the separation between commands and queries, were also analyzed to assess their impact on response time and system stability. The research employed automated load testing to evaluate application performance, with continuous monitoring and analysis of results through visual metrics and performance graphs. Based on the experiments conducted, the study seeks to demonstrate how thoughtful architectural planning contributes to the efficiency, reliability, and evolution of systems, providing valuable insights for both professionals and researchers in the field.*

***Keywords:*** *System integration, Software architecture, Performance, Scalability, Software engineering*

## 1. Introdução

Nos últimos anos, as *interfaces* de programação de aplicações consolidaram-se como componentes fundamentais no desenvolvimento e integração de sistemas modernos. Empresas de diferentes setores as utilizam para conectar soluções, oferecer serviços a parceiros e viabilizar aplicações escaláveis voltadas ao usuário final (Pavaneli, 2023). Esse cenário faz com que a qualidade arquitetural das APIs (*Application Programming Interfaces*) assuma um papel estratégico, influenciando diretamente desempenho, escalabilidade e a capacidade de evolução dos sistemas (Lauret, 2025).

Além disso, a crescente digitalização de mercados, impulsionada por iniciativas como *open banking*, comércio eletrônico e Internet das Coisas (IoT), intensifica a dependência de soluções baseadas em APIs, que passam a operar como verdadeiros pilares da economia digital (Bomfim and de Jesus Dias Santana, 2019; Cavalcante, 2021; Ribeiro and Bagnoli, 2020). Plataformas de grande alcance, como Google, Amazon e Meta, também reforçam essa tendência ao disponibilizar ecossistemas inteiros estruturados sobre *interfaces* programáveis, ampliando ainda mais o impacto desse modelo no cotidiano tecnológico (Wang and McLarty, 2021).

Apesar dessa relevância, observa-se que implementações de APIs ainda são desenvolvidas de forma simplificada, sem considerar boas práticas de *design* ou padrões arquiteturais estabelecidos. Essa abordagem, embora prática no início, pode trazer consequências negativas à medida que o sistema cresce (Lauret, 2025). Problemas de manutenibilidade[1] tornam-se comuns, já que o código tende a acumular complexidade e dependências difíceis de gerenciar. Além disso, surgem custos ocultos relacionados à necessidade de refatorações[2] frequentes, que acabam sendo mais onerosas do que um planejamento arquitetural realizado desde o início. Esses fatores mostram que, embora uma API simples possa atender demandas iniciais, sua evolução para ambientes de maior carga e criticidade tende a ser acompanhada de limitações significativas (Jin et al., 2018).

Nesse contexto, o presente trabalho propõe investigar como diferentes escolhas arquiteturais influenciam o desempenho de APIs. A análise envolve a implementação de duas versões de uma mesma aplicação: uma versão simples, sem divisão de camadas ou padrões de projeto, e uma segunda estruturada, fundamentada em princípios de arquitetura e *design* de *software*, incluindo *Domain-Driven Design* (DDD) e os princípios SOLID, que visam promover maior modularidade[3], manutenibilidade e flexibilidade de código (Evans, 2004; Martin, 2003). Adicionalmente, exploram-se incrementos específicos, como a utilização de mecanismos de *cache* e a aplicação do padrão *Command Query Responsibility Segregation* (CQRS), de modo a verificar sua influência no tempo de resposta e *throughput*[4] sob condições

[1]Capacidade de um sistema de ser modificado, corrigido ou aprimorado com facilidade e baixo risco de introduzir erros.

[2]Processo de reestruturação do código-fonte para melhorar sua organização interna sem alterar seu comportamento externo.

[3]Propriedade que permite dividir um sistema em partes independentes, facilitando o entendimento, a manutenção e a reutilização do código.

[4]Quantidade de requisições processadas por segundo em um sistema.

de carga elevada.

A relevância deste estudo está em compreender como diferentes escolhas podem afetar o desempenho de APIs, oferecendo subsídios para decisões que conciliem eficiência, escalabilidade e custo de implementação. Ao fornecer uma análise comparativa embasada em experimentos práticos, o trabalho busca contribuir não apenas para o campo acadêmico, mas também para o ambiente profissional, apoiando organizações na adoção de soluções que equilibrem robustez técnica e viabilidade de implementação em diferentes contextos de uso.

Dessa forma, formula-se a hipótese de que arquiteturas baseadas em *Domain-Driven Design* (DDD), princípios SOLID e *Clean Architecture* tendem a apresentar maior estabilidade sob condições de alta carga. Entretanto, considera-se que essas abordagens não são isentas de limitações, como aumento da complexidade, maior esforço de implementação e possíveis impactos negativos no desempenho em cenários de baixa concorrência.

## 2. Referencial Teórico

Esta seção apresenta os fundamentos conceituais que sustentam o desenvolvimento e análise deste trabalho. São discutidos os principais conceitos relacionados à arquitetura de *software*, padrões de projeto e aos princípios de *design* que orientam a construção de aplicações robustas e escaláveis.

Por fim, são descritas as ferramentas utilizadas nos experimentos, que compõem o ambiente de testes e observabilidade, utilizados para coletar, armazenar e analisar as métricas de desempenho das aplicações.

### *2.1. Arquitetura de Software*

A arquitetura de *software* pode ser compreendida como a estrutura fundamental de um sistema, composta por seus componentes, as relações entre eles e os princípios que norteiam sua evolução ao longo do tempo. Mais do que um diagrama estático, a arquitetura representa decisões críticas que impactam diretamente o desempenho, a manutenibilidade e a escalabilidade das aplicações (Bass et al., 2021).

Segundo Rozanski and Woods (2012), a arquitetura atua como um elo entre requisitos funcionais e não funcionais, buscando atender simultaneamente às demandas de negócio e às restrições técnicas. Dessa forma, o desenho arquitetural influencia não apenas a qualidade do *software* entregue, mas também a eficiência com que equipes conseguem desenvolvê-lo e mantê-lo a longo prazo.

No contexto de aplicações baseadas em APIs, a arquitetura também assume papel estratégico. Isso porque tais sistemas necessitam ser capazes de lidar com características que envolvem alto volume de requisições, necessidade de escalabilidade e flexibilidade para integrações com outros serviços. Autores como de Souza (2022); Jin et al. (2018) destacam que arquiteturas bem definidas permitem reduzir gargalos de desempenho e facilitam a introdução de novas funcionalidades sem comprometer a estabilidade.

A literatura também destaca que a negligência arquitetural pode levar a problemas como acoplamento excessivo, dificuldades de manutenção e baixa capacidade de adaptação a cenários de carga variável. Esses problemas tendem a ser ainda mais evidentes em sistemas distribuídos, nos quais a comunicação entre serviços exige padrões bem estabelecidos (Lauret, 2025).

Assim, compreender e aplicar a arquitetura de *software* no desenvolvimento de APIs é essencial não apenas para a robustez técnica imediata, mas também para assegurar a

sustentabilidade e a evolução contínua das aplicações em contextos de negócio que exigem alta disponibilidade.

### *2.2. REST*

O *Representational State Transfer* (REST) é um estilo arquitetural proposto por Fielding (2000) em sua tese de doutorado. Ele define REST como um conjunto de *constraints* (restrições) com o objetivo de promover a interoperabilidade, simplicidade e escalabilidade na troca de informações entre clientes e servidores por meio de recursos conhecidos e operações padronizadas.

No modelo REST, cada recurso é representado por uma URI (*Uniform Resource Identifier*) e manipulado através de métodos do protocolo HTTP, como *GET*, *POST*, *PUT* e *DELETE*. Essa padronização colabora para o desacoplamento entre cliente e servidor, facilitando a integração e o reuso dos componentes em diferentes tecnologias.

Entre as características que tornam o REST amplamente adotado estão sua natureza *stateless*, na qual cada requisição contém todas as informações necessárias para ser processada, e o uso de formatos leves de troca de dados, como JSON, que reduzem a sobrecarga de rede e simplificam o consumo das APIs.

### *2.3. Princípios de Design (SOLID)*

Os princípios de *design* reunidos sob o acrônimo SOLID foram formalizados por Martin (2003) como um conjunto de práticas para melhorar a qualidade de *software* orientado a objetos. Seu objetivo central é promover sistemas mais flexíveis, de fácil manutenção e capazes de evoluir de forma sustentável ao longo do tempo. Embora não representem regras rígidas, esses princípios constituem boas práticas amplamente adotadas no desenvolvimento de serviços modernos, principalmente em contextos que demandam escalabilidade e manutenibilidade.

#### *2.3.1. Single Responsibility Principle (SRP)*

O princípio da responsabilidade única estabelece que cada classe deve possuir apenas um motivo para mudar. Em *Application Programming Interfaces*, a aplicação desse conceito evita que componentes acumulem responsabilidades múltiplas, reduzindo o acoplamento e facilitando a testabilidade. Por exemplo, separar regras de negócio da lógica de persistência garante que mudanças no banco de dados não impactem o domínio central da aplicação (Martin, 2003).

#### *2.3.2. Open/Closed Principle (OCP)*

O princípio aberto/fechado propõe que classes devem estar abertas para extensão, mas fechadas para modificação. Isso significa que novas funcionalidades devem ser incorporadas sem a necessidade de alterar código preexistente (Martin, 2003).

Em APIs, esse conceito se reflete na capacidade de evoluir o sistema, como incluir novos *endpoints*[5] ou regras de negócio, sem comprometer versões anteriores. O uso adequado de encapsulamento, por meio de *getters* e *setters*, reforça essa ideia ao permitir o controle de acesso e a validação de propriedades sem expor diretamente a estrutura interna das classes.

[5]Em uma API, um *endpoint* representa um ponto de acesso específico por meio do qual a aplicação disponibiliza suas funcionalidades ou dados.

### *2.3.3. Liskov Substitution Principle (LSP)*

O princípio da substituição de Liskov, formulado inicialmente por Barbara Liskov (1987), indica que objetos de uma subclasse devem poder substituir objetos de sua superclasse sem comprometer o funcionamento correto do serviço. Em serviços orientados a contratos, esse princípio assegura que extensões ou implementações alternativas de serviços mantenham a consistência contratual esperada pelos consumidores externos.

### *2.3.4. Interface Segregation Principle (ISP)*

O princípio da segregação de *interfaces* estabelece que contratos devem ser específicos para as necessidades de seus clientes, evitando impor a implementação de métodos que não são relevantes para eles. No contexto de APIs, esse conceito incentiva a criação de contratos mais coesos e objetivos, prevenindo o surgimento de *interfaces* que concentrem múltiplas responsabilidades e dificultam o reaproveitamento do código (Martin, 2003).

### *2.3.5. Dependency Inversion Principle (DIP)*

Por fim, o princípio da inversão de dependência recomenda que módulos de alto nível não dependam de módulos de baixo nível, mas ambos dependam de abstrações. No desenvolvimento de APIs, esse princípio é particularmente relevante ao separar camadas de domínio das implementações de infraestrutura (como acesso a banco de dados ou *frameworks*) (Martin, 2003).

A aplicação integrada dos princípios SOLID contribui para o fortalecimento da arquitetura de sistemas, tornando-a mais robusta frente a cenários de carga intensa e facilitando sua evolução contínua. Dessa forma, além de melhorar a qualidade interna do código, tais práticas impactam positivamente métricas externas, como tempo de resposta e confiabilidade pelos usuários.

## *2.4. Domain-Driven Design (DDD)*

O *Domain-Driven Design* (DDD), proposto por Evans (2004), surgiu como uma abordagem de desenvolvimento orientada ao domínio do problema, buscando alinhar a modelagem de *software* com o conhecimento especializado do negócio. Diferentemente de metodologias centradas apenas em aspectos técnicos, o DDD enfatiza a importância de compreender profundamente o contexto em que o sistema atua, utilizando essa compreensão como base para o desenho de suas entidades, serviços e regras. Assim, o domínio torna-se o núcleo da aplicação, e a arquitetura se desenvolve em torno dele.

Segundo Vernon (2013), o *DDD* propõe uma separação clara entre o modelo de domínio e as camadas da infraestrutura, aplicação e apresentação, favorecendo a independência entre os componentes da aplicação. Essa organização permite que alterações tecnológicas, como troca de banco de dados, *frameworks* ou serviços externos, não afetem a lógica de negócio central, assegurando maior manutenibilidade e flexibilidade.

Outro aspecto fundamental é o conceito de *Ubiquitous Language*, uma linguagem ubíqua compartilhada entre desenvolvedores e especialistas do domínio. Conforme destacam Evans (2004) e Marinescu (2019), o uso de uma linguagem comum reduz ambiguidades, melhora a comunicação e facilita a tradução de requisitos de negócio em modelos técnicos coerentes. Essa prática contribui diretamente para a criação de sistemas mais consistentes e alinhados aos objetivos organizacionais.

Além disso, o *DDD* incentiva a decomposição do domínio em subdomínios e contextos delimitados (*bounded contexts*), permitindo que partes do sistema evoluam de forma

independente e controlada. Essa segmentação torna possível o uso de diferentes estratégias arquiteturais conforme a complexidade e criticidade de cada componente (Marinescu, 2019).

Portanto, o *Domain-Driven Design* consolida-se como um pilar conceitual para o desenvolvimento de sistemas complexos, promovendo alinhamento entre negócio e tecnologia, isolamento de responsabilidades e evolução sustentável.

### *2.5. Clean Architecture*

A *Clean Architecture* proposta por Martin (2019), apresenta uma forma de estruturar sistemas que prioriza a independência entre camadas e a centralização das regras de negócio. Seu princípio fundamental é evitar que o sistema dependa de *frameworks*, bancos de dados ou interfaces gráficas, tratando essas tecnologias como ferramentas substituíveis.

A organização proposta é comumente representada por círculos concêntricos, onde as camadas internas concentram as políticas e regras de negócio mais estáveis, enquanto as externas tratam de mecanismos de implementação. A chamada "Regra da Dependência" estabelece que as dependências de código devem sempre apontar para dentro, em direção ao núcleo (domínio), garantindo que decisões técnicas periféricas não interfiram nas camadas mais importantes do sistema.

### *2.6. APIs e Desempenho*

O desempenho de aplicações que expõem *interfaces* de comunicação está fortemente relacionado às decisões arquiteturais que orientam seu desenvolvimento. Aspectos como a forma de organização das camadas, o grau de acoplamento entre componentes e a definição de responsabilidades influenciam diretamente métricas como tempo de resposta, estabilidade e escalabilidade (Lauret, 2025; Jin et al., 2018). Em sistemas de alta volumetria de requisições, pequenas ineficiências estruturais podem se multiplicar, comprometendo o comportamento global da aplicação.

Entre os fatores determinantes para o desempenho, destaca-se a estruturação interna do código. Implementações que centralizam múltiplas responsabilidades em um mesmo módulo tendem a dificultar a manutenção, reduzir a testabilidade e introduzir gargalos em operações críticas. Por outro lado, arquiteturas que seguem princípios de separação de responsabilidades e isolamento entre camadas favorecem a reutilização de componentes e permitem otimizações pontuais sem afetar outros componentes do sistema (Vernon, 2013; de Souza, 2022).

Fronteiras bem definidas entre domínios, aplicação e infraestrutura também exercem papel essencial no controle do desempenho. Essa separação contribui para o uso mais eficiente de recursos, diminui redundâncias e simplifica a introdução de melhorias. Além disso, possibilita a adoção de estratégias complementares de otimização voltadas ao tratamento de consultas, comandos e armazenamento temporário de dados.

Em síntese, o desempenho de uma aplicação está diretamente ligado à maturidade de suas decisões arquiteturais. Projetos que se baseiam em boas práticas de *design* tendem a apresentar maior previsibilidade, estabilidade e capacidade de escalar, mesmo sob condições de carga elevada (Martin, 2019, 2003).

Assim, compreender o impacto dessas decisões arquiteturais exige mais do que uma análise conceitual, é necessário mensurar o comportamento das aplicações sob diferentes níveis de demanda. Os testes de desempenho, especialmente os de carga, permitem observar como as arquiteturas reagem ao aumento gradual de requisições.

O teste de carga consiste em submeter o sistema a volumes progressivos de requisições até atingir o limite operacional previsto, medindo a capacidade de resposta e a estabilidade

da aplicação. Essa abordagem fornece uma base para comparar soluções arquiteturais e identificar gargalos de desempenho.

### *2.7. Incrementos Arquiteturais*

A adoção de boas práticas de arquitetura fornece uma base sólida para a construção de sistemas eficientes, mas determinadas estratégias complementares podem potencializar ainda mais o desempenho e a escalabilidade das aplicações. Entre as diversas abordagens existentes, podem ser mencionadas técnicas como o *Command Query Responsibility Segregation* (CQRS) e o uso de mecanismos de *cache*, que foram selecionadas neste estudo por seu impacto direto sobre a performance em cenários de alta demanda e integração intensiva entre serviços.

#### *2.7.1. Command Query Responsibility Segregation (CQRS)*

O *Command Query Responsibility Segregation* é um padrão arquitetural que propõe a separação entre as operações de leitura (*queries*) e escrita (*commands*) em um sistema. Essa divisão objetiva otimizar o desempenho e a escalabilidade, permitindo que cada tipo de operação seja tratado de forma independente e mais eficiente (Richter, 2024).

De acordo com Vernon (2013), essa abordagem evita que as mesmas estruturas de dados e lógicas de negócio sejam utilizadas para finalidades distintas, reduzindo o risco de gargalos e inconsistências em sistemas de alta concorrência. Na prática, a leitura pode ser otimizada para consultas rápidas e eficientes, priorizando desempenho e baixo consumo de recursos. Já as operações de escrita tendem a concentrar a aplicação das regras de negócio e mecanismos de validação, garantindo consistência e integridade transacional.

Esse padrão se mostra particularmente relevante em aplicações onde o volume de leitura costuma ser maior do que o de escrita. Em muitos sistemas, o número de consultas pode superar em múltiplos o volume de atualizações. Nesses cenários, a separação dos fluxos permite otimizar o caminho mais exigido, reduzindo latência e aumentando a estabilidade sob alta demanda (Richter, 2024).

Além dos ganhos de desempenho, o CQRS favorece a clareza arquitetural e facilita a manutenção, reduzindo contenção de acesso aos recursos e tornando o comportamento do sistema mais previsível em contextos de grande escala e frequência de consultas.

#### *2.7.2. Mecanismos de Cache*

O uso de *cache* é uma das estratégias mais eficazes para reduzir o tempo de resposta em aplicações que executam operações repetitivas ou consultas intensivas. O conceito consiste em armazenar temporariamente os resultados de requisições, permitindo que futuras chamadas sejam atendidas de forma mais rápida, sem a necessidade de realizar consultas em banco de dados ou refazer processamentos custosos (Misturini, 2021).

Segundo Misturini (2021), o *cache* atua como uma camada intermediária entre a aplicação e suas fontes de dados, contribuindo para a redução do consumo de recursos e a melhoria da experiência do usuário. Em sistemas baseados em APIs, essa prática é especialmente benéfica em *endpoints* de leitura com alto volume de acesso, onde os mesmos dados são requisitados com frequência.

Contudo, sua implementação deve ser feita com cautela. É essencial definir políticas de atualização e expiração adequadas, evitando que informações desatualizadas sejam entregues ao consumidor. Em arquiteturas bem planejadas, o *cache* é utilizado de forma seletiva, priorizando operações críticas e dados com baixa taxa de alteração.

Portanto, essa técnica representa uma solução prática e eficiente para otimização de desempenho em APIs.

### *2.8. Docker e Containers*

O Docker é uma plataforma de virtualização em nível de sistema operacional que permite empacotar aplicações e suas dependências em unidades isoladas denominadas *containers*. Essa abordagem garante portabilidade, reprodutibilidade e isolamento entre ambientes, evitando incompatibilidades causadas por diferenças de configuração entre máquinas (Merkel, 2014).

Cada *container* executa de forma independente, compartilhando o mesmo kernel do sistema operacional, mas mantendo ambientes de execução isolados. Isso proporciona desempenho superior ao de máquinas virtuais tradicionais, já que não existe sobrecarga de múltiplos sistemas operacionais em execução simultânea (Boettiger, 2014).

### *2.9. Ferramentas de Teste e Observabilidade*

A análise de desempenho de aplicações exige não apenas o desenvolvimento de cenários de teste, mas também mecanismos que permitam observar e interpretar o comportamento do sistema sob diferentes condições. Ferramentas de teste e observabilidade possuem papel essencial nesse processo, pois possibilitam a identificação de gargalos, a validação de hipóteses arquiteturais e a mensuração do impacto de otimizações no ambiente de execução. Em conjunto, elas fornecem uma visão abrangente da eficiência e da estabilidade das aplicações, permitindo que dados experimentais sustentem análises comparativas e conclusões fundamentais.

#### *2.9.1. K6*

O K6 é uma ferramenta de código aberto voltada para testes de carga e desempenho em aplicações web e APIs, desenvolvida para ser simples, automatizável e facilmente integrada a fluxos de desenvolvimento contínuo. Seu diferencial está na definição de cenários de teste por meio de *scripts* escritos em JavaScript, permitindo a simulação de usuários virtuais (*virtual users* – VUs) que executam operações simultâneas em grande escala (Labs, 2025b).

Essa flexibilidade na definição de cenários torna os testes mais realistas e adaptáveis a diferentes contextos de uso. Durante os testes, o K6 coleta métricas, como tempo médio de resposta, latência, taxa de requisições por segundo (RPS) e percentuais de sucesso ou falha. Assim possibilitando avaliar a capacidade da aplicação em lidar com diferentes níveis de concorrência.

#### *2.9.2. InfluxDB*

O InfluxDB é um banco de dados especializado em séries temporais, projetado para armazenar e consultar métricas que variam ao longo do tempo, como taxas, tempos de execução e consumo de recursos. Segundo Data (2025), o sistema foi desenvolvido para lidar com gravações contínuas e consultas de alto volume, sendo amplamente utilizado em cenários de monitoramento, telemetria e análise de desempenho.

Diferentemente de bancos de dados relacionais tradicionais, o InfluxDB é otimizado para operações que envolvem grandes quantidades de dados gerados de forma contínua. Sua estrutura interna é baseada em um modelo *time series*, no qual cada registro é associado a um carimbo temporal (*timestamp*), facilitando a execução de consultas analíticas e agregações em intervalos específicos (Data, 2025).

Além disso, ele possui capacidade de integração com outras aplicações por meio de comunicação em rede, o que o torna adequado para ambientes distribuídos.

### *2.9.3. Grafana*

O Grafana é uma plataforma de visualização e análise de dados amplamente utilizada para o monitoramento de sistemas e aplicações. Segundo Labs (2025a), ela permite a criação de painéis interativos que transformam dados numéricos em representações gráficas dinâmicas, facilitando a visualização e interpretação de métricas de desempenho.

O uso combinado de ferramentas como K6, InfluxDB e Grafana constitui uma infraestrutura completa para avaliação e monitoramento de desempenho. Essa integração possibilita não somente a coleta precisa de métricas, mas também a análise visual e contínua do comportamento das aplicações, fornecendo uma base sólida para a condução de experimentos comparativos e a validação de hipóteses arquiteturais.

## 3. Estado da Arte

O estado da arte tem como objetivo apresentar pesquisas que abordam temas relacionados ao desempenho e à arquitetura de aplicações baseadas em *interfaces* de comunicação, destacando metodologias e resultados que fundamentam a proposta deste trabalho. Para isso, foram analisadas referências que discutem o impacto de decisões arquiteturais, a comparação entre tecnologias e a utilização de ferramentas de teste e observabilidade em ambientes de alta demanda.

### *3.1. Performance Impact of the CQRS Pattern in C# Web APIs*

O estudo de Richter (2024) avaliou o impacto do padrão *Command Query Responsibility Segregation* (CQRS) no desempenho de APIs desenvolvidas em C#, comparando uma implementação tradicional a duas variações do padrão, uma construída manualmente e outra utilizando a biblioteca MediatR. Por meio de testes de carga conduzidos com o K6, o autor observou que, embora o padrão tenha proporcionado melhorias pontuais em alguns *endpoints*, os resultados gerais não indicaram ganhos significativos de desempenho em relação à arquitetura tradicional.

Segundo Richter (2024), o principal benefício do CQRS está na separação entre operações de leitura e escrita, que favorece organização, manutenção e escalabilidade, mais do que velocidade de execução. O autor sugere, contudo, que o padrão pode apresentar resultados superiores quando aplicado em conjunto com outras abordagens de *design*, como *Domain-Driven Design* (DDD) e *Event Sourcing* (ES), especialmente em sistemas complexos.

Essas observações reforçam a motivação do presente trabalho, que busca avaliar o impacto combinado de práticas como CQRS, DDD, SOLID e *cache* no desempenho e na escalabilidade de APIs.

### *3.2. Performance Comparison of Java EE and ASP.NET Core Technologies for Web API Development*

O estudo comparativo conduzido por Kronis and Uhanova (2018) analisou o desempenho de APIs desenvolvidas nas plataformas Java EE e ASP.NET *Core*, considerando aspectos como tempo de processamento, estabilidade e consumo de recursos. Foram implementados cenários idênticos em ambas as tecnologias, de modo a avaliar a eficiência de execução em condições semelhantes de carga.

Os resultados indicaram que, embora o ASP.NET *Core* tenha apresentado tempos de processamento ligeiramente inferiores em algumas medições, as diferenças gerais de desempenho entre as duas plataformas foram irrelevantes, evidenciando que ambas são

capazes de oferecer respostas rápidas e estáveis. O estudo também observou que a maior influência sobre o desempenho não está na escolha da tecnologia em si, mas sim na qualidade do código e na eficiência dos algoritmos empregados.

Além disso, os autores observam que o ambiente de execução e a configuração dos servidores, como o uso de *Kestrel* no ASP.NET e *GlassFish* no Java EE, podem impactar os resultados, e sugerem investigações futuras com diferentes sistemas operacionais e métricas adicionais de consumo de recursos. Em síntese, a pesquisa reforça a ideia de que o desempenho de uma API depende mais de boas práticas de implementação do que da linguagem ou do *framework* utilizado, conclusão que se alinha à perspectiva adotada neste trabalho.

*3.2.1. Validação de Carga em Testes de Desempenho de APIs de Cadastro de Usuários*

O estudo desenvolvido por Chimuco and Barbosa (2024) teve como objetivo avaliar o comportamento de uma API de cadastro de usuários sob diferentes níveis de carga, utilizando o conjunto de ferramentas K6, InfluxDB e Grafana em um ambiente conteinerizado com Docker. A pesquisa concentrou-se em analisar a estabilidade, a taxa de sucesso e a latência das requisições durante cenários de estresse controlado, com variação progressiva do número de usuários virtuais.

Os resultados mostraram que a API manteve desempenho satisfatório até o limite de cem usuários simultâneos, com tempos de resposta considerados adequados e taxa de sucesso de 100%. Entretanto, à medida que a carga aumentou, observou-se elevação significativa na latência, evidenciando a necessidade de otimizações para cenários de maior concorrência.

O trabalho de Chimuco and Barbosa (2024) reforça a importância da observabilidade e dos testes de carga na análise de desempenho de APIs, destacando como a coleta e visualização de métricas em tempo real podem direcionar melhorias de eficiência e confiabilidade em ambientes de alta demanda. Esses resultados reforçam a relevância do uso combinado dessas ferramentas, que também são adotadas neste trabalho para mensurar e comparar o desempenho de diferentes arquiteturas de API.

A partir da análise dos trabalhos relacionados, observa-se que as pesquisas existentes concentram-se, em sua maioria, na avaliação pontual de padrões arquiteturais específicos, na comparação entre tecnologias ou na validação de desempenho de APIs em cenários isolados de carga. Embora tais estudos apresentem contribuições relevantes, nota-se a ausência de uma abordagem comparativa que investigue, de forma integrada, o impacto de diferentes decisões arquiteturais sobre desempenho, estabilidade e escalabilidade em um mesmo sistema. Nesse contexto, o presente trabalho diferencia-se ao propor a comparação direta entre uma API básica e uma API estruturada, avaliando o comportamento de ambas sob diferentes níveis de carga e analisando os efeitos combinados da adoção de boas práticas arquiteturais, como DDD, SOLID, *Clean Architecture*, CQRS e mecanismos de *cache*.

## 4. Metodologia

Esta seção descreve os procedimentos utilizados para o desenvolvimento, implementação e avaliação das aplicações utilizadas neste estudo. São apresentados o desenho experimental, a forma de implementação das APIs, a configuração do ambiente de testes e os cenários executados.

A metodologia adotada tem caráter quantitativo e experimental, buscando garantir reprodutibilidade e comparabilidade entre as diferentes abordagens arquiteturais analisadas.

### *4.1. Desenho Experimental*

O presente estudo tem como propósito analisar o impacto de decisões arquiteturais no desempenho e na estabilidade de aplicações baseadas em *interfaces* de comunicação. Para isso, foram desenvolvidas duas versões de uma mesma aplicação: uma denominada API Básica, implementada de forma simplificada, sem camadas de abstração ou princípios de design, e outra denominada API Estruturada, elaborada com base em práticas consolidadas de engenharia de *software*, como *Domain-Driven Design*, princípios SOLID e *Clean Architecture*.

De modo a comparar o comportamento dessas abordagens sob condições idênticas de carga, foram avaliadas métricas incluindo latência média, taxa de requisições processadas por segundo (RPS) e estabilidade sob concorrência elevada.

A metodologia adotada segue um modelo quantitativo e experimental, em que as duas APIs são submetidas a testes automatizados de carga utilizando a ferramenta K6, com os resultados sendo coletados e armazenados no InfluxDB para posterior visualização e análise no Grafana.

O estudo também contempla incrementos arquiteturais graduais na API Estruturada, com a inclusão dos padrões CQRS e *cache*, permitindo observar de forma incremental como cada aprimoramento impacta na aplicação resultante.

Com a metodologia e o ambiente de teste definido, a etapa seguinte descreve a implementação das aplicações.

### *4.2. Implementação das Aplicações*

Esta subseção descreve a implementação das aplicações utilizadas nos experimentos, apresentando inicialmente as funcionalidades expostas, o modelo de dados adotado e, em seguida, as características arquiteturais de cada versão desenvolvida. O objetivo é detalhar os aspectos técnicos relevantes que fundamentam a comparação entre a API Básica e a API Estruturada, assegurando clareza quanto às similaridades funcionais e às diferenças arquiteturais analisadas ao longo do estudo.

#### *4.2.1. Endpoints e Modelo de Dados*

Ambas as aplicações foram implementadas na linguagem C#, seguindo o padrão REST, o que garante comunicação padronizada entre cliente e servidor por meio de recursos e métodos HTTP.

As versões desenvolvidas, API Básica e API Estruturada, possuem o mesmo conjunto de funcionalidades expostas por meio de dois *endpoints* principais, responsáveis pelo cadastro e pela consulta de itens no banco de dados:

- **POST /item**: recebe um objeto em formato *JavaScript Object Notation* (JSON) contendo as informações do item e realiza a inserção no banco de dados.
- **GET /item/{id}**: recupera as informações de um item previamente cadastrado a partir do seu identificador único.

Para garantir imparcialidade entre os experimentos, cada interface opera em uma base de dados independente, evitando interferências em métricas de acesso e conexões simultâneas. Apesar disso, ambas as bases utilizam o mesmo esquema de dados, de forma que os resultados obtidos possam ser comparados de maneira justa e controlada.

O modelo de dados adotado é composto por uma única tabela denominada *items*, apresentada na Figura 1, utilizada como estrutura comum para as operações de cadastro e consulta.

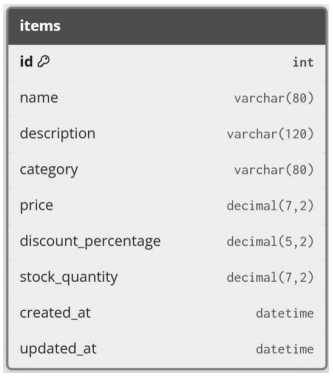


Figura 1: Modelo de dados da tabela *items* utilizada nas aplicações.

**Fonte:** Elaboração própria.

A padronização da estrutura do banco de dados assegura que as variações de desempenho observadas resultem exclusivamente das diferenças arquiteturais entre as aplicações.

### *4.2.2. Arquitetura da API Básica*

A API Básica foi desenvolvida com o propósito de representar uma implementação direta e simplificada, sem divisão em camadas ou aplicação de padrões arquiteturais. Todo o processamento, incluindo validação, regras de negócio e persistência, é concentrado em uma única camada de controle, resultando em uma estrutura linear e dependente, na qual os componentes de código estão acoplados.

Esse modelo, apesar de simples de implementar, tende a dificultar a manutenção e a evolução da aplicação à medida que novas funcionalidades são incorporadas. A ausência de separação entre responsabilidades resulta em modificações em partes do código impactando outras, comprometendo a escalabilidade e aumentando a probabilidade de falhas (Martin, 2003).

As rotas de cadastro e consulta interagem diretamente com o banco de dados MySQL, sem a mediação de camadas intermediárias como serviços, repositórios ou objetos de transferência de dados (DTOs).

A Figura 2 apresenta a estrutura geral da API Básica.

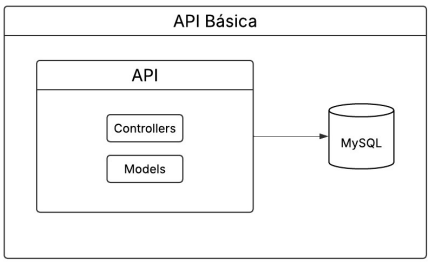


Figura 2: Estrutura simplificada da API Básica.

**Fonte:** Elaboração própria.

*4.2.3. Arquitetura da API Estruturada*

A API Estruturada foi desenvolvida para representar uma aplicação orientada a manutenibilidade, extensibilidade e robustez. Esta versão adota os padrões *Domain-Driven Design*, os princípios SOLID e conceitos de *Clean Architecture*, buscando baixo acoplamento e alta coesão entre componentes.

A solução é organizada em cinco camadas:

- **API**: expõe os *endpoints* HTTP e trata a entrada/saída dos dados.
- ***Application*:** orquestra casos de uso e mapeia DTOs em modelos de domínios.
- ***Domain*:** concentra entidades, regras de negócio e serviços de domínio.
- ***Data*:** encapsula o acesso ao banco de dados MySQL por repositórios e abstrações de persistência, mantendo o domínio independente de detalhes de infraestrutura.
- **Infra:** responsável pelo *container* de injeção de dependências.

Essa divisão permite evoluir cada componente de modo independente dos demais. O uso de injeção de dependências facilita a substituição de implementações sem alterar o código de domínio, além de facilitar a implementação de testes.

A Figura 3 apresenta a estrutura geral da API Estruturada, evidenciando o fluxo de comunicação entre as camadas e a interação indireta com o banco de dados por meio de repositórios e serviços de domínio.

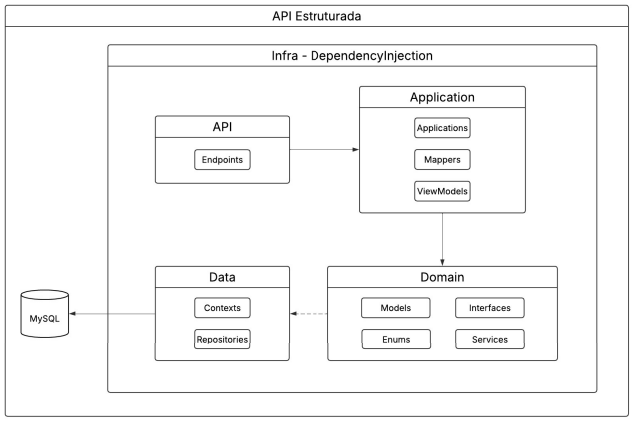


Figura 3: Arquitetura em camadas da API Estruturada.

**Fonte:** Elaboração própria.

A API Estruturada apresenta um fluxo de execução mais extenso e composto por múltiplas etapas internas, o que pode resultar em aumento da complexidade inicial de implementação.

Contudo, a organização adotada resulta em uma arquitetura mais estável, modular e de fácil manutenção.

*4.3. Configuração do Ambiente de Testes*

Os experimentos foram conduzidos em um ambiente totalmente conteinerizado, com o objetivo de garantir isolamento, reprodutibilidade e controle das variáveis externas. Cada componente foi executado em *containers* Docker, assegurando uma configuração padronizada de rede e ambiente, de modo a reduzir interferências externas e manter a comparabilidade entre os testes.

No que se refere à alocação de recursos computacionais, alguns serviços do ambiente foram executados com limites explícitos de CPU e memória, definidos por meio das diretivas de *deploy resources* do Docker. O banco de dados MySQL e as ferramentas de observabilidade e teste (InfluxDB, Grafana e K6) tiveram limites configurados, variando entre 1 e 2 núcleos de CPU e entre 512 MB e 2 GB de memória, de forma a reduzir interferências entre os componentes e evitar a saturação de recursos críticos durante a execução dos testes.

O K6 foi utilizado como ferramenta de geração de carga, responsável por simular o comportamento de múltiplos usuários virtuais (*Virtual Users* – VUs) realizando requisições aos *endpoints* das APIs. Durante a execução dos testes, a ferramenta coleta automaticamente métricas de desempenho e as transmite ao InfluxDB por meio de *output* nativo, que registra as informações em formato de séries temporais.

O InfluxDB atua como repositório central dessas métricas, permitindo que o Grafana as consuma diretamente para visualização.

A configuração do ambiente foi projetada para reduzir interferências externas e simular condições realistas de uso. As aplicações, o banco de dados e as ferramentas de monitoramento foram implantadas em *containers*, interconectados por uma mesma rede virtual. Essa abordagem garante que as diferenças observadas nos resultados experimentais refletissem exclusivamente o impacto das decisões arquiteturais adotadas.

As métricas observadas durante os testes foram:

- ***Peak* RPS**: representa o maior valor de requisições por segundo atingido.
- ***Max Response Time***: corresponde ao maior tempo de resposta registrado entre todas as requisições.
- **HTTP *Failures***: contabiliza o número de requisições que resultaram em falha.
- ***Average Response Time***: indica o tempo médio entre o envio da requisição e o recebimento da resposta.
- ***Requests Made***: total de requisições enviadas durante a execução do teste.

Essas métricas foram selecionadas por refletirem a capacidade de processamento, estabilidade e resiliência das aplicações frente a diferentes níveis de carga.

## 5. Resultados Experimentais

Esta seção apresenta os resultados obtidos a partir da execução dos testes de desempenho realizados com as duas implementações propostas. São descritos os cenários avaliados, as métricas observadas e a análise comparativa entre as abordagens arquiteturais.

### *5.1. Execução dos Testes de Desempenho*

Os testes de desempenho foram conduzidos com o objetivo de avaliar a influência das decisões arquiteturais sobre o comportamento e a estabilidade das aplicações em diferentes condições de carga. Para isso, foram definidos múltiplos cenários experimentais.

#### *5.1.1. Cenários de Teste*

Foram executados seis cenários de teste:

1. **POST (100 VUs por 1 minuto)**: cenário de escrita com carga moderada, em que 100 usuários virtuais executam requisições de inserção de dados. O objetivo é coletar métricas em uma situação estável e controlada.
2. **POST (1000 VUs por 1 minuto)**: operação de escrita sob carga intensa, destinada a avaliar a resiliência das aplicações frente a um alto volume simultâneo de inserções.
3. **POST com rampa de carga**: cenário dinâmico de escrita, com aumento gradual do número de usuários virtuais de 50 até 2000, simulando picos de tráfego reais. O objetivo é identificar pontos críticos de desempenho durante o crescimento e redução da carga.
4. **GET (100 VUs por 1 minuto)**: cenário de leitura com carga moderada, projetado para avaliar o tempo de resposta e a estabilidade das consultas em um ambiente de uso regular.
5. **GET com CQRS**: cenário de leitura utilizando o padrão CQRS, onde as operações de leitura são isoladas das de escrita. O teste visa analisar o impacto da separação de responsabilidades.
6. **GET com CQRS e *Cache***: cenário de leitura otimizado, combinando a segregação de consultas (CQRS) com o uso de *cache* em memória. Essa configuração avalia o ganho de desempenho obtido pela reutilização de resultados previamente processados, reduzindo o acesso direto ao banco de dados.

Cada execução foi realizada em ambiente isolado, com configurações idênticas de *hardware* e rede, e os bancos de dados foram restaurados a cada teste, assegurando consistência e comparabilidade entre os resultados.

Ainda que esse controle experimental tenha garantido condições equivalentes para comparação entre as implementações, o contexto adotado apresenta limitações relacionadas ao uso de contêineres Docker sem um mecanismo de orquestração. Os limites de CPU e memória foram aplicados principalmente aos serviços de infraestrutura, como o banco de dados e as ferramentas de monitoramento, enquanto as aplicações avaliadas não possuíam restrições explícitas de recursos. Dessa forma, os resultados obtidos devem ser interpretados como uma análise comparativa do comportamento das arquiteturas em condições controladas, não sendo necessariamente representativos de todos os cenários encontrados em ambientes reais de produção.

#### *5.1.2. Resultados e Análises Comparativas*

Os resultados, obtidos a partir dos testes realizados, evidenciam o comportamento contrastante entre as duas abordagens, em diferentes níveis de carga e cenários de operação.

*Cenário 1*

Neste teste, a API Básica atingiu um total de 107.148 requisições processadas, enquanto a API Estruturada realizou 76.771. O *peak* RPS também foi superior na versão simples, reflexo de sua execução linear e fluxo reduzido. O tempo médio de resposta foi de 55,39 ms na Básica, contra 77,49 ms na Estruturada, confirmando o impacto do maior número de camadas. Esses resultados comparativos estão descritos na Tabela 1.

Entretanto, a análise temporal revela comportamento mais instável na versão simples, com oscilações bruscas na latência, enquanto a Estruturada manteve curvas mais uniformes e previsíveis ao longo do teste. Essa estabilidade inicial sugere uma arquitetura mais resiliente, ainda que com custo de processamento ligeiramente maior.

Tabela 1: Resultado do cenário 1 - POST (100 VUs por 1 minuto).

| **Métrica** | **API Básica** | **API Estruturada** |
|---|---|---|
| Peak RPS | 2.913 | 1.390 |
| Max Response Time | 2,86 s | 476 ms |
| HTTP Failures | 0 | 0 |
| Average Response Time | 55,39 ms | 77,49 ms |
| Requests Made | 107.148 | 76.771 |

**Fonte:** Elaboração própria.

*Cenário 2*

No segundo cenário, os resultados se invertem em relação ao anterior. A API Básica apresentou 1.174 requisições, todas resultando em erro, decorrente do esgotamento do *pool* de conexões do MySQL. Esse problema ocorreu quando múltiplas requisições simultâneas tentaram abrir novas conexões sem reutilização, levando a falhas consecutivas em todas as tentativas de escrita.

Já a API Estruturada, ao empregar repositórios e injeção de dependências, manteve o gerenciamento de conexões de forma eficiente, evitando bloqueios e mantendo operação estável durante toda a execução.

O tempo médio de resposta na versão simples atingiu 50,74 s, enquanto a versão Estruturada registrou 675,37 ms. A diferença também se refletiu no volume de processamento: a arquitetura estruturada completou 88.772 requisições, superando amplamente as 1.174 da versão simples.

Além disso, o *Peak* RPS foi de 331 (Básica) contra 1.985 (Estruturada), e o tempo máximo de resposta atingiu 60 s (Básica) versus 903 ms (Estruturada). Esses resultados comparativos estão apresentados na Tabela 2.

Evidenciando a superioridade da arquitetura modular em cenários de alta concorrência.

Tabela 2: Resultado do cenário 2 – POST (1.000 VUs por 1 minuto).

| **Métrica** | **API Básica** | **API Estruturada** |
|---|---|---|
| Peak RPS | 331 | 1.985 |
| Max Response Time | 60 s | 903 ms |
| HTTP Failures | 1.174 | 0 |
| Average Response Time | 50,74 s | 675,37 ms |
| Requests Made | 1.174 | 88.772 |

**Fonte:** Elaboração própria.

*Cenário 3*

Nesse teste, ambas as APIs processaram volumes semelhantes de requisições totais. Porém, a API Básica apresentou 8.279 falhas e tempo máximo de resposta de 16,5 s, indicando forte degradação sob picos de demanda. Em contrapartida, a API Estruturada manteve operação estável, sem falhas registradas, e tempo máximo de resposta de 2,25 s. Esses resultados comparativos estão presentes na Tabela 3.

Tabela 3: Resultado do cenário 3 – POST com Rampa de Carga (50 a 2.000 VUs).

| **Métrica** | **API Básica** | **API Estruturada** |
|---|---|---|
| Peak RPS | 2.062 | 1.140 |
| Max Response Time | 16,5 s | 2,25 s |
| HTTP Failures | 8.279 | 0 |
| Average Response Time | 954,36 ms | 1,12 s |
| Requests Made | 117.745 | 117.211 |

**Fonte:** Elaboração própria.

A análise do gráfico de RPS *Over Time*, disponível na Figura 4, apresenta a taxa de requisições processadas por segundo ao longo do tempo de execução do teste, reforça esse comportamento. À esquerda, é possível observar o desempenho da API Básica, caracterizado por irregularidade e quedas bruscas na taxa de requisições. Já à direita, a API Estruturada demonstra variação consideravelmente menor de *throughput*. Essa consistência demonstra maior previsibilidade e controle sob variação de carga.

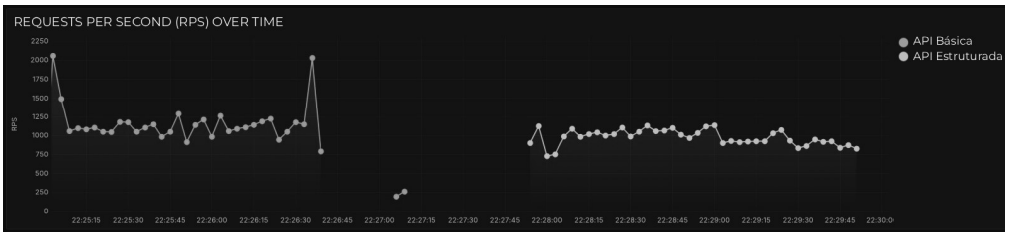


Figura 4: Resultado do cenário 3 – POST com Rampa de Carga (50 a 2.000 VUs): Gráfico RPS *Over Time*.

**Fonte:** Elaboração própria.

*Cenário 4*

Os resultados demonstraram desempenho consistente em ambas as implementações, sem falhas registradas. A API Básica apresentou tempo médio de resposta de 11,63 ms, pouco

inferior ao da Estruturada de 15,16 ms, diferença esperada devido ao fluxo de execução mais direto na versão simples. Estes resultados estão descritos na Tabela 4.

Tabela 4: Resultado do cenário 4 – GET (100 VUs por 1 minuto).

| **Métrica** | **API Básica** | **API Estruturada** |
|---|---|---|
| Peak RPS | 100 | 100 |
| Max Response Time | 48 ms | 108 ms |
| HTTP Failures | 0 | 0 |
| Average Response Time | 11,63 ms | 15,16 ms |
| Requests Made | 6.000 | 5.945 |

**Fonte:** Elaboração própria.

*Cenário 5*

O teste foi executado ao longo de 2 minutos, com variação de carga iniciando em 50 usuários virtuais, atingindo picos de 1.000 e retornando a 50, a fim de observar o comportamento das arquiteturas sob diferentes intensidades de acesso.

Os resultados demonstraram ganhos de desempenho na versão Estruturada. O tempo médio de resposta foi de 12,09 ms, 60% inferior ao registrado pela Básica, de 30,24 ms. Esses resultados comparativos estão apresentados na Tabela 5.

Tabela 5: Resultado do cenário 5 – GET com CQRS.

| **Métrica** | **API Básica** | **API Estruturada** |
|---|---|---|
| Peak RPS | 943 | 957 |
| Max Response Time | 284 ms | 218 ms |
| HTTP Failures | 1 | 0 |
| Average Response Time | 30,24 ms | 12,09 ms |
| Requests Made | 41.945 | 42.665 |

**Fonte:** Elaboração própria.

A versão sem CQRS apresentou picos acentuados de latência, enquanto a Estruturada, ao isolar os fluxos de leitura, manteve comportamento uniforme e previsível, confirmando a eficiência do padrão em cenários dominados por consultas.

O gráfico de tempo médio de resposta ao longo do tempo (*Average Response Time Over Time*), disponível na Figura 5, reforça visualmente essa diferença. À esquerda, observa-se o comportamento da API Básica, que apresenta uma variação acentuada e irregular, com picos de latência significativos ao longo da execução. Já à direita, a API Estruturada demonstra pequenas oscilações, mas mantém uma variação consideravelmente menor, evidenciando um comportamento mais previsível e consistente, mesmo durante os períodos de pico de requisições.

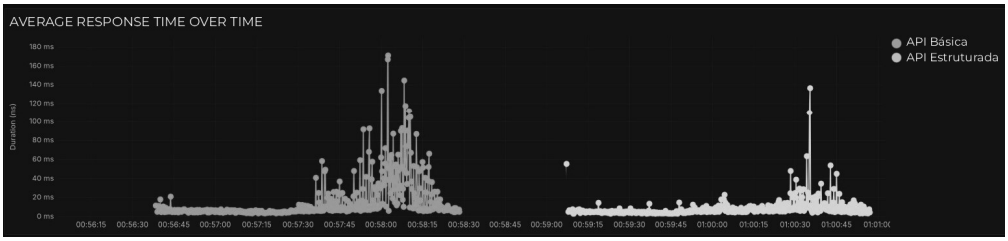


Figura 5: Resultado do cenário 5 – GET com CQRS: Gráfico *Average Response Time Over Time*.

**Fonte:** Elaboração própria.

*Cenário 6*

O teste manteve o mesmo modelo de variação de carga do cenário anterior, simulando acessos repetidos a recursos já consultados, a fim de medir os ganhos obtidos com a reutilização de dados.

Após o período inicial de aquecimento do *cache* (primeiras requisições), observou-se a melhor performance entre todos os experimentos realizados. O tempo médio de resposta reduziu de 14,38 ms, sem *cache*, para 2,52 ms com *cache*, redução de 82%. Essa diferença reflete o impacto da eliminação de consultas redundantes ao banco de dados, otimizando o tempo de processamento e reduzindo a latência geral das respostas. Esses resultados estão representados na Tabela 6.

Tabela 6: Resultado do cenário 6 – GET com CQRS e Cache em Memória.

| **Métrica** | **API Básica** | **API Estruturada** |
|---|---|---|
| Peak RPS | 978 | 973 |
| Max Response Time | 222 ms | 101 ms |
| HTTP Failures | 0 | 0 |
| Average Response Time | 14,38 ms | 2,52 ms |
| Requests Made | 42.592 | 43.087 |

**Fonte:** Elaboração própria.

Os resultados indicam que, embora a API Básica apresente bom desempenho em cenários de baixa carga, a API Estruturada se mostra amplamente superior em escalabilidade e estabilidade, principalmente quando associada a padrões arquiteturais como CQRS e técnicas de *cache*. Essa robustez é crucial para aplicações corporativas e sistemas distribuídos, nos quais a previsibilidade e a resistência sob carga são fatores determinantes de qualidade.

## 6. Considerações Finais

O presente trabalho teve como objetivo avaliar a influência de diferentes abordagens arquiteturais no desempenho e na estabilidade de APIs desenvolvidas em C#, comparando uma implementação básica, sem padrões de projeto, com uma versão estruturada, fundamentada em princípios de *Domain-Driven Design* (DDD), SOLID e *Clean Architecture*.

A metodologia experimental, baseada em testes de carga conduzidos com o K6 e visualização de métricas no Grafana, permitiu uma análise comparativa consistente entre as duas implementações. Em cenários de baixa concorrência, a API Básica demonstrou

maior velocidade média de resposta e maior taxa de requisições processadas, reflexo direto de seu fluxo reduzido e ausência de camadas intermediárias.

Entretanto, com o aumento da carga, a API Básica apresentou falhas e variações de latência. Já a API Estruturada manteve operação estável, sem falhas, e apresentou desempenho até 82% superior em termos de tempo médio de resposta, evidenciando os benefícios práticos de uma arquitetura modular e bem definida.

A introdução de padrões complementares, como o *Command Query Responsibility Segregation* (CQRS) e o uso de *cache* em memória, reforçou essa diferença, com redução da latência média e estabilizando o comportamento da aplicação. Assim, os resultados obtidos confirmam que uma arquitetura orientada a boas práticas, embora mais complexa em sua concepção inicial, proporciona ganhos reais de confiabilidade, escalabilidade e desempenho sustentado em ambientes de alta demanda.

### *6.1. Limitações do Estudo*

Reconhece-se que os experimentos foram realizados em ambiente controlado, com todos os componentes executando em contêineres Docker. Essa configuração assegura reprodutibilidade e isolamento, porém pode não refletir integralmente a variabilidade de cenários reais de produção, nos quais fatores como latência de rede, características de *hardware* e estratégias de balanceamento de carga exercem papel relevante.

Além disso, os cenários avaliados concentraram-se em operações específicas de leitura e escrita, utilizando um modelo de dados simplificado e uma única instância da aplicação. Dessa forma, não foram explorados cenários distribuídos ou com dependências externas, que poderiam impactar de maneira distinta o desempenho das arquiteturas analisadas.

Por fim, os resultados apresentados priorizam a análise comparativa de comportamento e tendências sob diferentes níveis de carga, não contemplando uma avaliação estatística aprofundada com múltiplas repetições e análise de variabilidade. Ainda assim, os comportamentos observados mostraram-se consistentes e suficientes para atender aos objetivos propostos neste estudo.

### *6.2. Trabalhos Futuros*

Embora os resultados obtidos tenham demonstrado de forma clara os benefícios da adoção de boas práticas arquiteturais sobre o desempenho e a estabilidade de APIs, há diversos caminhos que podem ser explorados em trabalhos futuros para ampliar e aprofundar os achados desta pesquisa.

Um primeiro ponto consiste na ampliação dos testes para cenários distribuídos, com múltiplas instâncias das aplicações operando em conjunto. A análise de escalabilidade horizontal permitiria compreender como a arquitetura estruturada se comporta sob balanceamento de carga, bem como avaliar o impacto de diferentes estratégias de distribuição, como *round robin* e *least connections*, sobre a latência e o *throughput* global do sistema.

Outra linha de investigação relevante envolve a adoção de mecanismos de cache distribuído, como Redis, em substituição ao cache em memória local utilizado neste estudo. Tal abordagem possibilitaria avaliar ganhos de performance e consistência em ambientes multi-instância, aproximando os experimentos de um cenário corporativo real.

Também seria pertinente incluir testes de consumo de recursos, medindo de forma detalhada a utilização de CPU, memória e I/O durante as execuções. Essas métricas complementariam a análise atual, oferecendo uma visão mais abrangente do custo computacional associado a cada arquitetura.

Por fim, recomenda-se realizar testes de longa duração e carga variável, para observar o comportamento das aplicações em condições prolongadas de estresse e possíveis degradações de desempenho ao longo do tempo.

## Referências


Bass, L., Clements, P., Kazman, R., 2021. Software Architecture in Practice: Fourth Edition. Addison-Wesley.

Boettiger, C., 2014. An introduction to docker for reproducible research .

Bomfim, J.G., de Jesus Dias Santana, L., 2019. A internet das coisas: Utilidade pública ou fetiche tecnológico? II Encontro Regional Norte-Nordeste da ABCiber .

Cavalcante, E.J., 2021. O novo paradigma tecnológico do setor financeiro nacional: a implantação do open banking no brasil. Desconhecido .

Chimuco, P.V.L., Barbosa, A.C.G., 2024. Validação de carga em testes de desempenho de apis de cadastro de usuários: Garantindo qualidade e eficiência com docker e k6 .

Data, I., 2025. Influxdb documentation. URL: `https://docs.influxdata.com/influxdb/v1`. acesso em: 17 out. 2025.

Evans, E., 2004. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.

Fielding, R.T., 2000. Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis. University of California, Irvine.

Filho, A.A., de Almeida, C.D.A., 2014. Garantia do princípio de liskov com testes em classes herdeiras URL: `https://www.quixada.ufc.br/wp-content/uploads/2014/04/GarantPrincipLiskov.226.pdf`.

Jin, B., Sahni, S., Shevat, A., 2018. Designing Web APIs: Building APIs That Developers Love. O'Reilly Media.

Kronis, K., Uhanova, M., 2018. Performance comparison of java ee and asp.net core technologies for web api development .

Labs, G., 2025a. Grafana documentation. URL: `https://grafana.com/docs/grafana/latest`. acesso em: 17 out. 2025.

Labs, G., 2025b. Grafana k6 documentation. URL: `https://grafana.com/docs/k6/latest`. acesso em: 17 out. 2025.

Lauret, A., 2025. The Design of Web APIs. 2 ed., Manning Publications, Shelter Island.

Marinescu, F., 2019. Domain-Driven Design: The First 15 Years. Leanpub. Includes contributions by Eric Evans, Vaughn Vernon, Alberto Brandolini, Scott Millett, among others.

Martin, R.C., 2003. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, New Jersey.

Martin, R.C., 2019. Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software. Com contribuições de James Grenning e Simon Brown.

Merkel, D., 2014. Docker: Lightweight linux containers for consistent development and deployment. Linux Journal .

Misturini, L.T., 2021. Análise e implementação de cache para aplicação web.

Pavaneli, T.H., 2023. Desenvolvimento de uma api e uma aplicação web para auxiliar na análise de dados providos por aplicação móvel. https://repositorio.ufu.br/handle/123456789/37886.

Ribeiro, A.O., Bagnoli, V., 2020. Open banking: impactos e desafios no mercado financeiro. Constituição, Economia e Desenvolvimento: Revista Eletrônica da Academia Brasileira de Direito Constitucional 12, 219–242.

Richter, J., 2024. Performance Impact of the Command Query Responsibility Segregation (CQRS) Pattern in C# Web APIs. Tese de doutorado. Universitats-und Landesbibliothek Sachsen-Anhalt.

Rozanski, N., Woods, E., 2012. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

de Souza, J.C., 2022. A importância de uma arquitetura no desenvolvimento de um software.

Vernon, V., 2013. Implementing domain-driven design. Addison-Wesley.

Wang, T.X., McLarty, M., 2021. Apis aren't just for tech companies. Harvard Business Review URL: https://hbr.org/2021/04/apis-arent-just-for-tech-companies.