INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO SERTÃO PERNAMBUCANO

COORDENAÇÃO DO CURSO DE Tecnologia em Sistemas para Internet CURSO

Tecnologia em Sistemas para Internet

DAVID AMANDO BARBOSA

ANÁLISE COMPARATIVA DE DESEMPENHO ENTRE ARQUITETURAS CRUD E CQRS POR MEIO DE TESTES DE CARGA

SALGUEIRO

2026

DAVID AMANDO BARBOSA

Trabalho de Conclusão de Curso apresentado à Coordenação do curso de Tecnologia e Sistema Para Internet do Instituto Federal de Educação, Ciência e Tecnologia do Sertão Pernambucano, campus Salgueiro, como requisito parcial à obtenção do título de Tecnólogo em Sistemas Para Internet.

Orientador(a): Prof. Gustavo Freitas Sanchez.

SALGUEIRO

2026

Dados Internacionais de Catalogação na Publicação (CIP)

B238 Barbosa, David.

Análise Comparativa de Desempenho Entre Arquiteturas CRUD e CQRS por Meio de Testes de Carga / David Barbosa. - Salgueiro, 2026.
32 f. : il.

Trabalho de Conclusão de Curso (Sistemas para Internet) -Instituto Federal de Educação, Ciência e Tecnologia do Sertão Pernambucano, Campus Salgueiro, 2026.
Orientação: Prof. Dr. Gustavo Freitas Sanchez.

1. Engenharia de software. 2. CQRS. 3. Desempenho de Software. 4. Persistência Poliglota. 5. CRUD. I. Título.

CDD 005.1

Gerado automaticamente pelo sistema Geficat, mediante dados fornecidos pelo(a) autor(a)

ANÁLISE COMPARATIVA DE DESEMPENHO ENTRE ARQUITETURAS CRUD E CQRS POR MEIO DE TESTES DE CARGA

Trabalho de Conclusão de Curso apresentado à Coordenação do curso de Tecnologia e Sistemas Para Internet do Instituto Federal de Educação, Ciência e Tecnologia do Sertão Pernambucano, Campus Salgueiro, como requisito parcial à obtenção do título de Tecnólogo em Sistemas Para Internet.

Aprovado em: ___/___/_____.

BANCA EXAMINADORA

_____
Prof. Gustavo Freitas Sanchez Orientador(a)
IF Sertão PE – Campus Salgueiro

_____
Prof. Allanyo Antonio Silva Santos
IF Sertão PE – Campus Salgueiro

_____
Prof. Marcelo Anderson Batista Dos Santos
IF Sertão PE – Campus Salgueiro

SALGUEIRO

2026

“Quem terceiriza o pensamento enfraquece a própria mente. Pensar é como um músculo: se não for exercitado, atrofia.”

(Auto Desconhecido)

AGRADECIMENTOS

Agradeço primeiramente a Deus, por ter me sustentado até hoje e ter ajudado a superar os desafios desta jornada académica e pela oportunidade de concluir mais esta etapa.

Aos meus pais e familiares, pelo apoio incondicional, pela paciência durante as longas horas de estudo e por acreditarem no meu potencial mesmo nos momentos de incerteza. Sem o suporte de vocês, esta conquista não seria possível.

Ao meu orientador, Prof. Gustavo Freitas Sanchez, pelas orientações precisas, pelo rigor técnico e por me desafiar a transformar este projeto numa investigação científica de valor. A sua paciência e conhecimento foram fundamentais para o amadurecimento desta análise.

Aos professores do curso de Tecnologia em Sistemas para Internet do IF Sertão PE, pelos conhecimentos compartilhados e por contribuírem para a minha formação profissional e crítica ao longo destes anos.

A Dona Sandra Carvalho de Souza, por ter acreditado sempre em mim, ter torcido pela minha formação e acreditar no meu potencial.

A equipe do Instituto Federal de Ciência e Tecnologia do Sertão Pernambucano (IF SERTÃO PE) Campus Salgueiro pelas oportunidades proporcionadas. Agradeço aos servidores do IF e docentes do Curso de Sistema para internet pelo exemplo de ética e profissionalismo.

Ao meu Amigo Gabriel Lopes, futuro Doutor em medicina pelo apoio desde o ensino médio até o presente momento, seus incentivos foram fundamentais na minha vida.

A Arquiteta Grazzi Freire pelos conselhos e por ser um ombro amigo.

Por fim, a todos que, direta ou indiretamente, contribuíram para a realização deste trabalho e para a minha evolução como desenvolvedor e cidadão.

RESUMO

O presente trabalho implementa uma análise comparativa de desempenho entre duas arquiteturas o CRUD(Create, Read, Update and Delete) e CQRS (command Query Responsibility Segregations) em aplicações web na plataforma .NET O objetivo do experimento é abordar o impacto ao sistema de cada arquitetura sob altos volumes de dados considerados as variáveis métricas de vazão, latência, P95 e P99. Portanto, para isso foram desenvolvidas implementações distintas utilizando bancos de dados SQL Server e MongoDB. Os testes foram realizados utilizando ferramentas de teste de código aberto, como o Locust baseado em Python, e o Nbomber, em C# puro. A Infraestrutura é controlada com Docker e WSL 2. Os resultados obtidos demonstraram diferenças características no comportamento das arquiteturas, evidenciando que o CQRS possibilita melhor desempenho para sistemas em cenários de altas cargas. Em contrapartida, conclui-se que deve-se considerar não apenas o volume de dados, mas também a complexidade arquitetural, requisitos e contexto da aplicação.

Palavras-chave: CQRS. CRUD. Desempenho de Software. Teste de Carga. Persistência Poliglota.

ABSTRACT

This work implements a comparative performance analysis between two architectures: CRUD (Create, Read, Update, and Delete) and CQRS (Command Query Responsibility Segregation) in web applications on the .NET platform. The objective of the experiment is to address the system impact of each architecture under high data volumes, considering the metric variables of throughput, latency, P95, and P99. Therefore, distinct implementations were developed using SQL Server and MongoDB databases. The tests were performed using open-source testing tools, Locust (based on Python) and Nbomber (in pure C#), and controlled infrastructure with Docker and WSL 2. The results obtained demonstrated characteristic differences in the behavior of the architectures, showing that CQRS enables better performance for systems in high-load scenarios. Conversely, it is concluded that not only the volume of data but also the architectural complexity, requirements, and context of the application should be considered.

Keywords: CQRS. CRUD. Software Performance. Load. Testing. Polyglot Persistence.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
ASP.NET	Active Server Pages .NET
CQRS	Command Query Responsibility Segregation
CRUD	Create, Read, Update, Delete
CPU	Central Processing Unit
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JSON	JavaScript Object Notation
NBomber	Ferramenta de teste de carga para aplicações .NET
Locust	Ferramenta de teste de carga baseada em Python
RAM	Random Access Memory
RPS	Requests Per Second (Requisições por Segundo)
REST	Representational State Transfer
SQL	Structured Query Language
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

SUMÁRIO

1 INTRODUÇÃO

A Engenharia de Software desempenha um papel fundamental na garantia da qualidade dos sistemas, promovendo a funcionalidade e o desempenho atendendo as normas e documentação associada. Nesse contexto, o guia SWEBOK (*Engineering Body of Knowledge*) que serve de referência global para profissionais da área, estabelece o teste de software como uma das áreas essenciais para validação de sistemas. Segundo Sommerville (2011), a atividade de teste é destinada a demonstrar o que o programa cumpre o seu propósito, reduzindo custos financeiros e risco de falhas, como o Ariane 5 em 1996, ocasionado por erro de processamento de dados. (J. L. LIONS, 1996).

Dentre os fatores que compõem a qualidade do software, o desempenho tem relevância crítica em sistemas submetidos a cenários de alta demanda, a capacidade de resposta a múltiplas requisições simultâneas afeta diretamente a experiência dos usuários e a escalabilidade da aplicação. A norma ISO/IEC/IEEE 29119-1:2013 define o teste de carga como um subconjunto que avalia o comportamento de aplicações submetidas a condições de volumes de requisições.

Mais do que um conceito prescritivo, a Engenharia de Software moderna exige a validação dos modelos arquiteturais.

No contexto arquitetural, aplicações baseadas em arquitetura tradicional CRUD (Create, Read, Update, Delete) tendem a enfrentar gargalos de IO, tornando-se insuficiente para garantir o desempenho e a escalabilidade em sistemas de alta escala. Diante desses problemas, surge como alternativa o padrão arquitetural CQRS (Command Query Responsibility Segregation), proposto por Martin Fowler, que sugere a segregação de leitura e escrita que permite a otimização independente em sistemas pequenos o CRUD se torna a melhor escolha devido a sua simplicidade.

Diante desse cenário, este estudo investiga experimentalmente se a arquitetura apresenta desempenho superior ao modelo CRUD tradicional quando submetidas a testes de carga gradual. A fim de viabilizar a coleta de dados, o experimento foi configurado com taxa gradual de 10 usuários por segundo até atingir pico de 10.0000 usuários simultâneos e o ambiente controlado utilizando Wsl 2 (Windows Subsystem for Linux) e Docker. Esta infraestrutura foi utilizada para isolar variáveis externas, garantido que as métricas coletadas pelo Locust e o NBomber refletem no desempenho estrutural do modelo Crud (monolítico) e CQRS (segregação de comandos e consultas).

2 REFERENCIAL TEÓRICO

Este capítulo apresenta o embasamento teórico para a compreensão dos padrões de base conceitual para a análise comparativa entre as arquiteturas CRUD e CQRS sob cenários de alta carga e conceitos relacionados a métricas de teste em Engenharia de Software. A fundamentação aborda a manipulação de dados, destacando arquiteturas centralizadas e as propostas de segregação de responsabilidades.

Contudo, são abordados princípios de comunicação via API Rest, modelo conceitual em sistemas distribuídos e fundamentos da engenharia de software necessários para interpretar adequadamente as métricas coletadas, como throughput, latência média e percentis P95 e P99.

A estruturação deste referencial está dividida da seguinte forma: a seção 2.1 aborda os fundamentos de APIs REST e Web Services como padrão de comunicação; a seção 2.2 discute a Consistência Eventual e o padrão CQRS; e a seção 2.3 explora os fundamentos do Teste de Software.

2.1 APIs Rest e Web Services

As APIs (*Application Programming Interface*) ou interface de programação de aplicação é responsável pela comunicação de diferentes serviços, fazendo uma ponte entre as aplicações. em geral, é um conjunto de regras ou protocolos para troca de dados, e que mantem ocultos dados interno do sistema Michael Goodwin, (2023). APIs baseadas em Web Service utilizam protocolo HTTP em servidores com os métodos como GET, POST, PUT e DELETE para representar operações.

O estilo arquitetural REST é um acrônimo para *Representational State Transfer*, definido por Roy Fielding, e fundamentada em princípios de comunicação como *stateless*, métodos padronizados HTTP e recurso por URL.

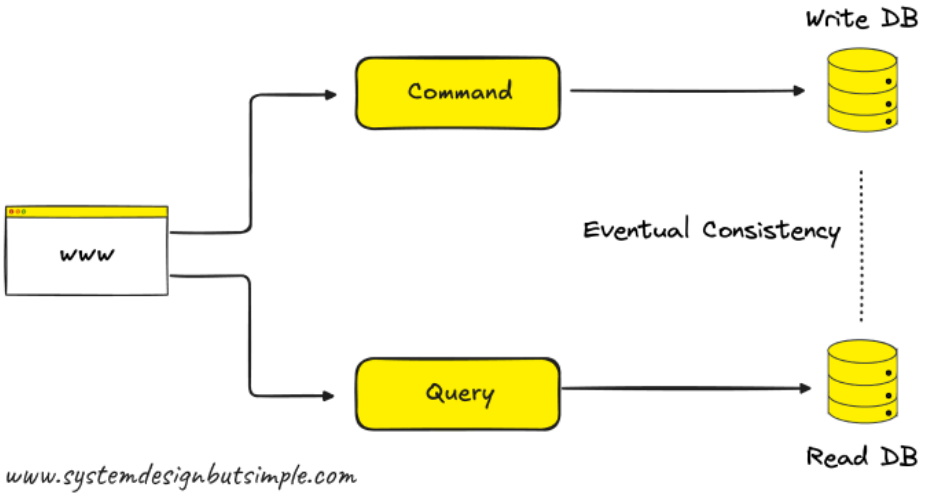
Para este trabalho, a conformidade com princípios Rest garante a uniformidade das métricas entre as arquiteturas usadas. Esta padronização garante que interferências ocorrem por problemas internos e não por variação na camada de comunicação.

2.2 Consistência eventual e CQRS

A consistência eventual, definida por Burckhardt (2014), permite maior escalabilidade ao aceitar pequenos atrasos na sincronização dos dados em troca de melhor desempenho sob

alta carga Figura 01. Esse equilíbrio entre desempenho e consistência garante equilíbrio a sistemas de altos volumes ou sistemas distribuídos. Entretanto, a consistência eventual também traz desafios, como não há garantia de atualização imediata dos dados, pode acarretar situações em que o usuário consulte informações que ainda não refletiram na operação realizada e também na complexidade de sincronização desses dados.

Figura 01 - Representação simples de consistência eventual.



Fonte: Stephane Moreau (2026).

No CQRS essa característica se torna mais presente, pois a arquitetura separa a escrita (command) e a leitura (Query) e atualiza de forma assíncrona, significando que em um período de tempo os dados não refletem instantaneamente, somente após a propagação dos eventos. Conforme discutido por Sebastian Burckhardt (2014), ao aceitar uma pequena janela de desatualização de dados em suas múltiplas réplicas, no caso cópias ou replicas, após nenhuma atualização for feita, elas convergiram para o mesmo dado.

Essa lógica aparece quando o banco de leitura é alimentado a partir de eventos gerados no banco de escrita. O modelo de leitura funciona, na prática, como uma réplica especializada e otimizada para consultas. Assim, pode haver um pequeno atraso entre a execução de um comando e sua visualização nas consultas.

2.3 Teste de Software e Desempenho

A disciplina de teste de software desde a década de 80 vem ganhando maturidade como uma das áreas fundamentais da Engenharia de Software. Normas como ao padrão IEEE 829 estabeleceu o padrão de documentação de teste de software (RIOS, 2022). Naquele

período, as atividades formais predominantes eram práticas manuais, documentais e voltadas à verificação funcional de sistemas isolados.

Com o aumento das complexidades dos sistemas, o foco dos testes expandiu-se de ser meramente manuais, para simulações de cargas automatizadas e ferramentas modernas seguindo a análise do comportamento do software sob condição de estresse. Segundo Sommerville (2011), o teste moderno deixa de ser uma atividade isolada para se tornar um modelo contínuo de qualidade de software, se subdividindo em entidades menores até a mais complexa infraestrutura.

3 METODOLOGIA E AMBIENTE EXPERIMENTAL

Para a realização do estudo, desenvolveu-se uma aplicação web utilizando a Linguagem C# e o Framework ASP.NET Core. O sistema atual é objeto de estudo experimental, estruturado para permitir a comparação de arquiteturas distintas: o modelo convencional *CRUD (Create, Read, Update, Delete)* e o *CQRS (Command Query Responsibility Segregation)*.

Ambos os projetos mantêm a mesma interface mantendo o foco na comunicação padronizada (Silva de Souza, 2021). No modelo Crud Tradicional a escrita e leitura são centralizadas em um modelo monolítico e essa centralização pode gerar gargalos e menor desempenho sobre alta concorrência. Em contrapartida, a arquitetura CQRS separa essas responsabilidades em comandos (Escritas) e consultas (leitura), viabilizando a otimização independente do modelo (Elsyyad, 2024).

Embora a estrutura base da aplicação tenha sido baseada em modelos educacionais em referência a (BALTIERI, 2015), o projeto foi adaptado e ajustado para suportar o ambiente de teste de desempenho, incluindo a integração de bases de dados, padrões de desempenho e população do Banco de dados com dados Fake pela biblioteca Bogus. As seções seguintes detalham as tecnologias empregadas, os padrões arquitetônicos adotados e os critérios estabelecidos para a coleta de dados e análise de dados experimentais.

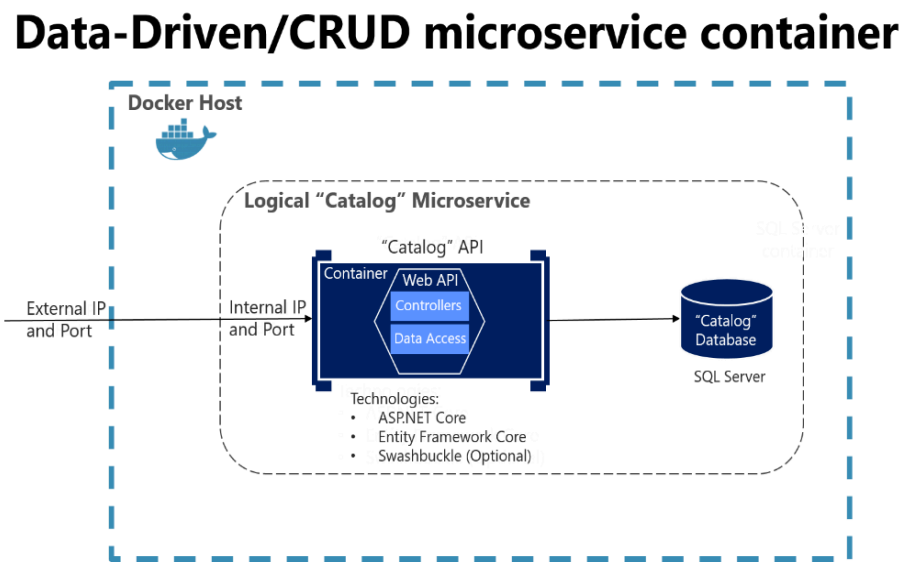
3.1 Padrão Arquitetural

O *CQRS* (*Command Query Responsibility Segregation*) é um padrão de arquitetura que propõe a separação entre as operações de leitura e escrita. Conforme RICHTER (2024)

explica que o *CQRS* quebra o modelo conceitual único em dois: um voltado para atualizações (Commands) e outro para consultas (Queries). Essa distinção permite que a lógica de negócios e os esquemas de dados evoluam o desempenho de acordo com a necessidade de cada fluxo. À medida que o empreendimento expande, o volume de transações cresce proporcionalmente, exigindo maior capacidade de processamento da aplicação. O modelo CRUD (*Create, Read, Update, Delete*), Figura 02, fundamenta-se em um modelo de dados que prioriza a integridade de dados por meios das propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade).

Nesta abordagem tradicional, todas as operações afetam diretamente a tabela do banco de dados relacional. Embora isso possibilita a confiabilidade dos dados, essa centralização gera contenção de recursos, limitando a escalabilidade horizontal, no caso sua expansão. Portanto o conceito de Persistência Poliglota, popularizado por Fowler e Sadalage (2012) consiste na prática de utilização de múltiplas tecnologias de armazenamento de dados em um mesmo ecossistema.

Figura 02 - Design de micros serviço controlado por dados/CRUD simples



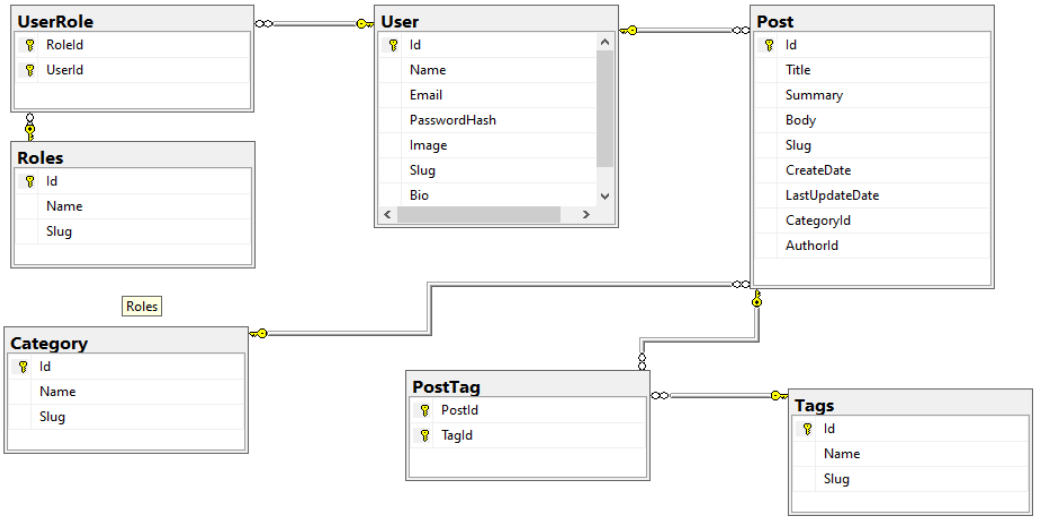
Fonte: César de la Torre Bill Wagner Mike Rousos, (Microsoft Corporation)

Essa estratégia arquitetural, utiliza se um banco de dados não relacional (Mongodb) para modelo de escrita (Write), priorizando a flexibilidade e taxa de ingestão de dados, enquanto o banco Relacional (SQL Server) e empregado para modelo de leitura (Read) garantindo a capacidade de leituras, o desempenho e escalabilidade de forma otimizada.

3.2 Diagrama do Banco de Dados e Estrutura de Persistência

A base de dados experimental é representada pelo diagrama ilustrado na Figura 03. Este esquema, feito no SQL Server, representa visualmente as tabelas e os relacionamentos usados em ambas as arquiteturas, garantindo que as informações processadas e o volume de registros fossem idênticos em todos os cenários de teste. Embora ambas as abordagens adotem o mesmo modelo relacional, o modo como os dados são manipulados diverge estruturalmente.

Figura 03- Modelo de dados relacional e relacionamentos no SQL Server.



Fonte: O Autor (2024).

A transição do modelo CRUD para o CQRS exigiu a implementação de gerência a persistência poliglota e a comunicação entre as camadas do mapeamento híbrido de identificadores (Mongo Sql Mapping) e Manipuladores de Lógica (Handlers).

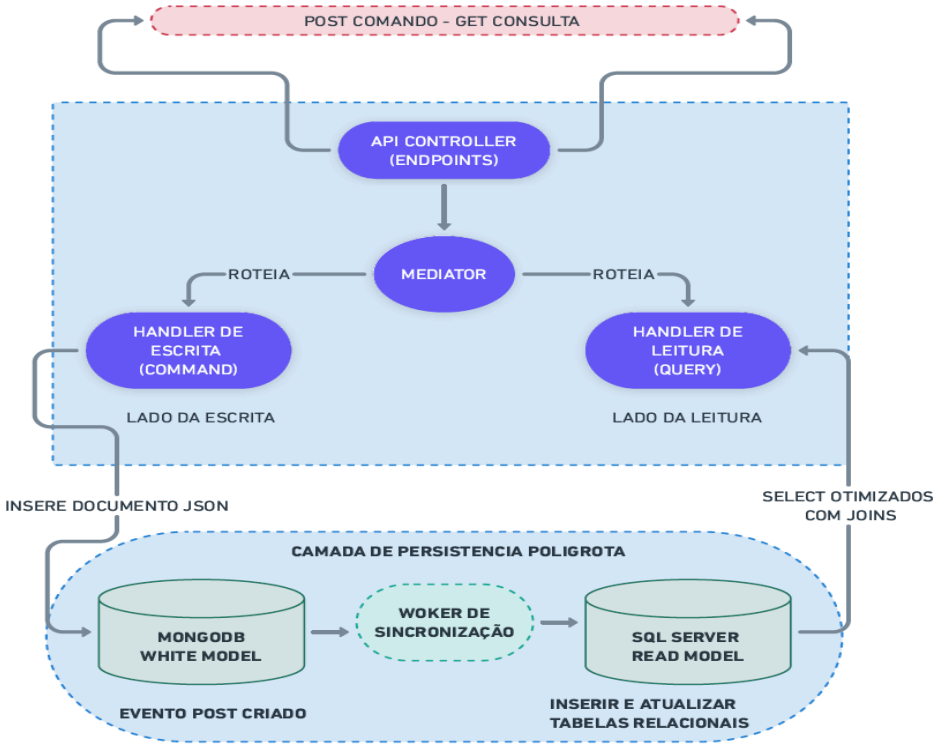
O Mapeamento híbrido de identificadores diferente do modelo CRUD, onde uma única chave primária atende a aplicação, a arquitetura CQRS exige a coexistência de esquemas de identificação distintos e trabalhar com identificadores híbridos, uma escolha estratégica para otimizar o sistema e ambientes distribuídos (KLEPPMANN, 2017). Portanto, há a necessidade de sincronização. De um lado, temos o MongoDB usando GUIDs e do outro, o SQL Server utiliza chaves sequenciais para manter e organizar os índices, o que reflete diretamente em leituras mais ágeis e respostas imediatas para o usuário, mesmo sob forte estresse.

- **Chaves Alfanuméricas (*GUID/ObjectId*):** Utilizadas pelo MongoDB para garantir a unicidade e a performance em operações de escrita distribuída.

- **Chaves Inteiras Sequenciais (*Identity*):** Mantidas no SQL Server para preservar a integridade referencial e a eficiência em consultas relacionais e ordenações

Segundo o princípio da responsabilidade única, a lógica de negócio foi encapsulada a em classes denominadas *handlers*, onde cada componente executa uma tarefa, Figura 04.

Figura 04 - Esquema de domínio e camada poliglota



Fonte: Autor (2026)

Destacam-se:

- **RegisterUser Handler:** Detém a responsabilidade exclusiva de validar e persistir o estado inicial do usuário na base de escrita (MongoDB).
- **User Created EventHandler:** Atua como um observador de eventos; uma vez que o usuário é persistido no MongoDB, este *handler* é disparado para realizar a propagação e sincronização dos dados para o SQL Server.

3.3 Vantagens e Desafios da Segregação

Vantagens:

Conforme a documentação da (Microsoft, 2024), o principal benefício dessa abordagem reside na escalabilidade independente, separação de responsabilidades e Assimetria de processamento. Por exemplo: em cenários onde as taxas de leitura são desproporcionais em relação à escrita, em suposto, faz a lógica do Princípio de Pareto (ou regra 80/20) criado pelo economista italiano Vilfredo Pareto (1896) e adotado por Joseph Juran na aplicação de gestão de qualidade, como uma heurística fundamental na engenharia de software.

Este fenômeno é muito bem observado na maioria das aplicações web através do termo "*Read-to-Write Ratio*", razão Leitura/escrita, uma vez que os usuários consomem mais conteúdo do que produzem. Além disso, a segregação permite a otimização do banco de escrita otimizando a taxa de ingestão (*throughput*).

Desvantagem:

Em contrapartida, Fowler (2011) alerta sobre a complexidade do código fonte de um programa ou projeto. o desafio de manter dois armazenamentos de dados sincronizados introduz modelo o de consistência de dados eventual (*Eventual Consistency*).

Diferentes dos sistemas monólitos que usam o princípio *ACID* (Atomicidade, Consistência, Isolamento e Durabilidade) que garante a confiabilidade das transações em uma única base, o *CQRS* opera com um atraso na propagação dos dados sob o modelo de consistência eventual causado pela divisão física dos bancos de dados, resultando no fenômeno de latência de aplicação (*replication lag*),contudo que o disparo do evento e essencial para informar ao componente responsável pela sincronização do banco que o dado já foi persistido .

3.4 Docker e WSL 2

Docker é uma plataforma de código aberto que utiliza recursos do *kernel* para isolar a execução de processos através de contêineres. Diretas de máquinas virtuais tradicionais, que compartilham o mesmo hardware, o container compartilha o mesmo kernel do sistema hospedeiro.

De acordo com a (AWS, 2021), essa tecnologia possibilita a execução de aplicações como micros serviços em ambientes distribuídos, funcionando de forma idêntica em qualquer máquina, mas funcionando apenas em ambientes Linux. A partir das imagens do container, que são executáveis portáteis e imutáveis com bibliotecas e dependências necessárias para rodar sua aplicação. Como o Docker opera sobre o Kernel Linux, sua operação em sistemas windows requer a utilização do Windows Subsystem for Linux 2 (WSL 2) que permite a execução real do Linux.

Foram utilizadas imagens oficiais do Microsoft SQL Server e do MongoDB, obtidas no repositório público Docker Hub para garantir a padronização experimental e isolamento dos serviços.

3.5 Ambiente

Para garantir o controle das variáveis e elaboração dos experimentos, os testes foram feitos em ambiente local e isolado no próprio computador, utilizando container Docker em conjunto com WSL 2.

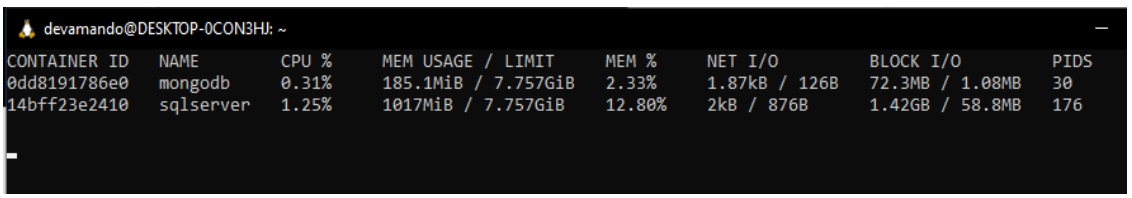
A infraestrutura possui as seguintes configurações:

- Processador Ryzen 7 5700x
- 32 GB memória Ram
- 512 GB Ssd Armazenamento
- Windows 10

Os bancos de dados executados em contêiner foram configurados com as seguintes restrições:

- **8 GB de memória RAM**
- **4 núcleos de CPU**

Figura 05 - Comando Docker stats com banco hardware limitado.



Fonte: Autor (2026).

O computador por conta dos processos em segundo plano foi realizado a partir de 20% da CPU e memória RAM em 25%, não sendo necessário todas essas configurações para elaborar o mesmo teste.

4 FERRAMENTAS E PARÂMETROS PARA O TESTE

4.1 Locust

O Locust é um Framework de teste de carga de código aberto que permite escrever scripts em Python em contrapartida a interfaces tradicionais e arquivos verbosos, permitindo moldar as ações e comportamentos dos usuários, utilizando múltiplos threads que simulam usuários virtuais. Segundo (Heyman e Byström, 2011) os criadores do Framework, a ferramenta foi projetada para testes de desempenho/carga para HTTP e outros protocolos. Sendo reconhecida pelo seu uso em simular milhões de usuários no Battlelog, uma aplicação web dos jogos Battlefield (Locust).

Após a instrumentação da ferramenta, desenvolveu-se para avaliar os seguintes endpoints das rotas da aplicação:

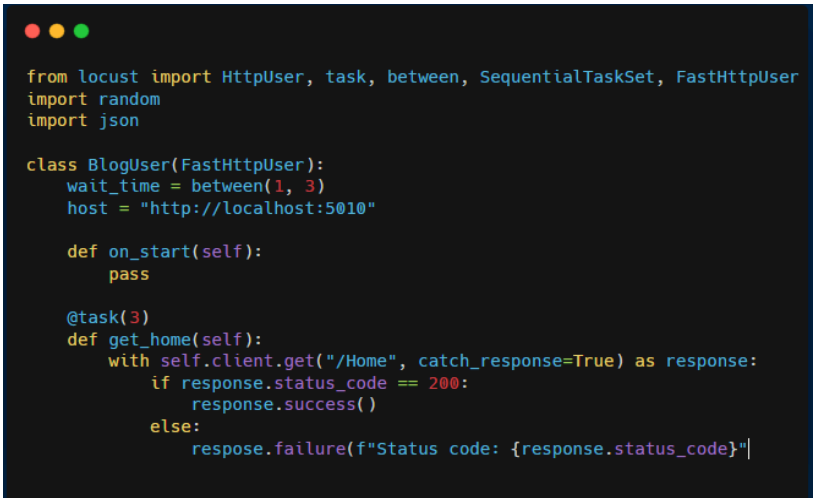
/Home.
/Post/v1/posts (Listagem).
/Post/v1/posts/1 (Detalhe).
/Post/v1/posts/category/backend.
/Post/v1/posts/category/frontend.
/v1/categories (Listagem).
/v1/categories/1 (Detalhe).

Normalmente, o locust opera com requisitos baseado no *Http User* que utiliza a biblioteca *python-requests* em clientes HTTP. Neste caso o padrão em Python puro apresentou um consumo maior em memória RAM e latência sob estresse extremo, como alternativa optou-se pela utilização do componente *Fast Http User* que utiliza a biblioteca *gevent http client,* um cliente de alto desempenho baseado em C (Locust, 2009-2025) com menor consumo de recursos permitindo a simulações de carga teste mais requisições por segundo. Dentro do framework Locust (2009-2025), as ações dos utilizadores virtuais são definidas

pelo decorador *@taks* que permite o teste baseado em comportamento. Cada método marcado com o decorador permite simular o comportamento humano, neste caso, atribuindo o peso superior às rotas de leitura GET em comparação a Escrita POST/PUT em escala aplicável a sistemas webs e utilizou-se a função *between.* Esta configuração define o intervalo de espera entre a conclusão de uma tarefa e início de outra com a introdução de atraso de atraso em segundos, ilustrada na Figura 06.

No endereço padrão http://localhost:8089, permite acompanhar o teste em tempo real no navegador com as informações de Request Statistics, Response Time Statistics, Failures Statistics, Chart (gráficos dinâmicos), Logs e Downloads Data(dado).

Figura 06 - Trecho de Configuracao do Scrip Teste.py



```python
from locust import HttpUser, task, between, SequentialTaskSet, FastHttpUser
import random
import json

class BlogUser(FastHttpUser):
    wait_time = between(1, 3)
    host = "http://localhost:5010"

    def on_start(self):
        pass

    @task(3)
    def get_home(self):
        with self.client.get("/Home", catch_response=True) as response:
            if response.status_code == 200:
                response.success()
            else:
                respose.failure(f"Status code: {response.status_code}"
```

Fonte: Autor (2026)

Ao final do teste, os dados disponíveis para análise são arquivos .*CSV*, formato de texto simples separado por vírgulas e estruturado em linhas e colunas (Microsoft), e um relatório completo em html das informações, hora, data de início, fim do teste, endereço e nome do Script.

4.2. Nbomber

O NBomber é uma ferramenta de teste de carga de código aberto nativo da Plataforma .NET que utilizar C# puro, oferecendo diversas maneiras de controlar e adicionar usuários virtuais por meio de simulação de carga de cenários. A ferramenta oferece suporte a modelos de carga abertos e fechados. Diferente de outras ferramentas que dependem de extensões

adicionais, o NBomber independente de pacotes externos para protocolos específicos e integrasse de forma direta no Ecossistema .Net e bibliotecas nativas da plataforma.

Para o teste experimental, foi criado o projeto utilizando .Net 9.0, no qual foram configurados cenários de carga baseado de modelo aberto por meio de aumento gradual controlando as requisições por segundo e adicionamento configuração ao arquivo pelo mecanismo Garbage Collection, que gerencia a memória para o servidor de Multithreaded de alto desempenho e otimização (Microsoft, 2025).

4.3 Parâmetros para o Teste

O teste de desempenho visa simular carga de dados reais em variados níveis de intensidade. Os parâmetros selecionados para examinar o teste de carga baseiam-se nas permutações das variáveis proposta por Pressman (2011) para validar a capacidade de resposta de uma aplicação. O desempenho geral (P) é dado pelas relações de usuários, transações e volume de dados, conforme a seguinte equação:

$$P = N \times T \times D$$

Onde

- N, número de usuários concorrentes
- T, número de transações on-line por usuários por unidade de tempo
- D, carga de dados processados pelo servidor por transação

Neste cenário, os limites de operações do sistema são definidos pelo aumento gradual de usuários (Ramping) até a variável N = 10.000 com Span Rate de 10 users/s. T = Transações por segundo por usuário (RPS \ N) e D = Dados Processados.

4.4 Variáveis Dependentes

A tabela a seguir detalha as métricas extraídas das ferramentas de teste:

Tabela 01 - Variáveis métricas

Variável	Nome Variável	Importância
Type	Método HTTP	Identifica a operação (ex: GET para consultas, POST para comandos)
Name	Recurso (URI)	Identifica o endpoint ou a

		rota da API que está sendo estressada.
Request Count	Volume de Amostragem	Garante a relevância estatística do teste através do total de solicitações.
Fail Count	Taxa de Erro (Error Rate)	Mede a taxa de erros.
Average	Latência Média	Fornece uma visão geral do tempo de processamento do servidor.
Max Response Time	Latência Máxima	Revela o pior cenário e o limite de tolerância do sistema antes do timeout.
Requests/s	Throughput (Vazão)	Mede a capacidade produtiva da API e sua escalabilidade real.
P95	95th Percentile	**Métrica Chave:** Válida a estabilidade da experiência do usuário em alta escala

Fonte: docs.locust (2026)

5 ANÁLISE COMPARATIVA DOS RESULTADOS

5.1 Locust

Após a realização dos testes de carga com volume de usuários escalados até 10.000 usuários simultâneos, os arquivos gerados requests.csv ao final do teste foram analisados os dados agregados utilizando a biblioteca pandas da linguagem Python, permitindo manipulação e extração de análise de dados, conforme a Tabela 02.

Tabela 02 - Comparativo de Vazão entre CRUD e CQRS em Aggregated

Métrica	Modelo CRUD (Tradicional)	Modelo CQRS (Otimizado)	Melhora
Volume de Requisições	362.269	964.786	Volume
Vazão Média (RPS)	354,98 req/s	927,50 req/s	Velocidade
Latência Média	10.763,05 ms	2.448,84 ms	Tempo
P95 (95% das req.)	26.000 ms	6.900 ms	Tempo
P99 (99% das req.)	28.000 ms	7.800 ms	Tempo
Latência Máxima	30.342,11 ms	9.942,40 ms	Tempo
Average size (bytes)	1124,25773 bytes	1297,533628 bytes	Carga de Dados

Fonte: Author (2026)

A Vazão de dados (throughput) do modelo CQRS atingiu 964.786 req/s sendo duas vezes e meia superior ao modelo CRUD com 354,98 req/s para realizar o mesmo cenário. Está métrica válida a hipótese de que a segregação de responsabilidades permite a melhoria na capacidade de processamento e considerando a duração de teste de aproximadamente 17 minutos (1020 segundos), o modelo CQRS foi capaz de processar 964.786 requisições enquanto o modelo tradicional obteve 362.269 requisições.

A análise dos percentis demonstrada no modelo CRUD, que 95% dos usuários enfrentaram esperas superiores a **26 segundos**, o que em um cenário real resultaria em abandonos e *timeouts*. Em contrapartida, o modelo CQRS obteve P95 em **6,9 segundos** e o P99 (7,8s) no CQRS é muito menor do que no CRUD. Isso indica que o sistema com persistência poliglota é mais resiliente e previsível sob estresse extremo.

Ao analisar a latência média agregada, o impacto da persistência poliglota fica evidente:

- **CRUD (SQL Server):** Latência média de 10.765 segundos.

- **CQRS (SQL Server otimizado para leitura):** Latência média de 2.448 segundos.

A redução de 77,26% no tempo de resposta para consultas complexas justifica a complexidade técnica de manter bases separadas. A eficiência do SQL Server, quando aliviado das operações de escrita concorrentes (que no CQRS ocorrem no MongoDB), permite uma recuperação de dados significativamente mais ágil, Figura 06.

Ao analisar o volume de dados trafegados, permitiu a validação da hipótese da segregação de reponsabilidade dos dados tornando-se mais evidente através do cálculo Throughput real $P = RPS \times D$:

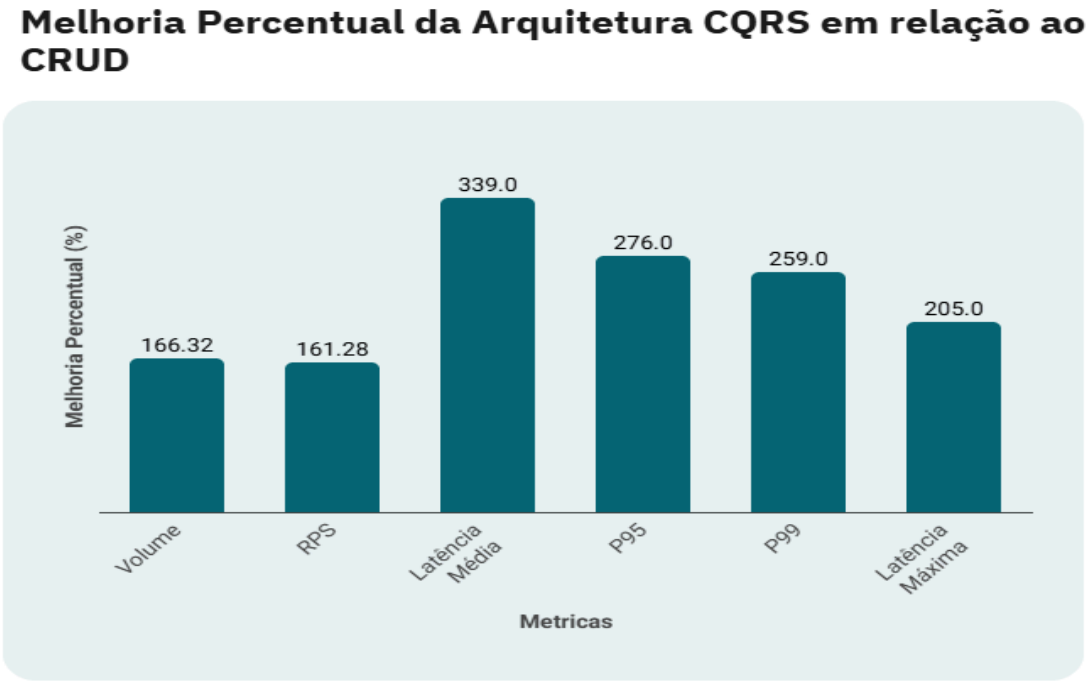
- **No modelo CQRS:** $P = 927{,}50 \times 1297{,}53362 \approx 1.203.690$ bytes/s, convertendo em Megabytes a vazão se limitou a 1,15 MB/s.

- **No modelo CRUD:** $P = 354{,}98 \times 1124{,}25773 \approx 398.113$ bytes/s, convertendo em Megabytes a vazão se limitou a 0,38 MB/s.

Além do ganho em throughput, o volume de dados transferidos também foi significativamente superior no CQRS, totalizando 8.286,41 MB, em contrapartida de 1.608,86 MB no modelo tradicional

Isso prova que o gargalo do CRUD estava no processamento e não na rede, já que o CQRS conseguiu transferir quase 3x mais dados.

Figura 07 - Melhoria Percentual



Fonte (Autor, 2026)

5.2 NBomber

Além dos testes de estresse realizados no Locust, utilizou-se a ferramenta NBomber para validar o comportamento das arquiteturas em um cenário baseado em modelo aberto com *RampingInject 1000 req/s*. Diferente do teste anterior, onde se buscou o ponto de ruptura mais moderado, este teste injetou uma carga constante de requisições.

Tabela 03 - Comparativo de Vazão entre CRUD e CQRS no NBomber

Métrica	Modelo CRUD (Tradicional)	Modelo CQRS (Otimizado)
Requisições OK	11.840	50.860
(RPS)	44,34	143.66
Latência Média	14.08 ms	28,073 ms
P95 (95% das req.)	45,38 ms	115,84 ms
P99 (99% das req.)	59,62 ms	259,07 ms
Average Size (bytes)	1.241,28	1.627,53
Latência Máxima	157,95	512,27
Dados transferidos (MB)	1.608,86	8.286,41

Fonte: (Autor, 2026)

Observa-se que o modelo CRS apresentou desempenho superior a alcançando 143,66 requisições por segundo, enquanto o modelo CRUD atingiu 44,34 RPS. Isso representa um aumento de aproximadamente **224% na capacidade de processamento**.

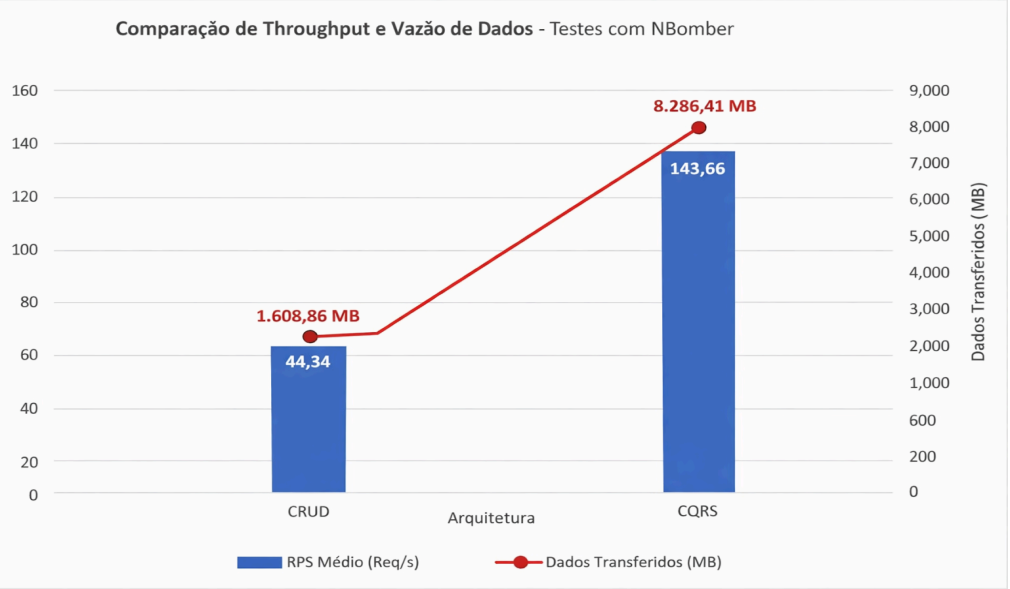
Analisando as métricas, o CQRS conseguiu lidar com mais requisições em relação ao CRUD no mesmo cenário de teste, suportou mais. Isso mostra que, em um ambiente com envio constante de requisições, aproveitando melhor os recursos do sistema. E o tempo médio de resposta do CQRS foi maior (28,07 ms contra 14,08 ms no CRUD), porque ele estava processando muito mais requisições e transferindo muito mais dados. Quando o sistema trabalha mais normalmente o tempo de resposta aumenta um pouco e que haja maior variação nos tempos mais altos (como P95 e P99).

Mesmo com essa diferença na latência, o CQRS se manteve estável durante o teste, houve perda de requisições, mas, mesmo com falhas ligado à lógica do código projetado que acarretou em falhas de requisições não consideradas, pela ausência de between das carga injetadas no cenário, acarretou uma maior latência, contudo ele transferiu mais dados e continuou funcionando de forma consistente. O volume de dados transferidos foi cerca de cinco vezes maior no CQRS.

A Figura 07 apresenta a comparação entre as arquiteturas CRUD e CQRS quanto à vazão média (RPS) e ao volume total de dados transferidos durante os testes realizados com o NBomber.

Além do ganho em throughput, o volume de dados transferidos também foi significativamente superior no CQRS, totalizando 8.286,41 MB, em contrapartida de 1.608,86 MB no modelo tradicional

Figura 08 - Comparação de Throughput e Vazão de dados



Fonte: Autor (2026)

Esse aumento indicou maior capacidade de resposta sob carga constante, evidenciando melhor aproveitamento dos recursos computacionais. O crescimento proporcional entre vazão e volume de dados mostrou que o modelo CQRS processa mais requisições e também mantém estabilidade na entrega de respostas sob taxa fixa de injeção de carga (1000 req/s).

Os resultados demonstram que a separação entre comandos e consultas permitiu a melhor otimização do processamento refletindo diretamente no sistema.

6 CONCLUSÃO

Este trabalho teve como objetivo comparar o desempenho das arquiteturas CRUD e CQRS em cenários de carga utilizados nas ferramentas de teste como o Locust e o NBomber. Valendo ressaltar que os dados analisados foram de forma agregadas, ou seja, não foi analisado individualmente o endpoints dos projetos, mas no geral, a ideia foi entender como cada abordagem se comporta em situações de estresse e carga contínua, buscando perceber suas diferenças na prática, e não apenas na teoria. Enquanto o modelo CRUD costuma ter um bom desempenho em cenários mais simples, quando submetido a volumes elevados de requisições seu limite é atingido mais rápido. Em contraposto, o CQRS mostrou uma maior capacidade de processamento, suportando um volume maior de requisições e transferindo mais dados durante os testes.

De forma geral, os experimentos confirmam que a escolha da arquitetura impacta diretamente no comportamento do sistema sob carga alta. O modelo CRUD é suficiente para aplicações menores ou de menor escalabilidade. Já o CQRS se mostra mais adequado para sistemas que precisam lidar com grande volume de requisições, oferecendo melhor aproveitamento dos recursos e maior capacidade de crescimento.

Assim, conclui-se que não existe uma solução universal que se aplique a todos os sistemas, mas sim arquiteturas mais adequadas para cada contexto. A decisão deve considerar não apenas a complexidade de implementação, mas principalmente os requisitos de desempenho e escalabilidade da aplicação.

7 REFERÊNCIAS

AMAZON WEB SERVICES (AWS). **O que são imagens e contêineres do Docker?** 2024. Disponível em: <https://aws.amazon.com/pt/compare/the-difference-between-docker-images-and-containers/>. Acesso em: 15 dez. 2025.

BALTIERI, André. **ASP.NET Core, CQRS e Mediator**. Balta.io, 13 abr. 2020. Disponível em: <https://balta.io/blog/aspnet-core-cqrs-mediator>. Acesso em: 12 out. 2025

BALTIERI, André. **Windows Terminal**. Balta.io, 14 jun. 2020. Disponível em: <https://balta.io/blog/windows-terminal>. Acesso em: 12 out. 2025.

CORONEL, Carlos; MORRIS, Steven. **Database Systems: Design, Implementation, & Management**. 13. ed. Boston: Cengage Learning, 2018.

DATA STORAGE. **O que é Throughput? Entenda a importância para o armazenamento de dados**. 2023. Disponível em:<https://www.datastorage.com.br/post/throughput>. Acesso em: 04 mar. 2026.

ELSAYYAD, Amr. CQRS vs CRUD: **Choosing the Right Pattern for Your Application**. Medium, 2023. Disponível em: <https://medium.com/@AmrElsayyad/cqrs-vs-crud-choosing-the-right-pattern-for-your-application-0d614167d12b>. Acesso em: 01 jan. 2026.

FOWLER, Martin; SADALAGE, Pramod J. **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**. 1. ed. Upper Saddle River: Addison-Wesley, 2012.

GARCIA, Diego. **Testes de carga com o Locust**. Python Club, 11 jan. 2015. Disponível em: <https://pythonclub.com.br/testes-de-carga-com-o-locust.html>. Acesso em: 15 nov. 2026.

IEEE COMPUTER SOCIETY. **Software Engineering Body of Knowledge (SWEBOK)**. 2024. Disponível em: <https://www.computer.org/education/bodies-of-knowledge/software-engineering>. Acesso em: 12 jan. 2026.

ISO/IEC/IEEE. ISO/IEC/IEEE 29119-2:2013. **Software and systems engineering Software testing** — Part 2: Test processes. Genebra: ISO, 2013.

KLEPPMANN, Martin. **Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems**. Sebastopol: O'Reilly Media, 2017.

LOADVIEW. **Teste de desempenho vs. Teste de escalabilidade**. 2025. Disponível em: <https://www.loadview-testing.com/pt-br/blog/teste-de-desempenho-vs-teste-de-escalabilidade/>. Acesso em: 17 nov. 2025.

LOCUST**. Increase performance**. 2026. Disponível em: <https://docs.locust.io/en/stable/increase-performance.html>. Acesso em: 25 fev. 2026.

MICROSOFT. **Configurações de tempo de execução do coletor de lixo (GC)**. 2024. Disponível em: <https://learn.microsoft.com/pt-br/dotnet/core/runtime-config/garbage-collector>. Acesso em: 02 fev. 2026.

MICROSOFT. .NET Microservices: **Architecture for Containerized .NET Applications**. Version 7.0. Redmond: Microsoft Developer Division, 2023.

MOREAU, Stephane**. CQRS in 1 diagram and 178 words**. System Design, but Simple, [s.l.], [s.d.]. Disponível em: <https://www.systemdesignbutsimple.com/>. Acesso em: 10 fev. 2026.

PRESSMAN, Roger S. **Engenharia de software: uma abordagem profissional**. 7. ed. Porto Alegre: AMGH Editora, 2011.

RICHTER, Jens. **Performance Impact of the Command Query Responsibility Segregation (CQRS) Pattern in C# Web APIs**. 2024. Bachelor Thesis (Wirtschaftsinformatik) Hochschule Merseburg, Merseburg, 2024.

RIOS, Emerson. **A História do Teste resumida em fatos**. ACERTBR, 15 ago. 2022. Disponível em: <https://acertbr.com.br/historia-do-teste-resumida-em-fatos/>. Acesso em: 14 jan. 2026.

SIQUEIRA, J. M. et al**. O uso de software open source nas organizações: uma análise sob a ótica da gestão estratégica**. Revista Gestão.Org, Recife, v. 15, n. 2, p. 210-225, 2017. Disponível em: <https://periodicos.ufpe.br/revistas/gestaoorg/article/view/22555>. Acesso em: 10 jan. 2026.

SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.