UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL

KERMAN MENDES PEREIRA JÚNIOR

APLICAÇÃO DE TESTES DE CARGA COM TECNOLOGIAS DE *CACHE* NA ARQUITETURA DE MICROSSERVIÇOS PARA SISTEMAS WEB

CAMPO GRANDE

2024

KERMAN MENDES PEREIRA JÚNIOR

APLICAÇÃO DE TESTES DE CARGA COM TECNOLOGIAS DE *CACHE* NA ARQUITETURA DE MICROSSERVIÇOS PARA SISTEMAS WEB

Application of Load Testing with Cache Technologies in Microservices Architecture for Web Systems

Monografia apresentada como requisito para obtenção do título de Bacharel em Ciência da Computação do Bacharelado em Ciência da Computação da Universidade Federal De Mato Grosso Do Sul.

Orientador: Prof. Dr. Ricardo Theis Geraldi

CAMPO GRANDE
2024

4.0 Internacional

Esta licença permite download e compartilhamento do trabalho desde que sejam atribuídos créditos ao(s) autor(es), sem a possibilidade de alterá-lo ou utilizá-lo para fins comerciais. Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

AGRADECIMENTOS

Gostaria de expressar minha profunda gratidão a Deus, por iluminar o meu caminho e conceder sabedoria nos momentos de maior necessidade.

Ao Prof. Dr. Ricardo Theis Geraldi, minha sincera admiração e agradecimento pela confiança em meu trabalho, pela orientação, amizade e por compartilhar seu vasto conhecimento comigo.

Aos meus pais, que sempre se dedicaram incansavelmente para me oferecer o melhor e me apoiar em todas as etapas dessa jornada.

RESUMO

MENDES, Kerman. **Aplicação de testes de carga com tecnologias de *cache* na arquitetura de microsserviços para sistemas web**. 2024. 82 f. Monografia (Bacharel em Ciência da Computação) – Universidade Federal De Mato Grosso Do Sul, Campo Grande, 2024.

Contexto: A adoção de arquiteturas de microsserviços na indústria de software aumentou a demanda por tecnologias de *cache* para otimizar o desempenho de sistemas web. A escolha da tecnologia de *cache* certa é um desafio, pois afeta diretamente o desempenho, a eficiência e a escalabilidade desses sistemas. A utilização de tecnologias como Redis e Memcached gera incertezas sobre qual delas é mais adequada para diferentes cenários, dificultando a tomada de decisões e comprometendo a performance de sistemas web. A pesquisa visa comparar essas tecnologias, destacando suas vantagens, limitações e melhores configurações para esses sistemas. **Objetivo:** O objetivo deste trabalho é comparar Redis e Memcached em arquiteturas de microsserviços, analisando suas características e desempenho em sistemas web. **Método:** A pesquisa segue uma abordagem quantitativa/descritiva, focada em quantificar o desempenho das tecnologias de *cache* e descrever suas variações de tempo de resposta e *throughput*. As atividades principais incluem: (i) o desenvolvimento de protótipos de uma API para um sistema de *e-commerce* com funcionalidades de escrita e recuperação de dados de produtos; (ii) a modelagem das características de cada tecnologia de *cache* em *feature models*; e (iii) a execução de testes de carga em um ambiente local utilizando a ferramenta Gatling para avaliar as tecnologias em termos de tempo de resposta e *throughput*. **Resultados:** A análise dos resultados permitiu identificar configurações específicas e casos de uso para cada tecnologia, contribuindo para a compreensão das vantagens e limitações no contexto de microsserviços. Os *feature models* desenvolvidos oferecem uma visão estruturada das tecnologias avaliadas, ajudando arquitetos e desenvolvedores na escolha e implementação de soluções de cache. As principais contribuições incluem: (i) o desenvolvimento de protótipos demonstrando a implementação eficiente de *cache* em um ambiente local; (ii)a organização das características de Redis e Memcached em *feature models*; (iii) a avaliação do impacto das tecnologias no desempenho de microsserviços, utilizando testes de carga.

Palavras-chave: tecnologias de cache; arquitetura de microsserviços; redis; memcached; feature models; otimização de desempenho; engenharia de Software.

ABSTRACT

MENDES, Kerman. **Application of Load Testing with Cache Technologies in Microservices Architecture for Web Systems**. 2024. 82 f. Monografia (Bacharel em Ciência da Computação) – Universidade Federal De Mato Grosso Do Sul, Campo Grande, 2024. Título original: Aplicação de testes de carga com tecnologias de *cache* na arquitetura de microsserviços para sistemas web

Context: The increasing adoption of microservices architectures in the software industry has led to a higher demand for caching technologies to optimize the performance of web systems. Choosing the right caching technology is crucial as it directly affects performance, efficiency, scalability, and resource consumption in microservices. The widespread use of technologies like Redis and Memcached has created uncertainties about which is best suited for various scenarios. This lack of clarity complicates decision-making for architects and developers, potentially impacting the performance and efficiency of systems. This research aims to provide a comparative analysis of Redis and Memcached, clarifying their advantages, limitations, and optimal configurations for web applications. **Objective:** This study compares Redis and Memcached within microservices architectures, analyzing their features and performance in typical web system scenarios. **Method:** The following activities were conducted: (i) development of API prototypes representing an e-commerce system with product data writing and retrieval; (ii) detailed modeling of key characteristics of each caching technology through feature models, which help organize and visualize Redis and Memcached functionalities; and (iii) load tests using the Gatling tool to evaluate performance metrics such as response time and throughput. **Results:** The results of the tests and scenario analysis led to the identification of specific configurations and use cases for each technology. This contributed to a better understanding of their advantages and limitations within microservices. Additionally, the developed feature models offer a structured view of the evaluated technologies, helping architects and developers make informed decisions on implementing caching solutions in microservices architectures. This study provides a foundational body of knowledge supporting the selection and configuration of caching technologies in microservices-based systems.

Keywords: caching technologies; microservices architecture; Redis; Memcached; feature models; performance optimization; software engineering.

LISTA DE FIGURAS

Figura 1 – Etapas do método de pesquisa 15
Figura 2 – Topologia básica do estilo de arquitetura de microsserviços 22
Figura 3 – Modelo de qualidade para produtos de software da ISO/IEC 25010 24
Figura 4 – Funcionalidade X Desempenho 26
Figura 5 – Taxonomia dos tipos de *cache* 29
Figura 6 – Características e funcionalidades de alguns bancos de dados e servidores de cache 31
Figura 7 – Cache no Gateway da API 32
Figura 8 – Comparação de Uso de Cache com e sem Memcached 34
Figura 9 – Feature Model de E-commerce 37
Figura 10 – Latência X Receita Capturada 38
Figura 11 – Processo de teste de carga 40
Figura 12 – Diagrama de *Features* da Tecnologia de *Cache* Redis 46
Figura 13 – Diagrama de *Features* da Tecnologia de *Cache* Memcached 47
Figura 14 – Diagrama de *Features* da Tecnologia de *Cache* Memcached com foco em *Strings* 49
Figura 15 – Diagrama de *Features* da Tecnologia de *Cache* Redis com foco em *Strings* 51
Figura 16 – Configuração do Cliente Redis 57
Figura 17 – Função para Recuperação de Produtos com Cache Redis 58
Figura 18 – Configuração do Cliente Memcached 58
Figura 19 – Função para Recuperação de Produtos com Cache Memcached 59
Figura 20 – Requisições do Log do servidor da API 62
Figura 21 – Plano de teste gerado com a Ferramenta Gatling 62
Figura 22 – Execução do *script* de teste via terminal. 63
Figura 23 – Progresso de execução do *script* de teste 63
Figura 24 – Tempo de Resposta Médio por Número de Threads para 10 mil registros 66
Figura 25 – Tempo de Resposta Médio por Número de *Threads* para 20 mil registros 68
Figura 26 – Tempo de Resposta Médio por Número de *Threads* para 40 mil registros 70

LISTA DE TABELAS

Tabela 1 – Especificações de Hardware e Software 54
Tabela 2 – Quantidade de Registros e Tamanho Total de Cache 60
Tabela 3 – Análise de Requisições por Segundo 71

SUMÁRIO

1 INTRODUÇÃO 10
1.1 CONTEXTO E MOTIVAÇÃO 10
1.2 OBJETIVOS 11
1.2.1 Objetivo Geral 11
1.2.2 Objetivos Específicos 11
1.3 JUSTIFICATIVA 12
1.4 ESTRUTURA DO TRABALHO 12
2 ESTRUTURAÇÃO DA PESQUISA 14
2.1 PLANEJAMENTO 16
2.2 EXECUÇÃO 16
2.2.1 Definir e configurar ambientes de testes 17
2.2.2 Definir casos de teste 17
2.2.3 Implementar testes 17
2.2.4 Executar testes 18
2.3 ANÁLISE E INTERPRETAÇÃO 18
3 REVISÃO DA LITERATURA 19
3.1 ARQUITETURA DE SOFTWARE 19
3.2 ARQUITETURA DE MICROSSERVIÇOS 20
3.3 REQUISITOS NÃO-FUNCIONAIS 23
3.4 DESEMPENHO DE SOFTWARE 24
3.5 PROTOCOLO HTTP 26
3.6 CACHE NA WEB 28
3.7 CACHE DISTRIBUÍDO REDIS 30
3.8 CACHE DISTRIBUÍDO MEMCACHED 32
3.9 FEATURES E MODELAGEM DE FEATURES 35
3.10 TESTE DE DESEMPENHO 37
3.11 TESTE DE CARGA 39
3.12 GATLING 41
3.13 MÉTRICAS DE TESTE DE CARGA PARA AVALIAÇÃO DO *CACHE* 42
3.14 TRABALHOS RELACIONADOS 43
4 FEATURE MODELS: REDIS E MEMCACHED 45
4.1 FEATURE MODELS 45
4.2 CRITÉRIOS PARA SELEÇÃO DAS TECNOLOGIAS DE *CACHE* 47
4.3 CONFIGURAÇÕES DE ESTRUTURA DE DADOS PARA *CACHE* 49
5 IMPLEMENTAÇÃO DOS TESTES DE DESEMPENHO DE *CACHE* 53
5.1 CONSIDERAÇÕES INICIAIS 53
5.2 DEFINIR E CONFIGURAR AMBIENTE DE TESTES 53
5.2.1 Estrutura do Ambiente 54
5.2.2 Gatling 55
5.2.3 Implementação do Código para Redis 56
5.2.4 Implementação do Código para Memcached 56

5.3 DEFINIÇÃO DO PLANO DE TESTE 57
5.3.1 Plano de Teste Baseado no YCSB 58
5.3.2 Implementar testes 59
5.3.2.1 Definição do Tamanho e Quantidade de Dados 60
5.3.3 Produção e Inserção dos Dados no Banco de Dados 61
5.4 EXECUTAR TESTES 61
5.5 LIMITAÇÕES DO AMBIENTE DE TESTE LOCAL 64
6 RESULTADOS E DISCUSSÕES 65
6.1 CONSIDERAÇÕES INICIAIS 65
6.2 ANÁLISE DE DESEMPENHO DO REDIS E MEMCACHED 65
6.2.1 Descrição Geral do Comportamento de Redis e Memcached 65
6.2.2 Análise de 10 mil Registros 65
6.2.3 Análise de 20 mil Registros 66
6.2.4 Análise de 40 mil Registros 68
6.3 ANÁLISE DE *THROUGHPUT* (REQUISIÇÕES POR SEGUNDO) 70
6.4 DISCUSSÃO DOS PRINCIPAIS RESULTADOS 72
7 CONCLUSÃO 75
REFERÊNCIAS 78

1 INTRODUÇÃO

1.1 CONTEXTO E MOTIVAÇÃO

À medida que as empresas expandem suas operações digitais, seus aplicativos web enfrentam desafios cada vez maiores de escalabilidade e desempenho. O dimensionamento de aplicativos com alta carga de usuários, dados e transações impõe desafios específicos, como a identificação e resolução de gargalos de desempenho, bem como a superação de limitações de infraestrutura, como capacidade de rede, recursos de hardware e gerenciamento de carga (WTT SOLUTIONS[1], 2024).

Nesse cenário, a arquitetura de microsserviços surge como uma solução promissora. Ao dividir aplicações complexas em serviços menores, independentes e autônomos, essa abordagem arquitetural oferece maior flexibilidade, escalabilidade e resiliência (UEDA *et al.*, 2016). Um exemplo notável é a Netflix, que adotou essa arquitetura para garantir a independência e autonomia de suas equipes e serviços. Essa transformação permitiu que a Netflix reformulasse seu pipeline de processamento de vídeo, migrando de uma estrutura monolítica rígida para uma plataforma baseada em microsserviços, chamada Cosmos (NETFLIX TECHBLOG[2], 2024).

Sistemas que lidam com grandes volumes de dados, como redes sociais, *e-commerce* e serviços de *streaming*, têm recorrido ao uso de *caches* em memória para reduzir a latência e melhorar o desempenho (DECANDIA *et al.*, 2007). Soluções de *cache*, como Redis e Memcached, armazenam dados temporariamente na RAM, tornando-os prontamente disponíveis para recuperação rápida.

O Redis suporta estruturas de dados complexas, como listas, conjuntos e hashes, e pode oferecer armazenamento persistente opcional. Já o Memcached adota uma abordagem mais simples, otimizada para cenários em que a prioridade é a rápida recuperação de dados em *cache* (REDIS, 2023; MEMCACHED, 2023).

A arquitetura de microsserviços permite que cada serviço utilize uma tecnologia de *cache* dedicada, otimizada para suas necessidades específicas de dados e carga de trabalho (DRAGONI *et al.*, 2017). Essa flexibilidade abre a possibilidade de alcançar níveis mais altos de desempenho, escalabilidade e resiliência em aplicações web modernas,

[1] https://wtt-solutions.com/blog/top-challenges-in-scaling-high-load-web-applications

[2] https://netflixtechblog.com/rebuilding-netflix-video-processing-pipeline-with-microservices-4e5e6310e359

além da adoção dos microsserviços.

A escolha da solução de *cache* mais adequada, entretanto, depende de uma análise detalhada dos requisitos do sistema, como volume de dados, padrões de acesso, tipos de dados e tolerância a falhas. Uma escolha incorreta pode afetar o desempenho, aumentar a complexidade e dificultar a escalabilidade.

Estudos comparativos utilizando o benchmark YCSB mostram variações de desempenho significativas entre Redis e Memcached, dependendo do padrão de acesso aos dados. O Redis obteve melhor desempenho em operações que envolvem estruturas de dados complexas, com *throughput* até 35% superior, enquanto o Memcached é mais eficiente em operações simples de chave-valor, com menor latência (CAO *et al.*, 2016; COOPER *et al.*, 2010).

Apesar de já existirem estudos sobre o desempenho de Redis e Memcached, muitos se concentram em ambientes monolíticos ou em cenários específicos. Este trabalho busca preencher a lacuna de uma análise comparativa de desempenho no contexto de microsserviços, considerando diferentes cargas de trabalho e configurações, para ajudar desenvolvedores e arquitetos na escolha da melhor solução de *cache* para suas necessidades.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é realizar uma análise comparativa de desempenho entre as tecnologias de cache Redis e Memcached em uma arquitetura de microsserviços, com foco em como cada tecnologia opera em termos de desempenho em diferentes padrões de acesso a dados e cargas de trabalho. A pesquisa busca apresentar uma avaliação quantitativa, com base nos resultados dos testes realizados, sobre o desempenho de cada tecnologia em relação ao tempo de resposta e *throughput* em microsserviços. A pesquisa adota uma abordagem quantitativa/descritiva, pois visa quantificar o desempenho das tecnologias de *cache* em diferentes cenários e descrever suas variações em tempo de resposta e *throughput* com base nas medições realizadas.

1.2.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

1. Mensurar e comparar o desempenho de tecnologias de *cache* (Redis e Memcached) em operações de leitura, utilizando diferentes quantidades de *threads* e volume de dados para identificar a solução mais adequada em situações de alto desempenho.

2. Avaliar a capacidade de ambas as tecnologias de *cache*, com o objetivo de verificar seu desempenho e adequação para diferentes padrões de carga em ambientes de microsserviços.

3. Desenvolver *feature models* que ilustrem as principais características e capacidades de Redis e Memcached, para facilitar e guiar a escolha de tecnologia para arquiteturas que exijam alto desempenho.

1.3 JUSTIFICATIVA

A realização desta pesquisa surge da crescente demanda por otimização no desempenho em arquiteturas de microsserviços, onde escalabilidade e eficiência no gerenciamento de dados são aspectos relevantes. Tecnologias de *cache*, como Redis e Memcached, desempenham um papel fundamental nesse cenário, ajudando a reduzir a latência e aumentar a taxa de processamento. Apesar de sua ampla adoção, ainda há pouca análise detalhada na literatura sobre como essas duas tecnologias se comportam em diferentes condições de uso, particularmente em contextos de microsserviços. Nesse contexto, surge a necessidade de comparar essas duas tecnologias para entender suas diferenças e identificar qual delas é mais adequada para atender aos requisitos não funcionais de desempenho em microsserviços, como latência, *throughput* e escalabilidade.

Em resumo, a pesquisa justifica-se pela necessidade de melhorar a compreensão e a escolha das tecnologias de *cache* adequadas para arquiteturas de microsserviços. A eficiência do *cache* impacta diretamente o desempenho, a escalabilidade e o consumo de recursos desses sistemas, tornando essencial a comparação aprofundada entre tecnologias como Redis e Memcached. Ao abordar esse problema, o trabalho busca fornecer dados concretos e análises práticas que orientem arquitetos e desenvolvedores na seleção das melhores soluções de *cache* para diferentes cenários, contribuindo para a melhoria contínua das arquiteturas de microsserviços de alta performance.

1.4 ESTRUTURA DO TRABALHO

Este trabalho está estruturado da seguinte maneira:

1. **Capítulo 1:** Apresenta a introdução da pesquisa, enfatizando a importância do desempenho e da escalabilidade em microsserviços e a relevância das tecnologias de *cache*, como Redis e Memcached, nesse cenário. São descritos o objetivo geral, os objetivos específicos e a estrutura do trabalho.

2. **Capítulo 2:** Apresenta a estruturação da pesquisa expondo o método de pesquisa e as etapas adotadas para realizar a comparação entre Redis e Memcached, incluindo detalhes sobre o ambiente de teste, a configuração de *cache*, as ferramentas e o benchmark YCSB, com foco no desempenho das operações de leitura e escrita.

3. **Capítulo 3:** Apresenta a revisão da literatura com conceitos de tecnologias de cache distribuído e arquitetura de microsserviços. Esse capítulo também apresenta estudos comparativos prévios sobre o desempenho de Redis e Memcached, destacando fatores críticos para o desempenho e escalabilidade.

4. **Capítulo 4:** Apresenta os *feature models* que representam as *features* técnicas de Redis e Memcached, detalhando configurações de desempenho e aspectos de implementação. O capítulo explora as configurações práticas utilizadas para análise, com foco na estrutura de dados de *string* e seu gerenciamento em ambas as tecnologias.

5. **Capítulo 5:** Apresenta a implementação e execução dos testes de desempenho das tecnologias de *cache* Redis e Memcached em uma arquitetura de microsserviços. O capítulo inclui a configuração da infraestrutura e o ambiente de testes, a definição dos casos de teste com base no Yahoo Benchmark, e a execução prática dos testes, que utilizam *scripts* automatizados para simular múltiplos usuários interagindo com as APIs. Além disso, descreve a execução dos testes em um ambiente conteinerizado com a ferramenta Gatling para coleta de métricas de desempenho, e aborda as limitações do ambiente de teste, como a falta de escalabilidade horizontal.

6. **Capítulo 6:** Apresenta os resultados dos testes de desempenho realizados com as tecnologias de *cache* Redis e Memcached. Os dados coletados incluem métricas de tempo de resposta, throughput e percentis de tempo de resposta, além de uma análise comparativa entre as duas tecnologias. Este capítulo discute as implicações dos resultados para a implementação de cache em arquiteturas de microsserviços, destacando o impacto no desempenho das aplicações.

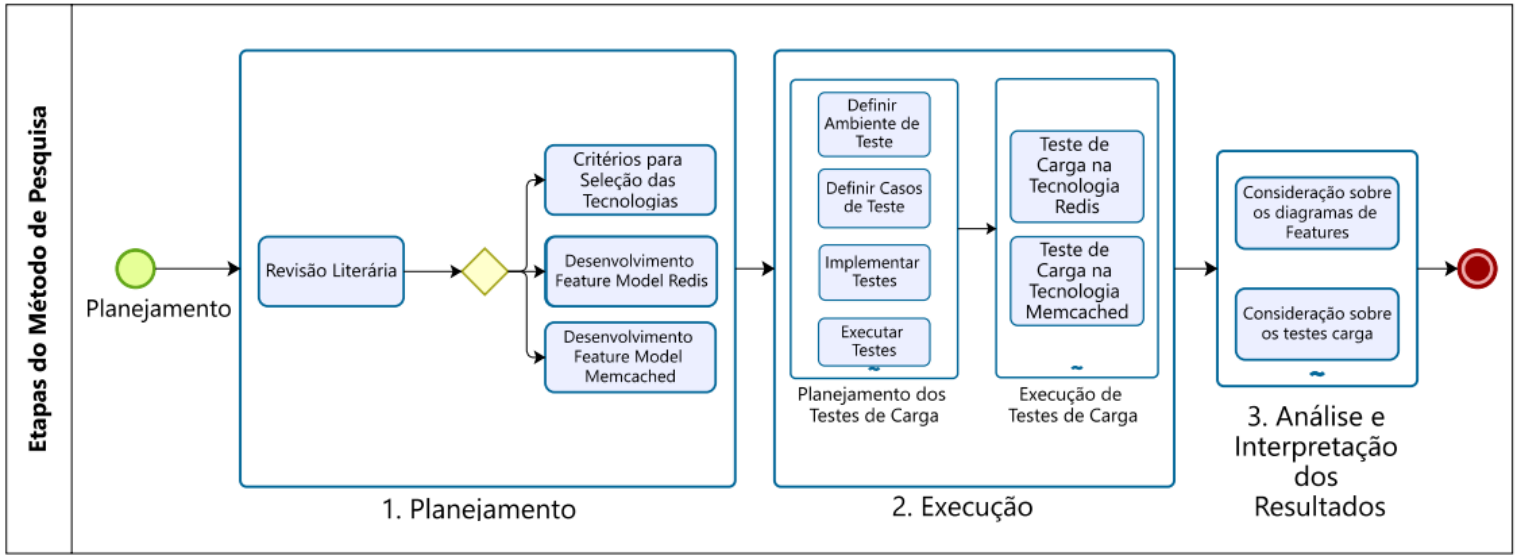
7. **Capítulo 7:** Apresenta as considerações finais acerca deste trabalho, incluindo suas principais limitações, contribuições e trabalhos futuros.

2 ESTRUTURAÇÃO DA PESQUISA

Este trabalho adapta o protocolo de engenharia de software experimental de Wohlin *et al.* (2012), ajustando-o para atender aos objetivos específicos do estudo sobre tecnologias de cache em arquiteturas de microsserviços. A estrutura da pesquisa é dividida em três etapas principais: planejamento, execução, e análise e interpretação.

A Figura 1 apresenta uma representação visual do processo metodológico, enquanto a descrição detalhada de cada etapa é fornecida a seguir.

Figura 1 – Etapas do método de pesquisa



Fonte: (Autoria Própria, 2024).

2.1 PLANEJAMENTO

A primeira etapa do trabalho se baseou no planejamento da pesquisa. As atividades realizadas nesta fase envolvem a revisão da literatura sobre arquitetura de software, abrangendo conceitos como arquitetura de microsserviços, tecnologias de *cache* no contexto web, tecnologias Redis e Memcached, requisitos não funcionais (RNFs), testes de desempenho e de carga, além dos fundamentos sobre *features* e modelagem de *features*.

Uma parte central do planejamento inclui a criação dos *feature models* para as tecnologias Redis e Memcached, utilizando a ferramenta FeatureIDE. Esses modelos são criados para representar as principais *features* das duas tecnologias, com base no padrão de armazenamento chave-valor e no uso de *strings* como formato de armazenamento. Embora ambas as tecnologias adotem essa abordagem de chave-valor, a modelagem de *features* ajuda a identificar as *features* específicas de cada uma no contexto da pesquisa. Além disso, serão desenvolvidos modelos mais específicos focados exclusivamente na estrutura de dados de *strings*, que será utilizada nos testes de desempenho.

2.2 EXECUÇÃO

A pesquisa será realizada por meio de testes de carga aplicados a uma API de um *e-commerce*, com o objetivo de avaliar o desempenho de Redis e Memcached sob condições de tráfego intenso, características de ambientes de microsserviços. Esses testes permitem simular cenários próximos aos reais, analisando métricas como tempo de resposta e *throughput*, e identificando gargalos, como atrasos na recuperação de dados devido à alta concorrência.

A execução seguirá o método de Meier *et al.* (2007), que abrange sete etapas: definição do ambiente, planejamento, configuração, implementação, execução, análise e reexecução dos testes. Este método foi adaptado para integrar algumas etapas, como unificar a definição e configuração do ambiente e consolidar o planejamento e modulação dos casos de teste, otimizando o processo.

A escolha do método de Meier justifica-se pela sua aplicabilidade prática e por ser amplamente reconhecida na literatura de engenharia de software, para a aplicação de testes de desempenho. Esse método é essencial para compreender o desempenho das tecnologias em estudo e garantir resultados consistentes.

2.2.1 Definir e configurar ambientes de testes

Neste trabalho, será configurado um único ambiente de desenvolvimento. Esse ambiente será configurado localmente na máquina do autor, utilizando a ferramenta Docker para garantir consistência nos testes. Esse ambiente simulará as condições de uso de sistemas de um *e-commerce*, embora em menor escala, e permitirá validar as tecnologias Redis e Memcached em um ambiente controlado. Esse ambiente de desenvolvimento será o único utilizado, e ele foi escolhido por razões práticas e econômicas, já que ele atende à necessidade de simular cenários de carga e desempenho para fins de validação experimental.

2.2.2 Definir casos de teste

Esta etapa da execução envolve a definição dos casos de teste e dos critérios de sucesso, sendo que os casos de teste são baseados nas funcionalidades das tecnologias Redis e Memcached, visando simular operações típicas de um sistema de um *e-commerce*. Cada caso de teste foi projetado com base nas métricas estabelecidas previamente, como tempo de resposta médio e *throughpt*, que servirão de base para a avaliação comparativa das tecnologias.

2.2.3 Implementar testes

Para implementar os testes, serão utilizados *scripts* desenvolvidos utilizando a ferramenta Gatling, que permite a automação de requisições simulando múltiplas *threads* (usuários virtuais) interagindo com as API's Redis e Memcached. Os testes serão organizados em cenários de carga que simulam diferentes condições de uso, como aumento picos de tráfego e uso contínuo. O foco será testar operações típicas de um sistema de *e-commerce*, como a recuperação de produtos do *cache* e a execução de consultas frequentes.

Passos da implementação incluem:

- Desenvolvimento de um plano de teste para simular operações de leitura (GET) *cache* por meio da ferramenta Gatling.
- Configuração de diferentes níveis de carga, simulando volumes variados de usuários (usuários virtuais) e quantidade de dados baseados na literatura.

2.2.4 Executar testes

A execução dos testes será realizada no ambiente de desenvolvimento Docker configurado, utilizando a ferramento Gatling. Durante a execução, serão coletadas métricas de desempenho, como tempo de resposta e *throughput*. Esses dados serão registrados para posterior análise comparativa entre as tecnologias Redis e Memcached, garantindo que as diferentes condições de carga sejam monitoradas.

2.3 ANÁLISE E INTERPRETAÇÃO

Nesta etapa, apresentaremos a análise e interpretação dos resultados obtidos. As atividades incluem: (i) análise quantitativa/descritiva comparando as tecnologias Redis e Memcached utilizadas, suas diferenças, pontos fortes e fracos em relação ao desempenho; (ii) análise dos *feature models* desenvolvidos, com base na capacidade de representar o requisito não funcional de desempenho das arquiteturas. Após isso, são realizadas discussões sobre a aplicação das tecnologias e dos *feature models* no desenvolvimento de sistemas *e-commerce* com foco em desempenho.

A etapa de conclusão do trabalho sintetiza os principais achados da pesquisa, destacando as contribuições para o campo da arquitetura de software e propondo direções para trabalhos futuros. Este processo estruturado permite uma abordagem sistemática e compreensiva para a pesquisa, resultando em uma análise comparativa detalhada entre Redis e Memcached, contribuindo diretamente para o campo da otimização de *cache* em aplicações *web*.

3 REVISÃO DA LITERATURA

3.1 ARQUITETURA DE SOFTWARE

Perry e Wolf (1992) propuseram uma tripla para definir arquitetura de software:

Arquitetura de Software = Elementos, Forma, Fundamentação

Os Elementos são unidades de processamento e dados que compõem o sistema. A Forma diz respeito às propriedades e relações que conectam esses elementos entre si. A Fundamentação é a lógica e as razões por trás da estrutura adotada na arquitetura (Perry e Wolf, 1992).

Expandindo essa visão, Garlan e Shaw (1994) definem a arquitetura de software como "a descrição dos elementos a partir dos quais os sistemas são construídos, interações entre esses elementos, padrões que guiam sua composição e restrições sobre esses padrões".

Mais recentemente, a ISO/IEC/IEEE 42010:2010 estabeleceu uma definição padronizada descrevendo a arquitetura de software como a estrutura básica de um sistema composta por seus módulos, relacionamentos entre os módulos, o ambiente em que operam e os princípios que orientam seu desenvolvimento e evolução. Para Valente (2020), arquitetura de software engloba decisões importantes de uma arquitetura, como a seleção da linguagem de programação e do banco de dados (VALENTE, 2020).

Seguindo essa abordagem de arquitetura de mais alto nível, Bass *et al.* (2010) oferecem uma análise detalhada dos componentes que integram uma arquitetura de software, que incluem:

- **Estrutura de módulos (*Module structures)*:** Estrutura de módulos (Module structures): Divisão do sistema em unidades modulares de código ou dados, como classes, camadas ou funcionalidades. Define a responsabilidade funcional de cada componente (BASS *et al.*, 2012).
- **Componentes e Conectores (*Components and Connectors)***: Conectores atuam como intermediários na comunicação entre componentes, facilitando interações como requisição-resposta, sincronização de processos e pipes. Já os componentes são unidades de processamento, podendo ser serviços, clientes, servidores ou filtros (BASS *et al.*, 2012).
- **Alocação de funcionalidade (*Allocation of functionality)***: Envolve decisões

> sobre a interação do sistema com estruturas externas ao software, como CPUs, sistemas de arquivos, redes e equipes de desenvolvimento, que gerenciam a relação entre os elementos de software (BASS *et al.*, 2012).

Assim, a arquitetura de software possibilita a criação de representações do sistema que se deseja desenvolver. Essas representações ajudam a gerenciar a complexidade envolvida na definição de requisitos e na busca por um design apropriado (ALBIN, 2003).

Para elaborar essas representações, Buschmann *et al.* (1996) introduzem o conceito de estilo arquitetural. Um estilo arquitetural é descrito como um padrão recorrente de elementos e suas relações dentro de um sistema (BUSCHMANN *et al.*, 1996). Taylor *et al.* (2009) ampliam essa definição, caracterizando um estilo arquitetural como "uma especialização de elementos e tipos de relações, acompanhada de um conjunto de restrições sobre como esses elementos podem ser utilizados".

Um exemplo de estilo arquitetural é a arquitetura monolítica, caracterizada por ser uma estrutura unificada onde todos os módulos e funcionalidades de uma aplicação são implementados como uma única unidade lógica (DRAGONI *et al.*, 2017). Nesse tipo de arquitetura, a aplicação é desenvolvida como um programa autônomo, geralmente utilizando uma única linguagem de programação, *framework* e *stack* tecnológica, o que permite aos desenvolvedores trabalharem em um ambiente coeso com uma base de código única, facilitando o processo de desenvolvimento (RICHARDSON, 2018; FOWLER e LEWIS, 2014).

Em contraste à arquitetura monolítica, surgem novas arquiteturas alternativas, como a arquitetura de microsserviços, definida por Newman (2015) como "pequenos serviços autônomos que trabalham juntos". A arquitetura de microsserviços consiste em desenvolver uma única aplicação como um conjunto de pequenos serviços, cada um executando em seu próprio processo e se comunicando com mecanismos leves, frequentemente uma API de recurso HTTP"(FOWLER E LEWIS, 2014).

A próxima seção apresenta em detalhes os princípios e características da arquitetura de microsserviços.

3.2 ARQUITETURA DE MICROSSERVIÇOS

Em linhas gerais, a arquitetura de microsserviços é definida como o desenvolvimento de uma aplicação única a partir de pequenos serviços independentes,

executando em processos isolados e comunicando-se tipicamente por API's HTTP (NEWMAN, 2015; DRAGONI *et al.*, 2017).

Segundo Richardson (2018), a arquitetura de microsserviços surge como uma solução escalável para aplicações modernas. Originada dos princípios da arquitetura orientada a serviços (SOA), a adoção de DevOps e a busca por escalabilidade impulsionaram o desenvolvimento dessa arquitetura (FOWLER e LEWIS, 2014). Suas características principais incluem serviços fracamente acoplados, alta coesão, automatização de implantação e descentralização de processamento (TAIBI; LENARDUZZI, 2018). A adoção de microsserviços tem se popularizado em domínios como comércio eletrônico, *streaming* de vídeo e sistemas financeiros. Empresas como a Amazon[1] e a Netflix[2] foram pioneiras na adoção dessa arquitetura (RICHARDSON, 2018). A Amazon, por exemplo, migrou de uma arquitetura monolítica para microsserviços, o que permitiu que as equipes desenvolvessem e implantassem serviços de forma independente, aumentando a agilidade e a inovação (NEWMAN, 2015). De maneira semelhante, a Netflix adota microsserviços para disponibilizar seu conteúdo de forma escalável e resiliente, lidando com picos de tráfego e falhas de forma transparente para os usuários (RICHARDSON, 2018).

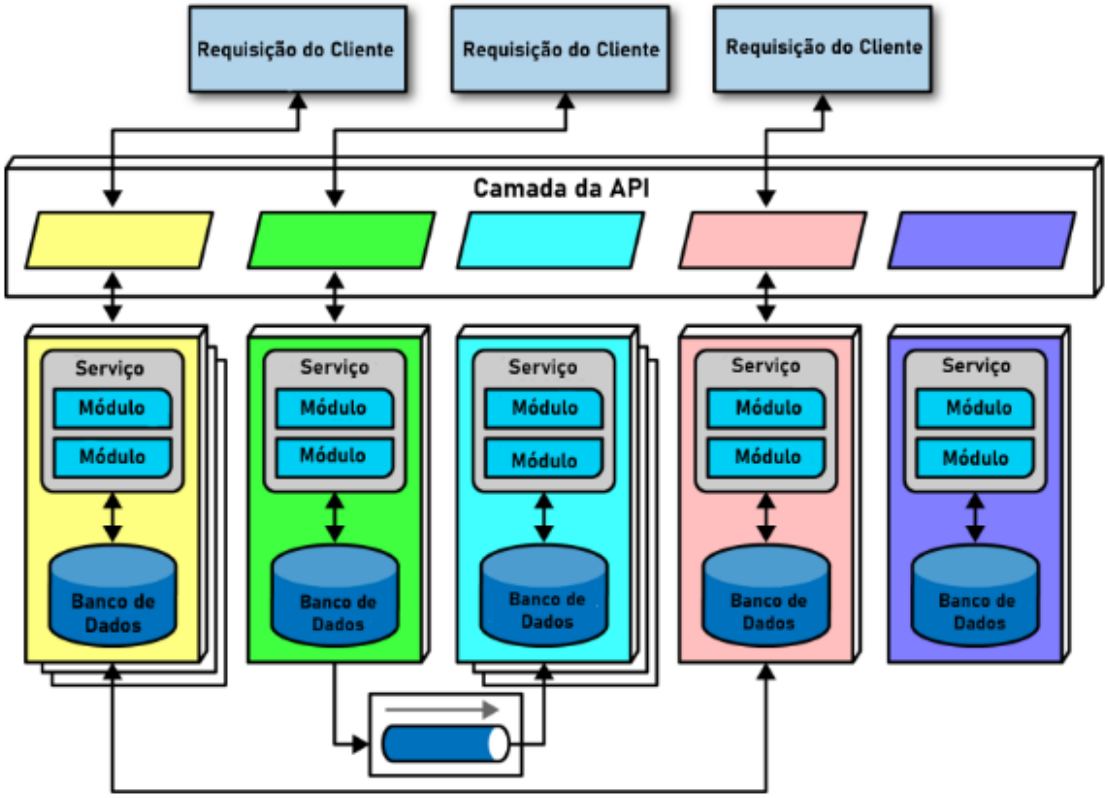
Destaca-se ainda a relação desse estilo arquitetural com a Lei de Conway (1968), segundo a qual sistemas de software tendem a espelhar a estrutura organizacional de suas empresas. Assim, os microsserviços têm sido amplamente adotados por grandes empresas de tecnologia, as quais comumente contam com centenas de equipes de desenvolvimento distribuídas e autônomas (VALENTE, 2020).

Conforme ilustrado na Figura 2, devido à sua natureza de propósito único, o tamanho de cada serviço em uma arquitetura de microsserviços é muito menor do que os componentes equivalentes em uma arquitetura monolítica.

Um estudo conduzido por Taibi *et al.* (2018) buscou identificar padrões de arquitetura comumente utilizados no desenvolvimento de aplicações baseadas em microsserviços, conforme relatado na literatura. Entre os principais padrões destacados estão: **API *Gateway***, que funciona como ponto de entrada único do sistema, direcionando as solicitações dos clientes para os microsserviços apropriados. Esse padrão pode invocar múltiplos microsserviços e agregar os resultados, aumentando o desempenho do sistema ao simplificar interações e reduzir o número de solicitações feitas por cliente.

[1]Amazon: https://amazon.com.br
[2]Netflix: https://netflix.com.br

Figura 2 – Topologia básica do estilo de arquitetura de microsserviços



Fonte: (Adaptado de RICHARDS e FORD, 2021).

Outro padrão é o ***Service Discovery***, que permite a localização dinâmica de instâncias de serviços em ambientes distribuídos. Essa abordagem gerencia a complexidade da comunicação entre microsserviços ao atribuir dinamicamente endereços IP e locais de rede para instâncias distribuídas em containers ou máquinas virtuais.

Villamizar *et al.* (2015) conduziram um estudo sobre a arquitetura de microsserviços com ênfase no requisito não funcional de desempenho. Requisitos não funcionais especificam critérios utilizados para julgar a operação de um sistema. Glinz (2007) propõe uma definição mais abrangente, descrevendo RNFs como atributos de qualidade, restrições ou requisitos de processo que um sistema deve atender.

Complementando essa análise, Villamizar *et al.* (2015) realizaram um estudo sobre a arquitetura de microsserviços com foco no requisito não funcional de desempenho. Segundo Glinz (2007), requisitos não funcionais (RNFs) englobam atributos de qualidade, restrições ou requisitos de processo que um sistema deve atender, sendo utilizados como critérios para avaliar sua operação. No contexto de microsserviços, Villamizar *et al.* (2015) identificaram que o desempenho é influenciado por fatores como: latência de

rede, que pode ser amplificada pelo uso de *gateways* e pela comunicação via REST, devido à necessidade de mais servidores e à lógica de negócios distribuída; configuração e escalabilidade, onde a capacidade de escalar microsserviços e gateways de forma independente, seja vertical (aumentando o poder de processamento de instâncias) ou horizontal (adicionando mais instâncias), é essencial para lidar com cargas de trabalho variáveis e picos de demanda; e o impacto da distribuição do sistema, que pode introduzir falhas, timeouts e complexidade adicional em transações distribuídas, afetando negativamente o desempenho geral.

3.3 REQUISITOS NÃO-FUNCIONAIS

Requisitos Não-Funcionais (RNF's) se referem a maneira em que um software irá operar sob determinadas circunstâncias e são relacionados ao desempenho, disponibilidade, usabilidade, dentre outros (POHL; RUPP, 2015). Na arquitetura de software, um sinônimo de RNFs são "atributos de qualidade", que são entendidos como um conjunto de propriedades relacionadas ao conceito de qualidade de software (BARBACCI *et al.*, 1995).

De acordo com Glinz (2007), os RNF's são definidos como "requisitos que descrevem os aspectos não comportamentais de um sistema, capturando as propriedades e restrições sob as quais o sistema deve operar, sendo considerados críticos para o sucesso de um projeto". Esses requisitos são fundamentais para garantir a viabilidade e a eficiência de uma arquitetura, influenciando a escolha de padrões de projeto, tecnologias e estratégias de implementação (BASS *et al.*, 2012).

Segundo a norma ISO/IEC 25010, as qualidades não funcionais de um software abrangem oito características principais, descritas na figura 3 abaixo, cada uma com suas subdivisões, totalizando 31 subcaracterísticas. Este conjunto de características desempenha um papel crucial na garantia de qualidade no software desenvolvido destacando o desempenho ("Eficiência no desempenho") como uma das características essenciais neste conjunto (REINEHR, 2020).

As RNFs ditam o sucesso no desenvolvimento de um software, determinando a aceitação do sistema pelos usuários (CYSNEIROS; LEITE, 2004). Ignorar os RNFs pode resultar em sistemas que não conseguem atender às expectativas dos *stakeholders*, mesmo que os requisitos funcionais sejam cumpridos (CHUNG *et al.*, 2009).

A definição e priorização dos requisitos de desempenho devem considerar as

Figura 3 – Modelo de qualidade para produtos de software da ISO/IEC 25010.

Fonte: (REINEHR, 2020).

necessidades específicas do sistema, o ambiente em que ele será aplicado e as expectativas dos usuários. Esses fatores são fundamentais para garantir que o software atenda aos objetivos de performance e seja aceito pelos stakeholders envolvidos (WIEGERS; BEATTY, 2013).

Pode-se citar o RNF de desempenho, que está relacionado à capacidade do sistema de oferecer tempos de resposta e taxas de processamento apropriados sob condições operacionais específicas (CHUNG *et al.*, 2012). Esse requisito está estreitamente associado à escalabilidade, que se refere à capacidade do sistema de suportar um aumento na carga de trabalho ou no número de usuários sem que haja uma degradação significativa no desempenho (BASS *et al.*, 2012).

3.4 DESEMPENHO DE SOFTWARE

O desempenho é definido como a capacidade de um sistema de software de fornecer um desempenho apropriado em relação à quantidade de recursos utilizados, sob condições estabelecidas (ISO/IEC 25010, 2011).

Bass, Clements e Kazman (2012) explicam que o desempenho de um sistema

envolve o cumprimento de suas funções dentro de restrições de tempo, precisão e uso de memória, considerando não apenas a velocidade de resposta, mas também a eficiência no uso de recursos computacionais (BASS; CLEMENTS; KAZMAN, 2012). Observa-se que o desempenho não se limita apenas à velocidade de resposta, mas engloba também a utilização eficiente de recursos computacionais (SMITH; WILLIAMS, 2002). O desempenho deve ser considerado em termos de responsividade (quão rápido o sistema responde a requisições) e escalabilidade (capacidade do sistema de manter o desempenho sob carga crescente) (SMITH; WILLIAMS, 2002).

Albin (2003) aborda duas visões conflitantes frequentemente debatidas em relação ao desempenho em detrimento da funcionalidade de uma aplicação:

- **Visão 1:** Propõe que a funcionalidade do software deve ser a prioridade inicial, com melhorias de desempenho sendo implementadas em estágios posteriores. Essa abordagem permite um lançamento mais rápido e evita otimizações desnecessárias (ALBIN, 2003).
- **Visão 2:** Defende a consideração do desempenho em todas as etapas do desenvolvimento, desde o design até a implementação. Ignorar o desempenho desde o início pode resultar em problemas graves e custosos (ALBIN, 2003).

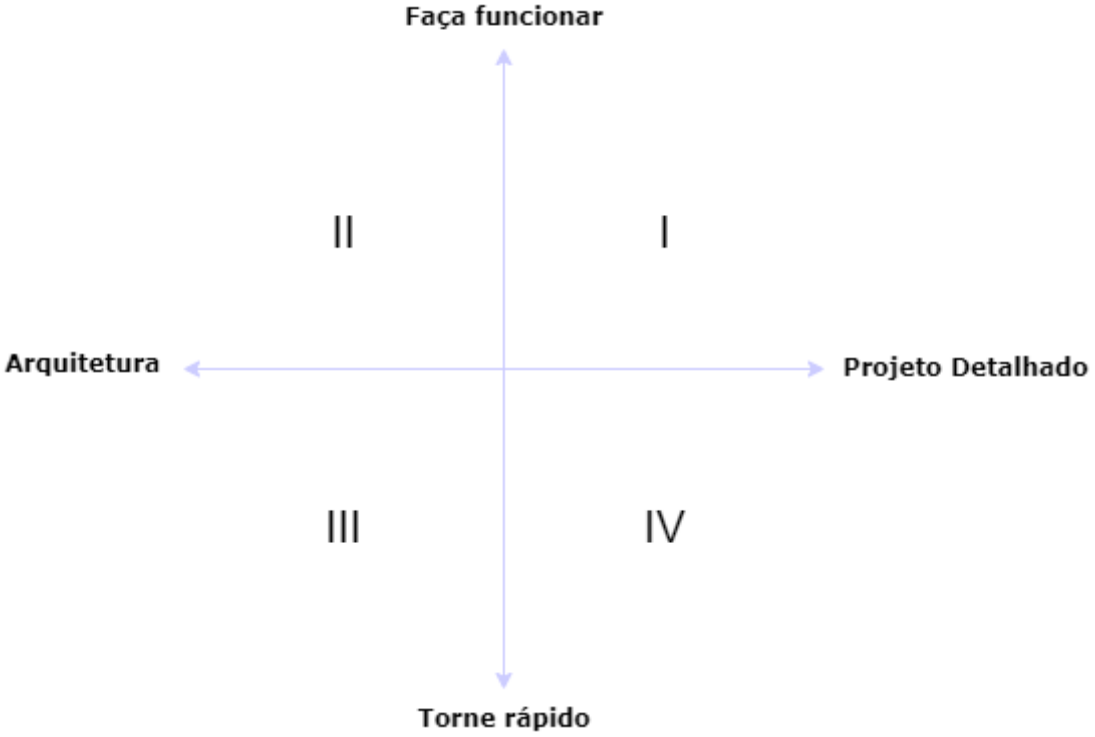
A Figura 4 propõe uma abordagem equilibrada que combina os pontos fortes de ambas as visões, destacando a importância de definir métricas de desempenho durante a fase de escolha da arquitetura adequada, representada pelos quadrantes II e III, bem como de realizar testes de desempenho em componentes menores durante a fase do projeto detalhado (quadrantes I e IV).

De acordo com Smith e Williams (2002), um sistema que não atinge um desempenho adequado pode levar à frustração dos usuários, perda de eficiência e até ao abandono completo do sistema. O gerenciamento eficaz de recursos computacionais é fundamental para evitar esses problemas (SMITH; WILLIAMS, 2002).

Um desempenho inadequado pode gerar custos adicionais, como a necessidade de melhorar o hardware ou até mesmo perdas financeiras devido ao tempo que o sistema fica fora do ar. É essencial que os desenvolvedores incorporem o desempenho como um atributo de qualidade desde as fases iniciais do projeto e adotem práticas que garantam que o sistema cumpra os requisitos estabelecidos (SMITH; WILLIAMS, 2002).

Menascé e Almeida (2002) destacam que o desempenho de uma aplicação depende

Figura 4 – Funcionalidade X Desempenho.



Fonte: (Adaptado de ALBIN, 1998).

de múltiplos fatores, como a arquitetura, a qualidade do código, a infraestrutura utilizada e a carga de trabalho que o sistema precisa gerenciar (MENASCÉ; ALMEIDA, 2002).

No contexto do desempenho de aplicações web, é fundamental entender o protocolo HTTP (*Hypertext Transfer Protocol*), que possibilita a comunicação entre clientes e servidores na web. O HTTP é a base para a transferência de vários tipos de dados, como hipertexto, imagens e vídeos (FIELDING *et al.*, 1999).

3.5 PROTOCOLO HTTP

O HTTP é o protocolo responsável pela comunicação na *World Wide Web* (WWW), determinando o formato e a transmissão de mensagens entre clientes, como navegadores, e servidores (FIELDING *et al.*, 1999). Operando na camada de aplicação sobre o TCP/IP, o HTTP possui uma natureza stateless, o que significa que cada requisição é processada de forma isolada, sem que haja retenção de informações sobre as requisições anteriores (BELSHE; PEON; THOMSON, 2015).

- As versões principais do HTTP incluem:

  - **HTTP/1.0 (1996):** Primeira versão padronizada, introduzindo os métodos GET, POST e HEAD.
  - **HTTP/1.1 (1997):** Trouxe melhorias significativas, incluindo conexões persistentes e pipelining.
  - **HTTP/2 (2015):** Introduziu multiplexação de streams, compressão de cabeçalhos e server push.
  - **HTTP/3 (2022):** Baseado no QUIC, oferece melhor desempenho em redes com alta latência.

- O HTTP utiliza um modelo de requisição-resposta. Os principais métodos de requisição incluem (FIELDING *et al.*, 2014):
  - **GET:** Solicita um recurso específico.
  - **POST:** Submete dados para serem processados por um recurso.
  - **PUT:** Atualiza um recurso existente.
  - **DELETE:** Remove um recurso específico.
  - **HEAD:** Similar ao GET, mas solicita apenas os cabeçalhos da resposta.
  - **OPTIONS:** Retorna os métodos HTTP suportados pelo servidor para um URL específico.

- As respostas HTTP são acompanhadas por códigos de status, que indicam o resultado da requisição (FIELDING *et al.*, 2014):
  - **1xx:** Informacional.
  - **2xx:** Sucesso (ex: 200 OK).
  - **3xx:** Redirecionamento.
  - **4xx:** Erro do cliente (ex: 404 Not Found).
  - **5xx:** Erro do servidor.

Como protocolo de comunicação entre clientes e servidores, seu desempenho impacta diretamente na velocidade e responsividade de aplicações web (FIELDING *et al.*, 1999).

Fatores relacionados ao HTTP que impactam o desempenho de aplicações incluem a latência de rede, que é o tempo necessário para uma requisição HTTP viajar do cliente ao servidor e retornar, influenciando diretamente o tempo de resposta percebido pelo

usuário (WANG *et al.*, 2013). Para mitigar esses impactos, estratégias como o uso de algoritmos de compressão, como gzip, podem reduzir significativamente o tamanho das transferências de conteúdo HTTP, acelerando os tempos de carregamento (GRIGORIK, 2013). Além disso, mecanismos de *cache* HTTP desempenham um papel crucial ao permitir que recursos sejam armazenados localmente ou em servidores intermediários, o que alivia a carga do servidor de origem e melhora os tempos de resposta (FIELDING; RESCHKE, 2014). Nesse contexto, o uso de *cache* é amplamente adotado como uma prática essencial para otimizar o desempenho de aplicações web, contribuindo para tempos de carregamento mais rápidos, maior eficiência nas respostas e maior disponibilidade dos sistemas (SHIVAKUMAR; SURES, 2018).

3.6 CACHE NA WEB

O *cache* se baseia em dois princípios fundamentais: localidade temporal, que descreve a tendência de acessar dados recentemente utilizados, e localidade espacial, que indica a tendência de acessar dados próximos aos que foram recentemente acessados (HENNESSY; PATTERSON, 2011).

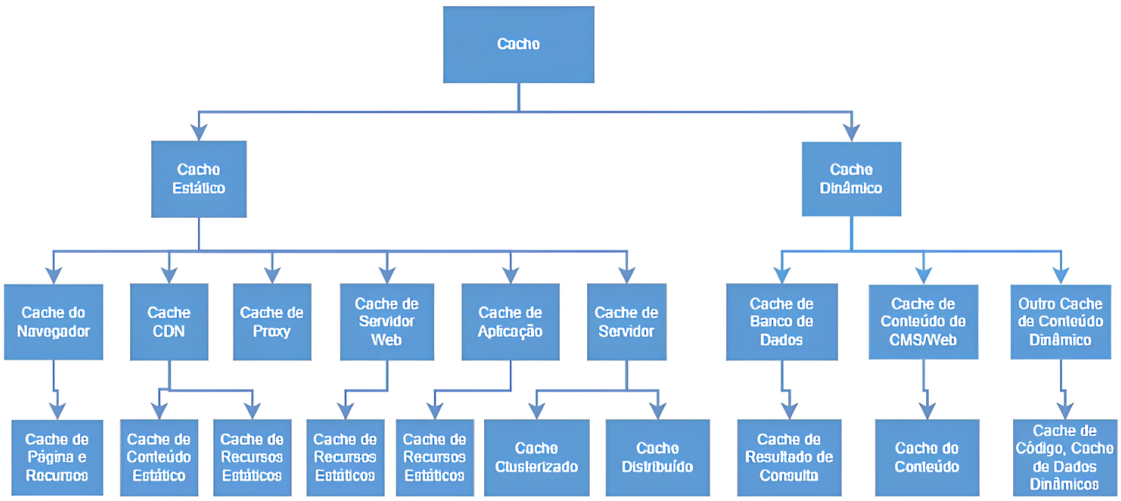
Podlipnig e Böszörmenyi (2003) observaram que os usuários frequentemente retornam às mesmas páginas da web, demonstrando localidade temporal, e acessam conteúdos relacionados em sequência, refletindo localidade espacial (PODLIPNIG; BÖSZÖRMENYI, 2003).

Na *web*, o *cache* armazena objetos usados com frequência próximos ao cliente, seja no navegador, proxy ou servidor. Isso evita que cada solicitação viaje até o servidor de origem, resultando em menor uso de largura de banda, carga reduzida no servidor e, principalmente, menor latência (SUBRAYA, 2006). Davison (2001) descreve o *cache* como "uma coleção de mecanismos de armazenamento em vários pontos da web, projetados para armazenar respostas HTTP frequentemente acessadas por períodos variáveis, servindo-às rapidamente a solicitações subsequentes"(DAVISON, 2001).

Shivakumar e Sures (2018) apontam que as técnicas de *cache* podem ser divididas em estáticas, que armazenam conteúdos imutáveis, e dinâmicas, que gerenciam dados que variam conforme o contexto (SHIVAKUMAR; SURES, 2018). Souders (2007) reforça que o *cache* estático melhora o desempenho ao armazenar recursos de acesso frequente sem a necessidade de processamento contínuo pelo servidor (SOUDERS, 2007). O *cache* dinâmico é responsável por gerenciar conteúdos que podem variar com base em múltiplos

fatores, como contexto de usuário, atributos de usuário, tipos de dispositivos, estado da aplicação e resultados de consultas ao banco de dados (SHIVAKUMAR; SURES, 2018). A figura 5 abaixo representa a taxonomia dos tipos de *cache* utilizados em aplicações web.

Figura 5 – Taxonomia dos tipos de *cache*.



Fonte: (Adaptado de SHIVAKUMAR e SURES, 2018).

No lado estático, a nível do navegador, o *cache* armazena recursos estáticos no lado do cliente, eliminando a necessidade de requisições repetidas ao servidor (MOLYNEAUX, 2013). O *cache* no nível de proxy atua como intermediário entre o cliente e o servidor, armazenando respostas para que possam ser servidas mais rapidamente em solicitações subsequentes (SOUDERS, 2007). No nível do servidor de aplicação, o *cache* pode armazenar resultados de consultas ao banco de dados, objetos de negócio e outros dados frequentemente acessados (SHIVAKUMAR e SURES, 2018).

Na arquitetura de microsserviços, as técnicas de *cache* são empregadas na indústria de software web, cada uma com características e casos de uso específicos (PODLIPNIG; BOSZORMENYI, 2003). O *cache* distribuído é uma técnica onde o *cache* é compartilhado entre vários servidores ou nós em um cluster, permitindo que os dados em *cache* estejam acessíveis em toda a infraestrutura, aumentando a escalabilidade e a disponibilidade do sistema (SHIVAKUMAR e SURES, 2018). Soluções populares de *cache* distribuído aplicadas a nível de servidor incluem Redis[3] e Memcached[4], conhecidos pelo suporte a estruturas de dados avançadas e alta performance (CARLSON, 2013; FITZPATRICK, 2004).

Ainda no contexto dos microsserviços, o Redis reduz a carga (e custo) no banco

[3]Redis: https://redis.io/.
[4]Memcached: https://memcached.org/

de dados, principalmente, em serviços *web* como na Amazon (AWS)[5] em que cada leitura que você faz é um evento cobrável (CARNELL, 2017). O Memcached também é um sistema de *cache* distribuído de alto desempenho. É comumente utilizado para acelerar aplicações *web* dinâmicas, aliviando a carga do banco de dados. O Memcached é utilizado no LiveJournal, Slashdot, Wikipedia e outros sites de alto tráfego (FITZPATRICK, 2004).

De forma geral, pode-se aplicar técnicas de *cache* em diversos pontos entre um cliente e um servidor. Os *caches* podem estar presentes no software do cliente (*cache* do navegador), dentro da rede (*cache* de rede) ou no servidor da aplicação (API) (CROVELLA, *et al.*, 1998). O resultado final de aplicar uma técnica de *cache* é sempre minimizar o tráfego de dados entre cliente e servidor, de forma a reduzir problemas de latência e (espera-se) melhorar a experiência do usuário final (MOLYNEAUX, 2013).

Nos tópicos a seguir, detalharemos duas técnicas de *cache* distribuído a nível de servidor de uma aplicação, Redis e Memcached, que serão utilizadas neste trabalho para realizarmos testes de carga e analisarmos o desempenho de cada ferramenta.

3.7 CACHE DISTRIBUÍDO REDIS

O Redis (*Remote Dictionary Server*) é conhecido como um banco de dados do tipo NoSQL (N*ot only* SQL), baseado no armazenamento de dados no modelo chave-valor (CARLSON, 2013). O Redis opera sem tabelas e utiliza uma estrutura não relacional que armazena diversos tipos de dados, como *Strings*, *Hashes*, *Lists*, *Sets*, *Sorted Sets*, *Bitmaps* e *HyperLogLogs* (DA SILVA e TAVARES, 2015).

O Redis armazena todos os dados na memória RAM, portanto, as operações de leitura e gravação são muito rápidas (DA SILVA e TAVARES, 2015). Essa característica, combinada com a possibilidade de persistir dados no disco por meio de *snapshotting* ou usando o recurso de *Append-Only File* (AOF), que registra cada operação de escrita em um *log*, permite que o Redis seja utilizado tanto como banco de dados primário quanto como um sistema de armazenamento de dados auxiliar (DA SILVA e TAVARES, 2015; REDIS, 2024).

Na figura 6 abaixo, podemos compreender como o Redis se posiciona em meio aos softwares de banco de dados e servidores de cache disponíveis.

O tipo de dado mais simples no Redis são as *strings*, que podem armazenar

[5]Amazon AWS: https://aws.amazon.com

Figura 6 – Características e funcionalidades de alguns bancos de dados e servidores de cache

Nome	Tipo	Opções de Armazenamento de dados	Tipos de consulta	Caracterísiticas adicionais
Redis	Banco de dados não relacional na memória	Strings, listas, conjuntos, hashes, conjuntos ordenados	Comandos para cada tipo de dados para padrões de acesso comuns, com suporte a operações em massa e transações	Publicação/Assinatura, replicação mestre/escravo, persistência de disco, scripts
Memached	Cache de chave-valor na memória	Mapeamento de chaves para valores	Comandos para criar, ler, atualizar, deletar, e outros	Servidor multithread para melhor desempenho
MySQL	Cache de chave-valor na memória	Bancos de dados de tabelas de linhas, visões sobre tabelas	SELECT, INSERT, UPDATE, DELETE, funções, procedures armazenadas	ACID com InnoDB, replicação mestre/escravo e mestre/mestre
PostgreSQL	Banco de dados relacional	Bancos de dados de tabelas de linhas, visões sobre tabelas, extensões de terceiros e espaciais, tipos customizáveis	SELECT, INSERT, UPDATE, DELETE, funções integradas, procedures armazenadas, tipos customizáveis	ACID, replicação mestre/escravo, replicação multi-mestre (com suporte a terceiros)
MongoDB	Banco de dados não relacional em disco	Bancos de dados de tabelas de documentos BSON sem esquema	Comandos para criar, ler, atualizar, deletar, consultas condicionais e outros	Suporta operações map-reduce, replicação mestre/escravo, shard, índices espaciais

Fonte: (Adaptado de CARLSON, 2018).

diversos tipos de informações, como texto (XML, JSON e HTML), inteiros, *floats* e dados binários, como vídeos, imagens e arquivos de áudio.

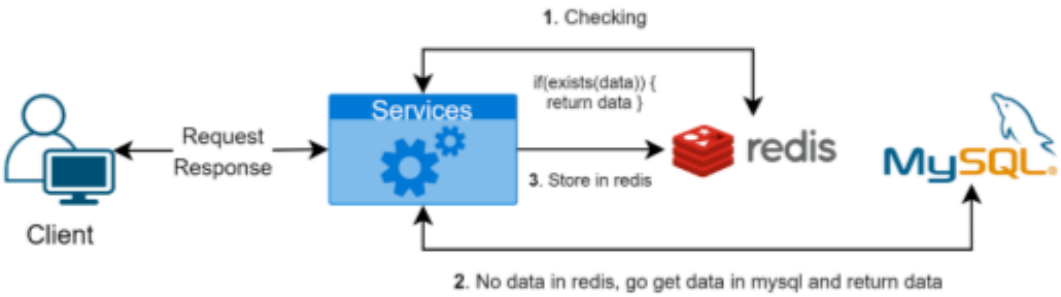
As *strings* são usadas no *cache* de dados, permitindo o armazenamento em *cache* de dados textuais ou binários, como páginas HTML, respostas de API, imagens e vídeos, utilizando comandos como SET, GET, MSET e MGET (DA SILVA E TAVARES, 2015). *Strings* também podem ser utilizadas para implementar contadores, por meio de comandos como INCR e INCRBY, adequados para monitorar visualizações de páginas, vídeos e curtidas (DA SILVA E TAVARES, 2015). É importante destacar que o tamanho máximo permitido para um valor armazenado no Redis é de 512MB (MACEDO E OLIVEIRA, 2011; DA SILVA E TAVARES, 2015).

Em aplicações de microsserviços, o Redis é utilizado como *cache* distribuído e

armazenamento de sessão compartilhado (RICHARDSON, 2018). Nessa arquitetura, o Redis pode otimizar o desempenho ao implementar o *caching* de dados acessados globalmente no nível de uma API *Gateway*. Isso inclui dados de sessão, como IDs de usuários e preferências, e dados de autenticação, como *tokens* e permissões. Ao armazenar em *cache* esses dados, eles ficam disponíveis para todos os serviços, reduzindo a latência da aplicação sem comprometer a integridade dos contextos de negócios de cada microsserviço (REDIS, 2024).

No estudo de Niswar *et al.* (2024) sobre a performance de protocolos de comunicação em microsserviços, o Redis foi utilizado em conjunto com o MySQL, no qual os dados foram inicialmente armazenados no MySQL. Em seguida, os dados foram exportados para o Redis. Observa-se na figura 7 abaixo o funcionamento da aplicação. Durante a recuperação de dados, o sistema priorizou o acesso ao Redis, permitindo uma recuperação mais eficiente. Caso os dados não estivessem disponíveis no Redis, o sistema os buscava no MySQL e os armazenava em cache no Redis, otimizando assim o desempenho das operações de leitura.

Figura 7 – Cache no Gateway da API



Fonte: (Adaptado de NISWAR *et al.*, 2023).

3.8 CACHE DISTRIBUÍDO MEMCACHED

O Memcached é um sistema de *cache* distribuído com foco no desempenho. O Memcached é muitotilizado para acelerar aplicações web dinâmicas, aliviando a carga no banco de dados (FITZPATRICK, 2004). O Memcached é utilizado em muitos sites de alto tráfego, como Wikipedia, Slashdot e Craigslist (FITZPATRICK, 2004). Assim como o Redis, o Memcached armazena os dados em memória RAM, permitindo um acesso extremamente rápido aos dados (NISHTALA e*et al.*, 2013).

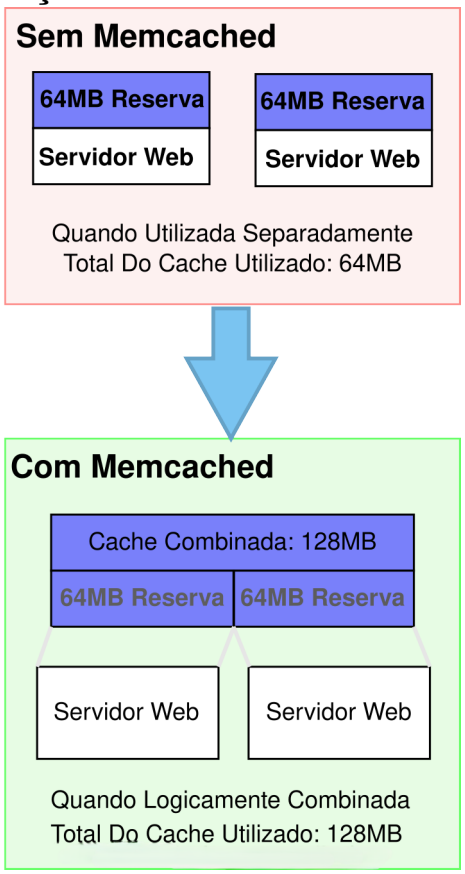
O Memcached é focado no armazenamento de pares chave-valor, e não oferece suporte a estruturas de dados mais complexas (CARLSON, 2013). Por possuir uma API simples, o Memcached é pode ser fácil de integrar em aplicações existentes e requer poucos recursos computacionais para operar (FITZPATRICK, 2004). Isso permite que o Memcached seja altamente escalável, distribuindo os dados entre vários servidores para lidar com grandes volumes de tráfego (NISHTALA *et al.*, 2013). Quando um servidor web solicita dados, primeiramente ele faz a solicitação ao Memcached fornecendo uma chave na forma de uma *string*. Se o item correspondente a aquela chave não estiver em cache, o servidor *web* recupera os dados do banco de dados ou de outro serviço *backend* e preenche o cache com o par chave-valor (NISHTALA *et al.*, 2013).

A Figura 8 apresenta uma comparação entre dois cenários de uso de *cache* em servidores web: um sem Memcached e outro com Memcached. No primeiro cenário, cada servidor web possui 32MB de memória de cache independente, totalizando 64MB utilizáveis por servidor. No segundo cenário, a memória de *cache* dos servidores é combinada logicamente, permitindo que todos os servidores acessem um pool de memória compartilhado de 128MB, resultando em uma utilização de *cache* mais eficiente e maior capacidade total. Isso demonstra como o Memcached pode otimizar o uso da memória e melhorar o desempenho da aplicação.

Por se concentrar apenas no armazenamento de pares chave-valor, diferentemente do Redis, o Memcached não oferece recursos avançados, como pub/sub, transações e persistência de dados (CARLSON, 2013). O Memcached também não possui mecanismos de replicação nativos, o que pode afetar a disponibilidade do sistema em caso de falhas de servidor (NISHALA *et al.*, 2013).

O Memcached utiliza um sistema de gerenciamento de memória chamado*Slab Allocation*, que visa otimizar o armazenamento de objetos de diferentes tamanhos. A memória é dividida em "slabs"de tamanho fixo, que por sua vez são divididos em "chunks". Os "chunks"são agrupados em "classes", cada uma com um tamanho específico, para armazenar objetos de tamanhos semelhantes (CARRA e MICHIARDI, 2019). No entanto, a alocação estática de slabs para as classes pode causar um problema conhecido como "calcificação". Esse problema surge quando a distribuição de tamanhos dos objetos solicitados muda ao longo do tempo. Por exemplo, se objetos maiores se tornam mais populares, as classes com chunks menores podem ficar com excesso de memória alocada, enquanto classes com chunks maiores podem sofrer com falta de espaço, resultando em uma queda na taxa de acertos do cache e um consequente aumento na pressão sobre os

Figura 8 – Comparação de Uso de Cache com e sem Memcached



Fonte: (Adaptado de MEMCACHED, 2024)

servidores de back-end (CARRA e MICHIARDI, 2019).

Apesar dessas limitações, o Memcached continua sendo uma escolha popular para cache distribuído por causa da sua simplicidade, desempenho e escalabilidade. Muitas empresas e projetos de código aberto utilizam o Memcached em conjunto com outros sistemas de armazenamento, como bancos de dados relacionais ou NoSQL para melhorar o desempenho de suas aplicações (FITZPATRICK, 2004).

As características (*features*) do Redis e Memcached, como suporte a diferentes estruturas de dados, podem ser modeladas utilizando modelagem de *features*. Deste modo, é permitido capturar variações e configurações desses sistemas de *cache*, facilitando a tomada de decisões arquiteturais em termos de desempenho. No próximo tópico, discutiremos a modelagem de *features* e como ela pode ser aplicada para representar essas características em sistemas de software web.

3.9 FEATURES E MODELAGEM DE FEATURES

De acordo com Kang *et al.* (1990), uma *feature* é definida como "um aspecto proeminente ou distintivo visível ao usuário, uma qualidade ou característica de um sistema de software ou de sistemas". Posteriormente, Kang *et al.* (1998) classificam *features* como "abstrações funcionais distintamente identificáveis que devem ser implementadas, testadas, entregues e mantidas". Em uma definição complementar, Bosch, Capilla e Hilliard (2015) e Bosch e Lee (2010) descrevem uma *feature* como "uma unidade lógica de comportamento especificado por um conjunto de requisitos funcionais e não funcionais". Dessa forma, features representam características funcionais e não funcionais abstraídas em diferentes fases do desenvolvimento de software (GERALDI, 2022). Apel *et al.* (2013) reforçam que uma *feature* é "uma característica ou comportamento visível ao usuário final de um sistema de software".

Em termos gerais, features são entendidas como funcionalidades ou preocupações de um sistema (BERGER *et al.*, 2020; KRÜGER *et al.*, 2018), utilizadas para especificar, gerenciar e comunicar propriedades funcionais e não funcionais de um sistema de software. Elas auxiliam os desenvolvedores a compreender e adaptar esses sistemas (KRÜGER *et al.*, 2019).

Decorrente da definição de *features*, a modelagem de *features* é uma técnica utilizada para capturar e organizar as funcionalidades de um sistema de maneira hierárquica. Kang e Lee (2002) descrevem a modelagem de *features* como a “atividade de identificar externamente *features* visíveis de produtos em um domínio e organizá-las dentro de um modelo denominado feature model”. Este conceito foi introduzido por Kang *et al.* (1990) no estudo *Feature-Oriented Domain Analysis* (FODA), que captura informações sobre *features* comuns e variantes da linha de produtos de software em diferentes níveis de abstração (BENAVIDES; SEGURA; RUIZ-CORTES, 2010).

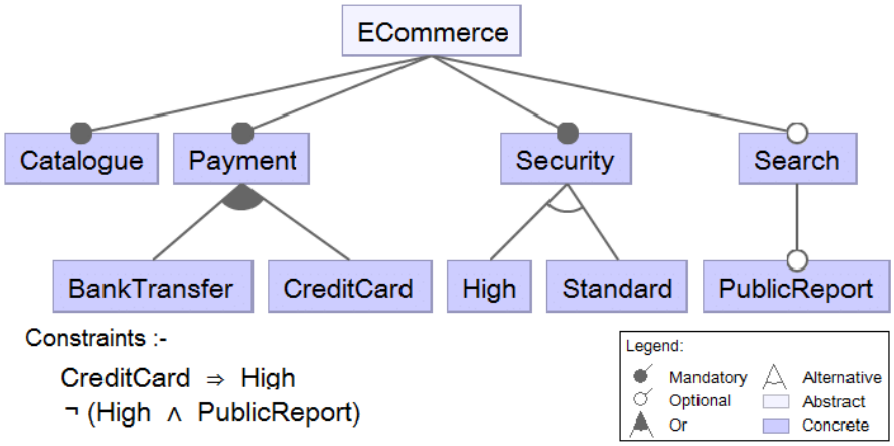
Diagramas de *features* são representações gráficas que mostram a hierarquia de *features* e seus relacionamentos (APEL *et al.*, 2013). Um diagrama de *features* é geralmente representado por uma árvore, onde a raiz representa um conceito central (por exemplo, sistemas de software) e as folhas representam as *features* (CZARNEKI; HELSEN; EISENECKER, 2004).

Analisando as características do *feature model* apresentado na figura 9 abaixo do sistema de *e-commerce* (SAHID *et al.*, 2016), podemos destacar o seguinte:

- **Escopo do modelo**: O modelo representa os principais componentes e recursos de um sistema de e-commerce, incluindo Catálogo, Pagamento e Segurança.
- **Hierarquia e composição:** O modelo possui uma estrutura hierárquica, em que os recursos principais são decompostos em sub-recursos. Por exemplo, o recurso de Pagamento é composto por TransferênciaBancária e CartãoDeCredito.
- **Tipos de relacionamentos:**
  - **Obrigatório (Mandatory)**: Alguns recursos, como Catálogo, Pagamento, Segurança, são considerados obrigatórios, pois devem estar presentes em qualquer implementação do sistema de e-commerce (BENAVIDES; SEGURA; RUIZ-CORTES, 2010; BRITO, 2013 e SAHID *et al.*, 2016).
  - **Opcional (Optional):** Alguns recursos, como os diferentes tipos de segurança (Alta, Padrão, RelatorioPublico), são opcionais e podem ser selecionados de acordo com as necessidades do sistema (BENAVIDES; SEGURA; RUIZ-CORTES, 2010; BRITO, 2013 e SAHID *et al.*, 2016).
  - **Alternativo (Alternative):** O modelo apresenta uma relação alternativa entre os tipos de segurança, indicando que apenas um deles pode ser escolhido por vez (BENAVIDES; SEGURA; RUIZ-CORTES, 2010; BRITO, 2013 e SAHID *et al.*, 2016).
- **Restrições:**
  - **Requerido (Requires):** O modelo indica que, se o recurso CartãoDeCredito for selecionado, o recurso de Alta segurança também deve ser incluído (BENAVIDES; SEGURA; RUIZ-CORTES, 2010; BRITO, 2013 e SAHID *et al.*, 2016).
  - **Excludente (Excludes):** Existe uma restrição de que os recursos Alta segurança e RelatorioPublico são mutuamente exclusivos, ou seja, não podem ser selecionados juntos (BENAVIDES; SEGURA; RUIZ-CORTES, 2010; BRITO, 2013 e SAHID *et al.*, 2016).

Além da modelagem de *features* para representar as características do Redis e Memcached nesta pesquisa, Krishnamurthy e Wills (2000) recomendam a utilização de métricas por meio de testes de desempenho para avaliar e comparar a eficácia de diferentes

Figura 9 – Feature Model de E-commerce



Fonte: (Adaptado de SAHID *et al.*, 2016)

tecnologias e técnicas de *cache* na web. Os testes de desempenho e métricas relacionadas ao *cache* são apresentados(as) nas próximas seções.

3.10 TESTE DE DESEMPENHO

Segundo Meier *et al.* (2007), "teste de desempenho é o processo de determinar a velocidade, a escalabilidade e a estabilidade características de um sistema ou aplicativo sob uma determinada carga de trabalho". Utiliza-se testes de desempenho para garantir que os sistemas de software possam lidar adequadamente com cargas de trabalho esperadas, proporcionando uma experiência positiva aos usuários (JAIN, 1991). Esses testes podem ser classificados em diferentes categorias, dentre as quais destacam-se: teste de carga, teste de estresse, teste de resistência e teste de volume. O teste de carga, por exemplo, simula cargas reais para identificar o limite operacional de um sistema (MEIER, 2013).

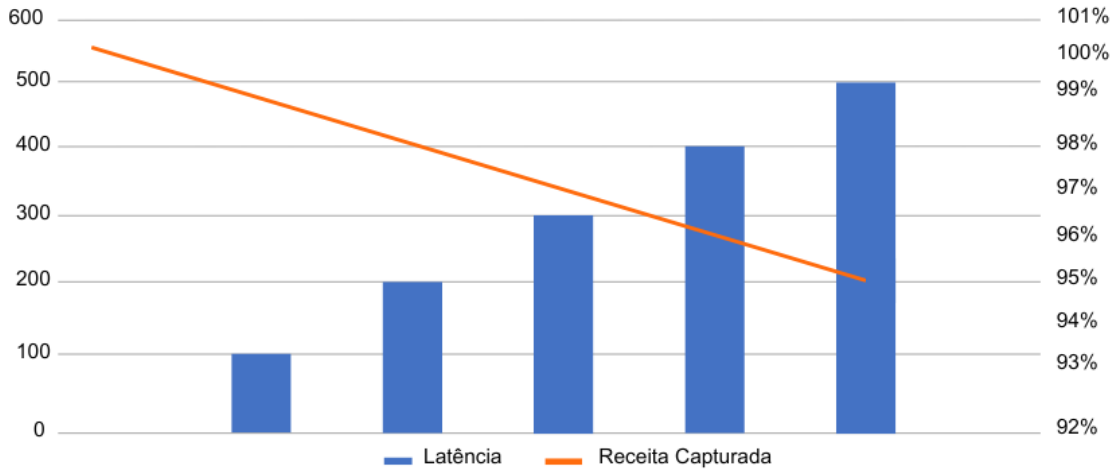
Entre as métricas utilizadas nos testes de desempenho estão o tempo de resposta, *throughput*, utilização de recursos e taxa de erros (MOLYNEAUX, 2009). O tempo de resposta mede o tempo para o sistema processar uma solicitação, enquanto o *throughput* representa o número de solicitações atendidas em um determinado período. A utilização de recursos avalia o consumo de CPU, memória e outros recursos, e a taxa de erros indica a porcentagem de solicitações que resultam em erros (MOLYNEAUX, 2009).

Realizar testes de desempenho é fundamental em operações de alta frequência ou iteração, como sistemas de negociação financeira ou grandes plataformas de *e-commerce*. Os testes visam assegurar que o sistema mantenha sua integridade e eficiência

operacional mesmo sob cargas intensas, minimizando riscos de falhas (MOLYNEAUX, 2009). Operações de alta frequência, como lojas online e aplicativos de mídia social, dependem significativamente de testes de desempenho. Por exemplo, a Amazon realiza testes de desempenho rigorosos para garantir que seu site possa lidar com picos de tráfego durante eventos de vendas como a Black Friday, evitando quedas de serviço e garantindo uma experiência de usuário satisfatória (COLE, 2024).

A Figura 9 ilustra a descoberta da Amazon sobre o impacto do desempenho do site nas vendas. A empresa observou que cada atraso de 100 milissegundos no carregamento do site, resulta em uma queda de 1% nas vendas e no engajamento dos clientes, afetando diretamente a receita. Esse atraso envolve múltiplos aspectos de desempenho, incluindo transmissão de rede, processamento de servidor e eficiência de banco de dados. Esses são os principais focos das equipes de engenharia de software durante a realização de testes de carga e desempenho para garantir a eficiência e a rapidez do site (COLE, 2024).

Figura 10 – Latência X Receita Capturada



Fonte: (Adaptado de TRICENTIS, 2024)

Por fim, coletar métricas de desempenho de aplicações e recursos do sistema, antes de enviá-lo à produção, ajuda a visualizar problemas de desempenho precocemente. Essa abordagem fornece um feedback valioso aos desenvolvedores, ajudando-os a tomar decisões chave e adotar estratégias (ERINLE, 2017). Por meio dessas métricas, é possível estabelecer uma base para conclusões lógicas respaldadas por evidências numéricas, facilitando a avaliação da estabilidade e disponibilidade de um sistema.

É pertinente destacar o teste de carga como um tópico de estudo subsequente. Esse tipo específico de teste de desempenho é fundamental para entender como o sistema

irá se comportar sob condições de carga esperadas em ambiente de produção, fornecendo insights valiosos para a tomada de decisões de otimização de desempenho (PEZZE, 2007).

3.11 TESTE DE CARGA

O teste de carga é um tipo de teste de desempenho que visa avaliar a resposta do sistema sob uma carga específica, utilizando dados ou transações previamente definidas (WAZLAWICK, 2019). Os objetivos principais dos testes de carga são: (i) confirmar se o sistema cumpre os requisitos de desempenho estabelecidos; e (ii) identificar possíveis gargalos de desempenho que necessitem de otimização.

Utiliza-se o teste de carga para analisar a capacidade do sistema em gerenciar múltiplas atividades, realizadas simultaneamente por diferentes usuários (JIANG e HASSAN, 2015). O termo carga refere-se ao número de usuários simultâneos ou ao tráfego no sistema (MENASCÉ e ALMEIDA, 2002; JIANG e HASSAN). Uma carga pesada refere-se a testes que submetem o sistema a condições extremas além de sua capacidade normal, gerando um grande número de solicitações simultâneas para identificar os limites do sistema (MOLYNEAUX, 2014; JAIN, 1991). Ao realizar testes de carga, existem alguns conceitos que devem ser entendidos:

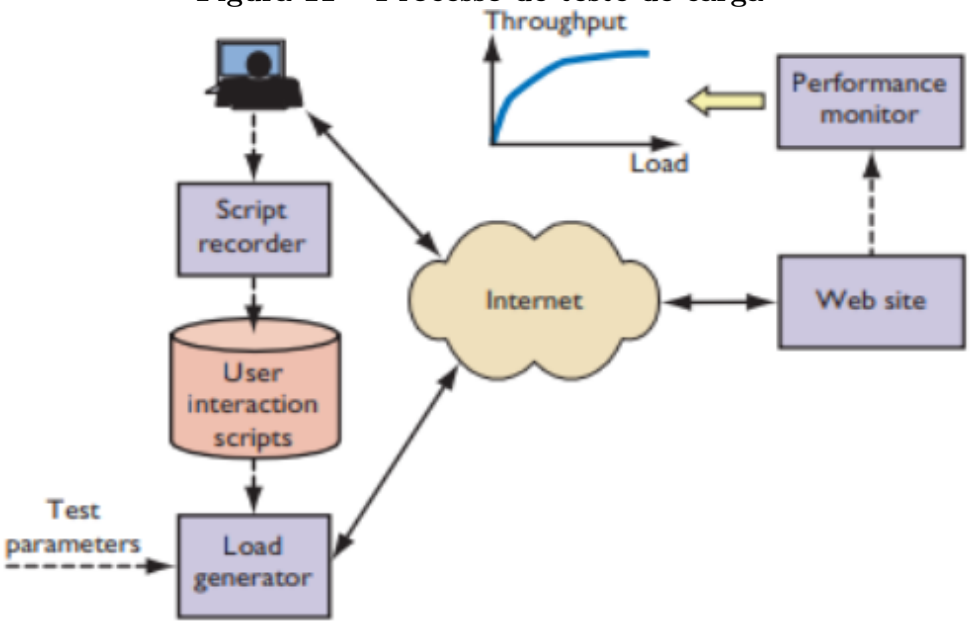
- **Usuários virtuais (VU's)**: São simulações de usuários reais que interagem com o sistema durante o teste de carga, permitindo uma representação precisa do comportamento do usuário (MOLYNEAUX, 2014; MEIER *et al.*, 2007).
- **Tempo de Subida (*Ramp-up Time*)**: É o período durante o qual a carga no sistema é gradualmente aumentada até atingir o nível desejado (MOLYNEAUX, 2014).
- **Carga sustentada (*Steady-state load*)**: O sistema é submetido a uma carga sustentada, que representa o uso normal do software (MOLYNEAUX, 2014; MEIER *et al.*, 2007).
- **Pico de carga (*Peak load*)**: É o período durante o qual o sistema é submetido a uma carga máxima, representando momentos de uso intenso (JAIN, 1991; MOLYNEAUX, 2014).

O teste de carga pode ser realizado por meio de um gerador de carga. Este gerador simula o comportamento de um navegador web enviando várias solicitações para

um site da *web*, emulando milhares de usuários simultâneos para testar o desempenho. Cada navegador é considerado um usuário virtual, um conceito-chave neste tipo de teste (MENASCÉ, 2003).

A Figura 10 ilustra o processo do teste de carga por meio de um gravador de scripts que cria scripts de interação do usuário com base em solicitações reais. O gerador de carga então envia uma carga realista, com base nos scripts e parâmetros de teste, para o site da *web*.

Figura 11 – Processo de teste de carga



Fonte: (Adaptado de MENASCÉ, 2003)

O teste de carga estático avalia o desempenho de um sistema sem alterar as condições de teste durante a execução. O teste é realizado pela aplicação de uma carga constante ao longo do tempo, o que permite observar como o sistema se comporta sob condições de uso previsíveis. Menascé e Almeida (2002) destacam que este tipo de teste é útil para identificar o comportamento do sistema sob um nível constante de demanda.

O teste de carga dinâmica adapta a carga sobre o sistema durante a execução do teste. Este método é utilizado para simular picos de carga e períodos de baixa demanda, proporcionando uma visão mais abrangente do desempenho do sistema em condições variáveis. De acordo com Jain (1991) e Meier *et al.* (2007), o teste de carga dinâmica facilita a compreensão de como o sistema responde a aumentos repentinos no número de usuários ou transações.

3.12 GATLING

De acordo com a documentação do Gatling (GATLING[6], 2023), a ferramenta é descrita como um "instrumento de teste de carga e desempenho para aplicações web modernas". Ela foi criada com foco em simplicidade, alto desempenho e facilidade de uso. Segundo a documentação oficial, algumas das principais características do Gatling incluem (GATLING, 2023):

- **Linguagem Scala:** O Gatling utiliza a linguagem de programação Scala, permitindo uma abordagem orientada a objetos e funcional para a criação de cenários de teste.
- **Arquitetura Assíncrona:** A arquitetura não bloqueante do Gatling, baseada em atores, possibilita a simulação de um alto volume de usuários com baixo consumo de recursos.
- **Suporte a Protocolos:** O Gatling oferece suporte nativo a diversos protocolos, como HTTP, WebSocket, JMS e outros.
- **Geração de Relatórios:** A ferramenta gera relatórios detalhados e visualmente ricos sobre os resultados dos testes.
- **Integração com Frameworks:** O Gatling pode ser integrado a ferramentas de build como Maven e Gradle.

A documentação destaca diversos casos de uso do Gatling, incluindo (GATLING, 2023):

- **Testes de Escalonamento:** Verificar o comportamento de aplicações em cenários de escalabilidade.
- **Testes de API:** Validar a resposta e desempenho de APIs HTTP, WebSocket e outras.
- **Testes de Regressão:** Garantir a manutenção do desempenho ao longo do tempo.

[6]Gatling: https://gatling.io

3.13 MÉTRICAS DE TESTE DE CARGA PARA AVALIAÇÃO DO *CACHE*

Para avaliar o desempenho das tecnologias de *cache* desta pesquisa no capítulo 5, as seguintes métricas são aplicadas:

Tempo de Resposta Médio: No trabalho de Niswar et al. (2023), múltiplos clientes iniciam requisições simultaneamente, variando de 100 a 500 requisições, para avaliar os tempos de resposta. Essa abordagem permite medir como o sistema se comporta quando sujeito a diferentes níveis de atividade de usuários concorrentes. As medições do tempo de resposta foram realizadas tanto para a busca de dados simples quanto de dados aninhados. Cada avaliação foi realizada em dez iterações. Neste caso, o tempo de resposta médio ($aveRT$) pode ser calculado usando a Eq. 1:

$$\text{aveRT} = \frac{1}{n}\sum_{i=1}^{n}(t_{\text{resp}} - t_{\text{req}}) \tag{1}$$

Vazão (*Throughput*): A vazão mede a taxa na qual os dados podem ser lidos ou gravados no *cache*. É geralmente medida em operações por segundo (ops/s) ou megabytes por segundo (MB/s). Uma vazão mais alta indica melhor desempenho, permitindo que o sistema de cache atenda a mais solicitações em um determinado período. Carra *et al.* (2015) avaliam o impacto de diferentes políticas de gerenciamento de memória no Memcached, utilizando a vazão média como uma das principais métricas. Wang *et al.* (2017) usam a vazão como uma medida chave para comparar o desempenho do Memcached com o Cavast. A fórmula da vazão pode ser calculada pela Eq. 2 abaixo:

$$\text{Vazão} = \frac{\text{Número de Operações}}{\text{Tempo}} \tag{2}$$

Taxa de Acertos no Cache (*Cache Hit*): Mede a proporção de objetos/recursos solicitados ao servidor que são atendidos pelo *cache* (CROVELLA *et al.*, 1998; MYERS *et al.*, 2012). Podemos calcular o *cache hit* pela Eq. 3:

$$\text{Taxa de Acerto} = \frac{\text{Número de acertos no } \textit{cache}}{\text{Total de solicitações ao } \textit{cache}} \tag{3}$$

3.14 TRABALHOS RELACIONADOS

Os estudos relacionados apresentam importantes contribuições para a avaliação de sistemas de banco de dados e *caches* distribuídos, fornecendo uma base comparativa para a análise de desempenho em diferentes contextos.

Abubakar *et al.* (2014) exploraram o desempenho de quatro sistemas NoSQL — MongoDB, ElasticSearch, Redis e OrientDB — utilizando o *framework* YCSB em um ambiente com recursos limitados, configurado em um único computador. A pesquisa visava auxiliar profissionais de TI em cenários onde restrições de recursos demandam soluções eficientes em sistemas compactos. Foram aplicados diversos *workloads* para analisar operações como inserção, leitura e atualização. Apesar dos resultados indicarem diferenças significativas no desempenho entre os sistemas, o estudo ressalta limitações ao considerar apenas um ambiente não distribuído, o que pode não refletir condições de larga escala.

Por sua vez, Salhi *et al.* (2018) realizaram uma avaliação detalhada de dois sistemas de cache distribuído amplamente utilizados, Hazelcast e Infinispan, empregando a ferramenta Yardstick para medir o impacto do número de clientes simultâneos e do tamanho dos dados no desempenho. Os resultados mostraram que o Infinispan apresentou desempenho superior na maioria dos cenários, especialmente em operações de recuperação de dados e consultas semelhantes a SQL, com exceção de cenários com dados pequenos, onde o Hazelcast se destacou. O estudo destaca a relevância de métricas dinâmicas no desempenho de *caches* distribuídos, evidenciando que análises baseadas apenas em fatores estáticos podem ser enganosas.

Outro estudo significativo foi conduzido por Zhang *et al.* (2014), que analisaram os sistemas Memcached, Redis e Resilient Distributed Datasets (RDD) do Spark, focando em operações analíticas e de granularidade fina, como SET e GET. A pesquisa revelou que, enquanto Memcached e Redis eram eficientes em operações de objetos, enfrentavam gargalos relacionados à CPU e à E/S. Em contrapartida, o RDD demonstrou desempenho superior em análises, embora penalizado pelo alto custo de inicialização em operações de recuperação aleatória. Os autores propuseram recomendações para o gerenciamento eficiente de dados na memória, como minimizar o movimento de dados, explorar paralelismo e otimizar a E/S.

Além disso, Cooper *et al.* (2010) introduziram o Yahoo! Cloud Serving Benchmark (YCSB), uma estrutura projetada para avaliar sistemas de dados em

nuvem, com *benchmarks* aplicados a Cassandra, HBase, PNUTS, Redis, Memcached e MySQL particionado. Este *framework* tornou-se uma ferramenta essencial para estudos comparativos de desempenho.

Por fim, Atikoglu *et al.* (2012) analisaram os padrões de carga de trabalho do Memcached no Facebook, uma das maiores implantações mundiais. O estudo investigou cinco *pools* de Memcached, observando diferenças nos padrões de leitura/gravação, tamanhos de requisição e taxas de acesso. Descobriram que valores pequenos predominavam no *cache* e que padrões temporais eram influenciados por ciclos de atividade do usuário. Os autores também desenvolveram um modelo analítico para gerar *workloads* sintéticas, baseado em distribuições de lei de potência.

Esses trabalhos enfatizam a importância de considerar fatores como recursos computacionais, volume de dados, padrões de acesso e arquiteturas subjacentes para a análise e otimização de sistemas distribuídos e microsserviços. Suas descobertas são fundamentais para direcionar estratégias em cenários reais de aplicação.

4 FEATURE MODELS: REDIS E MEMCACHED

Conforme apresentado na revisão da literatura, a modelagem de *features* permite organizar e representar as características e funcionalidades de diferentes softwares e ferramentas por meio de feature models. Neste capítulo, compararemos as principais *features* das ferramentas de cache Redis e Memcached. O objetivo é oferecer uma visão estruturada das particularidades de cada ferramenta com o intuito de fornecer um guia prático de adoção para desenvolvedores, tanto no meio acadêmico quanto na indústria.

4.1 FEATURE MODELS

Para facilitar a compreensão e comparação dessas ferramentas, foram desenvolvidos dois feature models utilizando o software FeatureIDE. O objetivo é representar graficamente as principais features do Redis e Memcached.

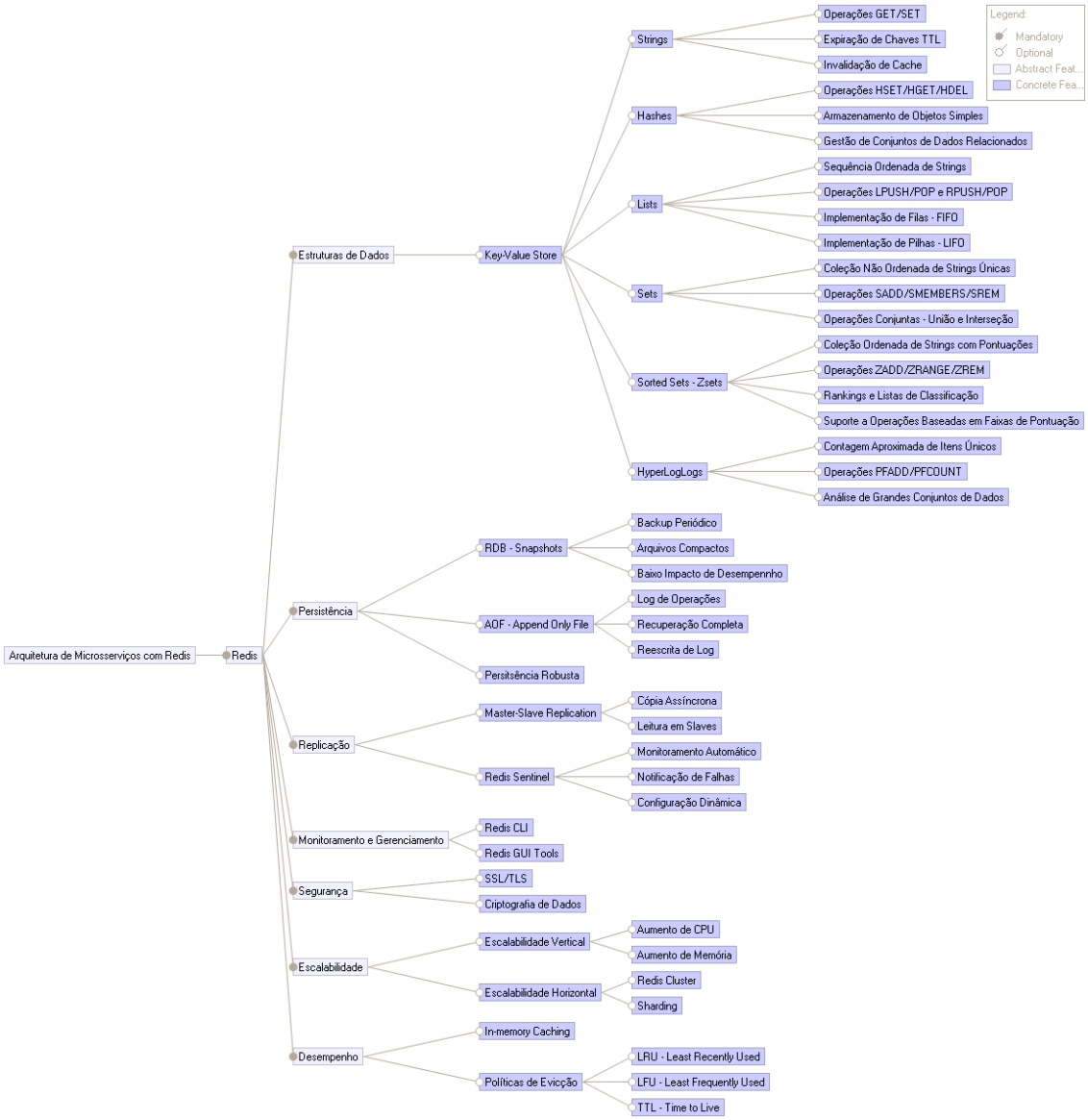
As categorias principais são representadas por abstract *features* (não é passível de implementação, está em tempo de projeto (design-time)), enquanto as funcionalidades específicas de cada categoria são representadas por concrete *features* (passível de implementação em código-fonte). As concrete *features* foram organizadas como child *features* das abstract *features*.

Para a tecnologia Redis, as features relacionadas à persistência de dados foram organizadas como child *features* da feature principal "Persistência". Isso inclui funcionalidades como "RDB - Snapshots"e "AOF - Append Only File". As diferentes estruturas de dados, como “Strings”, “Hashes”, “Lists” e outras, estão organizadas como child *features* da feature principal “Estruturas de Dados”. A escalabilidade, outra feature importante para o desempenho, é refletida em child *features* como "Redis Cluster"e "Sharding", que suportam a distribuição de dados e a escalabilidade horizontal. Adicionalmente, a feature "Política de Invalidação"inclui child *features* como "Tempo de Expiração"e "Evicção LRU", que são fundamentais para o gerenciamento eficiente do cache. Outras abstract *features* importantes, como "Segurança"e "Replicação", também são abordadas com funcionalidades como "Criptografia de Dados"e "Master-Slave Replication".

Para a tecnologia Memcached, as *features* relacionadas à persistência de dados são representadas por child *features* da feature principal "Persistência", que, neste caso, inclui "Sem Persistência Nativa"e "Dados Voláteis". A escalabilidade no Memcached,

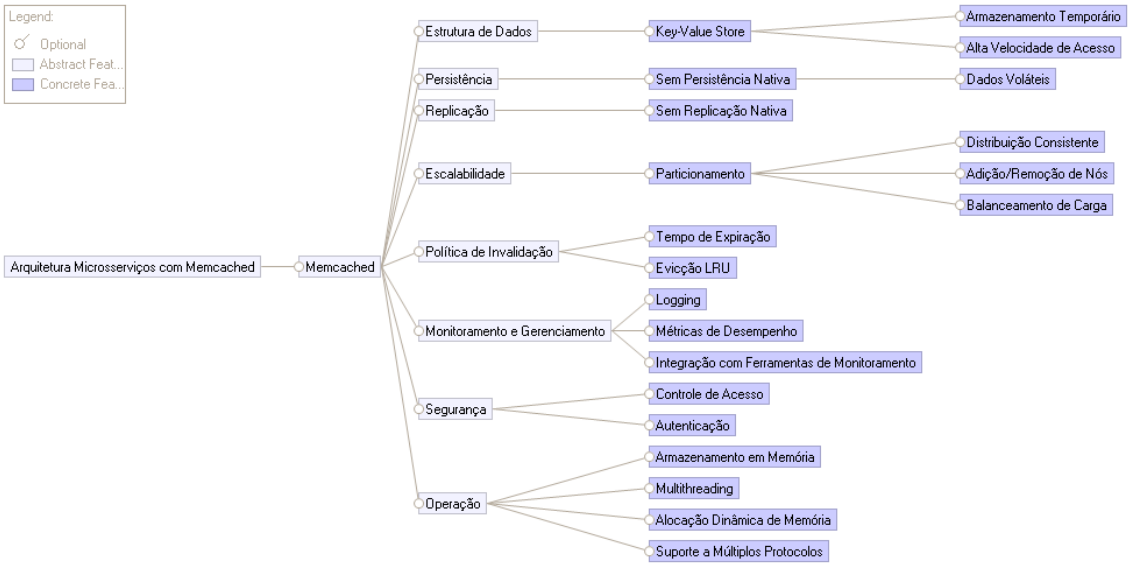
é refletida em child *features* como "Particionamento"e "Distribuição Consistente", que suportam a distribuição de dados e a adição/remoção de nós para balanceamento de carga. A independência de serviços e a flexibilidade operacional são destacadas pela feature "Operação", que inclui child *features* como "Multithreaded"e "Suporte a Múltiplos Processadores". A segurança é tratada através de funcionalidades como "Autenticação"e "Controle de Acesso", organizadas sob a feature "Segurança". Além disso, o monitoramento e o gerenciamento são suportados por *features* como "Loggins"e "Métricas de Desempenho". Os feature models completos podem ser visualizados nas Figuras 11 e 12 à seguir.

Figura 12 – Diagrama de *Features* da Tecnologia de *Cache* Redis



Fonte: (Autoria Própria, 2024).

Figura 13 – Diagrama de *Features* da Tecnologia de *Cache* Memcached



Fonte: (Autoria Própria, 2024).

4.2 CRITÉRIOS PARA SELEÇÃO DAS TECNOLOGIAS DE *CACHE*

Os *feature models* representam o conjunto total das *features* apresentadas de cada ferramenta de cache. No entanto, não foram utilizadas todas as *features* no desenvolvimento dos protótipos desenvolvidos neste trabalho. Considerando o atual cenário tecnológico, onde a indústria de software enfrenta crescentes demandas por alta performance e escalabilidade, o primeiro critério a ser considerado para a escolha das ferramentas foi o RNF de desempenho em operações de cache.

A ferramenta Redis se destaca por armazenar os dados em memória RAM, por isso possui acesso extremamente rápido aos dados, principalmente nas operações GET e SET (leitura e escrita, respectivamente), o que resulta na diminuição da latência associada ao acesso a discos (HD, SSD). Outra característica de desempenho importante a ser considerada no Redis é a utilização do modelo de dados chave-valor (*key-value*), que simplifica as operações de alocação e liberação de memória, e maximiza a utilização da memória (EL ALAMI *et al.*, 2019).

Em testes de benchmark, o Redis demonstrou ser capaz de realizar mais de 100.000 operações SET por segundo e mais de 80.000 operações GET por segundo em um servidor comum (REDIS LABS, 2021). Esta alta taxa de transferência é resultado não apenas do armazenamento em memória, mas também da arquitetura de evento único *(single-threaded)* do Redis.

Por sua vez, o Memcached possui uma arquitetura mais simples, o que permite que que essa ferramenta processe um grande número de operações por segundo com latência muito baixa. Em grandes implementações, como no caso do Facebook, Nishala *et al.* (2013) descreve que o desempenho pode variar dependendo da configuração do hardware e da rede, no entanto, é capaz de lidar com dezenas de milhares de operações por segundo em configurações típicas. Em testes realizados pela equipe do Facebook, um único servidor Memcached foi capaz de processar mais de 200.000 solicitações por segundo (NISHTALA *et al.*, 2013).

No estudo conduzido por Atigloku *et al.* (2012) sobre o Memcached, os autores avaliaram o desempenho do Memcached sob padrões de carga variáveis, aplicando cargas variáveis ao longo do tempo, em diferentes horas do dia e dias da semana. Eles verificaram que o sistema foi capaz de lidar com picos de carga, muitas vezes dobrando ou até triplicando a taxa normal de requisições. Outro detalhe verificado é que a alta taxa de acerto *(hit rate)* pode ser significativamente influenciada pelo tamanho do *pool* de *cache*. Em certos casos, a ampliação do *pool* de *cache* resultou em uma melhora mais substancial na taxa de acertos do que na otimização da localidade das chaves (ATIGLOKU *et al.*, 2012).

O segundo critério é a **adoção e suporte na indústria**. O relatório "DB-Engines Ranking"de 2023, uma referência respeitada na indústria de banco de dados, posiciona o Redis consistentemente na primeira posição entre os sistemas de gerenciamento de banco de dados *key-value*, enquanto o Memcached mantém uma posição constante entre os 10 primeiros. Este ranking considera fatores como popularidade e discussões técnicas, e ressalta a relevância contínua dessas ferramentas no cenário tecnológico atual.

A Stack Overflow Developer Survey[1] de 2023, uma das maiores e mais respeitadas pesquisas na comunidade de desenvolvedores, listou o Redis como uma das tecnologias mais utilizadas, em parte devido ao primeiro critério mencionado, o desempenho. O Gartner[2], uma das principais empresas de pesquisa e consultoria em tecnologia do mundo, em seu relatório de pesquisa “Magic Quadrant for Cloud Database Management Systems” de 2020, posicionou o Redis como umas das principais soluções líderes no segmento de cache em memória, destacando seu alto desempenho como um diferencial competitivo.

Já o Memcached é amplamente utilizado por organizações de alto perfil, como

[1]Stack Overflow Developer Survey: https://survey.stackoverflow.co/2023/
[2]Gartner: https://www.gartner.com.br/pt-br

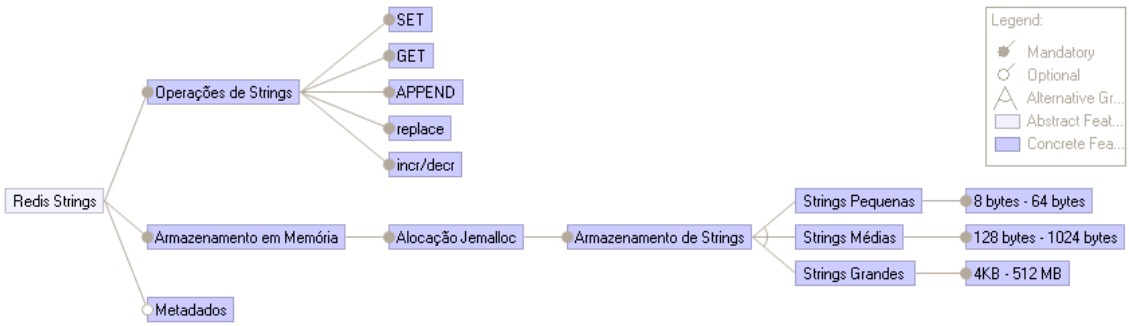
Facebook[3], Twitter[4] e Reddit[5], para otimizar o desempenho de suas aplicações em larga escala. Por exemplo, o Facebook, no artigo "Scaling Memcache at Facebook"de Nishala *et al.*, (2013), detalhou como utiliza o Memcached extensivamente em sua infraestrutura, gerenciando trilhões de itens em cache diariamente. Essa adoção por empresas líderes em tecnologia demonstra a confiabilidade e a eficiência contínuas do Memcached em ambientes de produção de alta demanda, mesmo com o surgimento de novas tecnologias de *cache*.

Para demonstrar a aplicação prática dos *feature models* criados, este trabalho inclui o desenvolvimento de protótipos de um sistema de *cache* utilizando Redis e Memcached. Embora não tenha sido realizada uma avaliação formal de desempenho para estes protótipos, as características mais críticas em sistemas de *cache* distribuídos foram consideradas como os principais fatores de análise.

4.3 CONFIGURAÇÕES DE ESTRUTURA DE DADOS PARA *CACHE*

Com base nos feature models desenvolvidos, foram estabelecidas duas configurações de *cache*, utilizando a ferramenta FeatureIDE como suporte, que podem ser visualizada nas figuras 13 e 14 abaixo. Essas configurações foram aplicadas na implementação de dois protótipos de uma API de *e-commerce*, ambos utilizando a mesma estrutura de dados, especificamente *strings*, e como seu gerenciamento de *strings* funciona, para garantir a consistência e comparabilidade entre as tecnologias de *cache* Redis e Memcached.

Figura 14 – Diagrama de *Features* da Tecnologia de *Cache* Redis com foco em *Strings*



Fonte: (Autoria Própria, 2024).

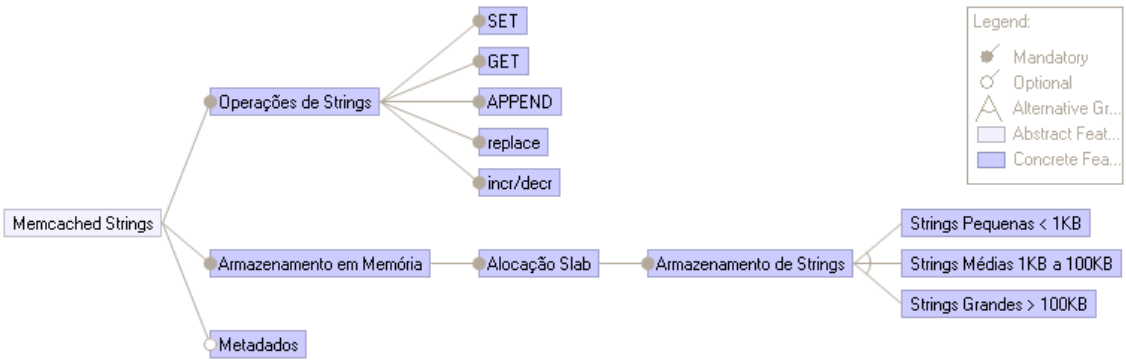
[3]Facebook: https://facebook.com
[4]Twitter: https://twitter.com
[5]Reddit: https://reddit.com

- **Redis Strings (Raiz)**:
  - **Tipo**: Abstrato.
  - **Descrição**: Representa a funcionalidade geral de armazenamento e manipulação de strings no Redis.
- **Armazenamento em Memória**:
  - **Tipo**: Concreto e Obrigatório.
  - **Descrição**: Abrange os mecanismos de alocação e gerenciamento de memória para strings.
  - **Subcaracterísticas**:
    - **Alocação Jemalloc**:
      - **Tipo**: Concreto e Obrigatório.
      - **Descrição**: Gerencia a alocação de memória para otimização de desempenho.
    - **Armazenamento de String**:
      - **Tipo**: Abstrato e Obrigatório.
      - **Descrição**: Classifica o armazenamento de strings em diferentes categorias de tamanhos.
      - **Subcaracterísticas (OR)**:
      - **Strings Pequenas**:
      - **Descrição**: Strings com tamanho entre 8 e 64 bytes.
      - **Strings Médias**:
      - **Descrição**: Strings com tamanho entre 128 bytes e 1024 bytes.
      - **Strings Grandes**:
      - **Descrição**: Strings com tamanho entre 4 KB e 512 MB.
- **Operações de Strings**:
  - **Tipo**: Concreto e Obrigatório.
  - **Descrição**: Inclui as principais operações que podem ser realizadas em strings.
  - **Subcaracterísticas**:
    - **SET**: Define um valor para a chave.

    * **GET**: Recupera o valor associado a uma chave.
    * **append**: Adiciona dados ao final de uma string.
    * **replace**: Substitui o valor atual de uma string.
    * **incr/decr**: Incrementa ou decrementa o valor numérico armazenado.
    * **strlen**: Retorna o comprimento da string.

- **Metadados**:
  - **Tipo**: Concreto e Opcional.
  - **Descrição**: Informações adicionais sobre as strings armazenadas.

Figura 15 – Diagrama de *Features* da Tecnologia de *Cache* Memcached



Fonte: (Autoria Própria, 2024).

- **Memcached Strings**:
  - Característica *abstrata*, representando a funcionalidade geral de armazenamento e manipulação de strings.
  - Organiza o modelo hierárquico, sem ser diretamente implementada.
- **Operações de Strings**:
  - Característica *concreta e obrigatória*.
  - Relacionamento *AND*: engloba as operações disponíveis (`set`, `get`, `append`, `replace`, `incr/decr`), todas coexistindo no sistema.

- **Armazenamento em Memória**:
  - Característica *concreta e obrigatória*.
  - Relacionamento *AND*: composta pelas subcaracterísticas **Alocação Slab** (obrigatória) e **Metadados** (opcional).
- **Alocação Slab**:
  - Característica *concreta e obrigatória*.
  - Representa o uso de blocos de memória fixos para otimizar o armazenamento.
- **Armazenamento de String**:
  - Característica *concreta e obrigatória*.
  - Relacionamento *Alternative (OR)*: classifica as strings em três categorias mutuamente exclusivas:
    - **Strings Pequenas** ($<$ 1 KB).
    - **Strings Médias** (1 KB a 100 KB).
    - **Strings Grandes** ($>$ 100 KB).
- **Metadados**:
  - Característica *opcional e concreta*.
  - Relaciona-se ao gerenciamento de informações auxiliares sobre as strings armazenadas.

5 IMPLEMENTAÇÃO DOS TESTES DE DESEMPENHO DE *CACHE*

5.1 CONSIDERAÇÕES INICIAIS

O desenvolvimento desta pesquisa segue as etapas de planejamento, execução, e análise e interpretação dos resultados (capítulo 2). Essas etapas são essenciais para avaliar o desempenho das tecnologias de cache em microsserviços. Para conduzir a avaliação dessas tecnologias, o método de testes de carga de Meier (2007) foi adaptado neste contexto. Assim, este capítulo possui o foco na implementação prática dos testes, utilizando o método de Meier (2007) para garantir a consistência dos testes nos experimentos.

Na seção 5.1, é apresentada a configuração da infraestrutura necessária para garantir que os serviços estivessem disponíveis e adequadamente configurados para simulações precisas de carga. A seção 5.2 detalha a definição dos casos de teste, elaborados com base no Yahoo Benchmark (YCSB). Na seção 5.3, é explicada a implementação dos testes, que utilizam scripts automatizados para simular múltiplos usuários virtuais, interagindo com as APIs e realizando operações de leitura e escrita. A seção 5.4 descreve a execução dos testes, realizada em um ambiente Windows, com serviços conteinerizados, permitindo a coleta de métricas detalhadas de desempenho por meio da ferramenta Gatling.

Por fim, a seção 5.5 apresenta as limitações do ambiente de teste local, que incluem a utilização de uma máquina com especificações limitadas, a falta de escalabilidade horizontal, e a ausência de uma rede distribuída ou balanceamento de carga adequado.

5.2 DEFINIR E CONFIGURAR AMBIENTE DE TESTES

Para a avaliação, foi planejado executar os testes em uma máquina local por causa da ausência de um ambiente de produção. Esse ambiente pode limitar os recursos disponíveis, mas ainda permitirá uma avaliação controlada do desempenho das tecnologias de cache Redis e Memcached. Na tabela N, pode-se observar a configuração da máquina utilizada para a realização dos testes de carga.

Tabela 1 – Especificações de Hardware e Software

Categoria	Especificações
Hardware	
Processador	AMD Ryzen 5 3500X, 6 núcleos, 3.60 GHz
Memória RAM	16GB
Armazenamento SSD	256GB
Software	
Sistema Operacional	Windows 11 Pro 64 Bit

Fonte: (Autoria própria, 2024).

5.2.1 Estrutura do Ambiente

A configuração do ambiente foi implementada utilizando a versão 3.8 do Docker Compose com uma arquitetura baseada em microsserviços para um sistema de cadastro de produtos de e-commerce. Dois arquivos `docker-compose.yml` foram configurados, cada um com uma tecnologia de cache específica, Redis ou Memcached. Assim, duas *Application Programming Interface* (API's) de microsserviços foram implementadas, mantendo a mesma estrutura para garantir consistência na avaliação de desempenho. Os componentes do ambiente são detalhados a seguir.

- **Docker Compose versão 3.8:** Implementado para criar um ambiente isolado para cada tecnologia, permitindo que Redis e Memcached operem em contêineres independentes.
- **API *Gateway*:** Funciona como o ponto de entrada centralizado do sistema. É responsável por receber e encaminhar as requisições para os microsserviços apropriados, como *auth*, *order* e *product*. O *gateway* foi implementado utilizando Node.js e está configurado para escutar na porta 3001. Todos os serviços são conectados por meio da *network* denominada *ecommerce*, que garante comunicação interna entre os contêineres.
- **Serviço de Produtos (*product*):** Responsável por gerenciar os produtos no *e-commerce*, este serviço se conecta ao MongoDB para persistência de dados e ao Redis ou Memcached para cache, dependendo do ambiente configurado. No primeiro arquivo, o Redis é utilizado como sistema de cache, configurado via as variáveis `REDIS_HOST` e `REDIS_PORT`. No segundo arquivo, Memcached substitui o Redis, sendo configurado via `MEMCACHED_HOST` e `MEMCACHED_PORT`. Esse serviço escuta na porta 3003 e depende do MongoDB.

- **MongoDB:** Banco de dados NoSQL responsável pelo armazenamento persistente de dados, sendo utilizado como banco de dados primário da aplicação. Utiliza a imagem oficial mongo:latest e escuta na porta 27017. O volume mongo-data é mapeado para persistir os dados entre os reinícios dos contêineres.

- **Redis e Memcached:** Redis e Memcached são configurados como sistemas de cache para melhorar o desempenho da aplicação, armazenando informações temporárias, como produtos no presente trabalho. No primeiro arquivo, Redis utiliza a imagem redis:7.4.0-alpine, escutando na porta 6379. No segundo arquivo, Memcached substitui o Redis e utiliza a imagem memcached:1.6-alpine, escutando na porta 11211. Ambos os serviços estão integrados à *network* ecommerce, permitindo comunicação com os demais microsserviços.

5.2.2 Gatling

Dois cenários de teste foram configurados utilizando a ferramenta Gatling, um para cada tecnologia de cache, seguindo a mesma estrutura para manter a consistência na comparação dos resultados. A configuração dos testes envolve:

- **Cenário de Usuários (*Scenario*):** O cenário simula múltiplos usuários simultâneos realizando requisições à API do *e-commerce*, permitindo avaliar como o sistema lida com a carga gerada por acessos concorrentes. Para isso, o Gatling utiliza *threads*, que representam cada usuário realizando operações de leitura no *endpoint* de produtos da API.

- **Requisição HTTP (HTTP *Request*):** No contexto dos testes, as requisições são do tipo **GET** e **WRITE**, direcionadas ao endpoint `/products`, que retorna a lista de produtos e/ou registra novos produtos. O código de teste inclui a verificação do status HTTP da resposta, assegurando que o servidor responda corretamente com o código 200, fundamental para validar o sucesso das requisições sob carga.

- **Gerenciamento de Cabeçalhos HTTP (HTTP *Header Management*):** Os cabeçalhos HTTP são configurados para garantir a correta formatação e envio das requisições. No presente teste, o cabeçalho `Content-Type` foi definido como `application/json`, garantindo que a comunicação com a API esteja alinhada ao formato de dados esperado, no caso deste trabalho, JSON.

- **Métricas de Desempenho e Relatórios:** O Gatling fornece métricas detalhadas sobre o desempenho da API durante os testes, como tempo de resposta, *throughput* e taxa de sucesso das requisições. Após a execução dos testes, a ferramenta gera relatórios visuais que facilitam a análise dos dados e a identificação de potenciais gargalos no desempenho.

5.2.3 Implementação do Código para Redis

O Redis foi configurado para conectar-se ao servidor Redis executando no Docker (porta 6379) e está integrado ao serviço de produtos da API. O código de conexão com Redis é apresentado na figura 16.

Neste trecho de código, o Redis é configurado para conectar-se à instância Redis no Docker. Caso ocorra um erro de conexão, ele é capturado e exibido no console. No `ProductController`, o método getProducts foi implementado para verificar se a lista de produtos já está armazenada no *cache* do Redis antes de buscar os dados diretamente do banco de dados MongoDB. O código é apresentado a seguir na figura 17:

Este código ilustra como o Redis armazena os produtos em *cache*. Se a lista de produtos já estiver armazenada no Redis, ela será recuperada diretamente do *cache*, economizando tempo de leitura. Caso contrário, os produtos serão recuperados do MongoDB, e o *cache* será atualizado.

5.2.4 Implementação do Código para Memcached

Por sua vez, o Memcached foi configurado para conectar-se ao servidor Memcached executando no Docker (porta 11211) e está integrado ao serviço de produtos da API. O código de conexão com Memcached é apresentado na figura 14 à seguir.

Neste trecho de código, o Memcached é configurado para conectar-se à instância Memcached no Docker. A biblioteca memjs foi utilizada para facilitar a comunicação com o servidor Memcached. Caso haja problemas na conexão ou recuperação de dados, os erros são capturados e tratados. No ProductController, o método getProducts foi implementado para verificar se a lista de produtos já está armazenada no cache do Memcached antes de buscar os dados diretamente do banco de dados MongoDB. O código é mostrado abaixo na figura 15:

Figura 16 – Configuração do Cliente Redis

```
// Importa o módulo do Redis para criar um cliente Redis.
const redis = require("redis");

/*Carrega variáveis de ambiente de um arquivo .env para o processo,
útil para configurar URLs, senhas, etc*/
require("dotenv").config();

/*Cria um cliente Redis, especificando a URL do servidor Redis
que está sendo executado na porta 6379.
*/
const client = redis.createClient({
  url: 'redis://redis:6379'
});

/*Define um evento para tratar erros que possam ocorrer na conexão
com o Redis, exibindo a mensagem de erro no console.*/
client.on("error", (err) => {
  console.error("Redis error:", err);
});

/* Define um evento que será disparado quando a conexão com o Redis
for bem-sucedida, exibindo uma mensagem de confirmação.*/
client.on("connect", () => {
  console.log("Connected to Redis");
});

/* Conecta ao servidor Redis de forma assíncrona e captura
qualquer erro que possa ocorrer durante a conexão.*/
client.connect().catch(console.error);

module.exports = client;
// Exporta o cliente Redis para ser utilizado em outras partes da aplicação.
```

Fonte: (Autoria Própria, 2024)

5.3 DEFINIÇÃO DO PLANO DE TESTE

O plano de teste foi desenvolvido com base no Yahoo! Cloud Serving Benchmark (YCSB), uma ferramenta amplamente reconhecida para a avaliação de desempenho de sistemas de armazenamento em cache. O YCSB foi selecionado por oferecer uma metodologia robusta para simular diferentes cargas de trabalho e, assim, avaliar o comportamento de sistemas de armazenamento em diversos cenários.

Figura 17 – Função para Recuperação de Produtos com Cache Redis

```
// Função assíncrona para obter produtos
async getProducts(req, res) {

  // Tenta recuperar os produtos do cache Redis usando a chave "products"
  const cachedProducts = await redisClient.get("products");

  // Se os produtos estiverem no cache, retorna-os como resposta em JSON
  if (cachedProducts) {
    console.log("Produtos recuperados do cache");
    return res.json(JSON.parse(cachedProducts));
  }

  // Caso não estejam no cache, busca os produtos do banco de dados
  const products = await Product.find();

  // Armazena os produtos no cache Redis e define uma expiração de 1 hora
  await redisClient.set("products", JSON.stringify(products), { EX: 3600 });

  // Retorna os produtos do banco de dados como resposta
  console.log("Produtos recuperados do banco de dados");
  res.json(products);
}
```

Fonte: (Autoria Própria, 2024)

Figura 18 – Configuração do Cliente Memcached

```
// Importa a biblioteca Memcached para criar um cliente Memcached
const Memcached = require('memjs');

// Configura o cliente Memcached conectando-se ao servidor na porta 11211
const client = Memcached.Client.create('memcached:11211');

// Exporta o cliente Memcached para ser utilizado em outras partes da aplicação
module.exports = client;
```

Fonte: (Autoria Própria, 2024)

5.3.1 Plano de Teste Baseado no YCSB

O plano de teste foi estruturado com base nas cargas de trabalho definidas pelo Yahoo Cloud Serving Benchmark (YCSB) para avaliar a eficiência de tecnologias de cache, Redis e Memcached, na API de *e-commerce*. Esse plano contempla um conjunto de etapas e parâmetros que simula um cenário realista de operações intensivas de leitura. Para isso, foram adotadas as configurações do *workload* C do YCSB, que é configurado para 100%

Figura 19 – Função para Recuperação de Produtos com Cache Memcached

```
// Função assíncrona para obter produtos com tratamento de cache usando Memcached
async getProducts(req, res) {
  try {
    // Recupera produtos do cache Memcached
    const cachedProducts = await new Promise((resolve, reject) => {
      memcachedClient.get('products', (err, value) => {
        if (err) return reject(err);
        resolve(value ? JSON.parse(value) : null); // Converte o valor JSON para objeto
      });
    });

    // Se os produtos estiverem no cache, retorna-os
    if (cachedProducts) {
      console.log("Produtos recuperados do cache");
      return res.json(cachedProducts);
    } else {
      // Se não encontrado no cache, busca no banco de dados e armazena no cache
      const products = await Product.find();
      await new Promise((resolve, reject) => {
        memcachedClient.set('products', JSON.stringify(products), 3600, (err) => {
          if (err) return reject(err);
          resolve(); // Indica que a operação de armazenamento foi bem-sucedida
        });
      });
      console.log("Produtos recuperados do banco de dados");
      return res.json(products);
    }
  } catch (err) {
    // Captura e registra erros, retornando um status de erro 500
    console.error('Erro:', err);
    return res.status(500).json({ message: "Server error" });
  }
}
```

Fonte: (Autoria Própria, 2024)

de operações de leitura. A estrutura do plano especifica variações na quantidade de dados e no número de *threads*, a fim de medir métricas como tempo de resposta e *throughput* sob diferentes condições de carga. Esse plano de teste detalha as ações necessárias para replicar consultas frequentes a produtos, característica típica de ambientes de *e-commerce*.

5.3.2 Implementar testes

Ainda com base nas diretrizes do YCSB e na necessidade de simular um cenário realista de e-commerce, definimos os parâmetros para nossos testes, adaptando-os às limitações do nosso ambiente local e às características específicas de uma aplicação de um e-commerce.

5.3.2.1 Definição do Tamanho e Quantidade de Dados

Para os testes realizados, foi necessário estabelecer as seguintes configurações:

1. **Tamanho dos registros de produtos:** O tamanho padrão foi definido como 1 KB por registro (10 campos de 100 bytes cada), tamanho este padronizado pelo YCSB Benchmarking. Esse valor foi escolhido para representar uma variação comum de objetos em um ambiente de *e-commerce*, como descrições de produtos, características e metadados.

2. **Número de produtos:** Foram utilizados três conjuntos de dados diferentes, com 10.000, 20.000 e 40.000 registros de produtos distintos, gerados por meio de um *script* personalizado. Esses valores foram selecionados para simular diferentes cargas de trabalho em cenários reais de sistemas de *e-commerce*. A utilização desses conjuntos de dados com 10.000, 20.000 e 40.000 registros, como proposto acima, encontra respaldo na literatura sobre testes de desempenho de bancos de dados. Abubakar *et al.* (2014) utilizaram quantidades semelhantes em seus experimentos com o *framework* YCSB, avaliando o desempenho de diferentes sistemas de *cache* sob *workloads* variadas, com número de registros entre 1.000 e 100.000.

3. **Threads concorrentes:** Os testes começarão com 10 *threads*, aumentando progressivamente para 20, 30, 50, 75, 100 e 125 *threads*. Cada *thread* representa um usuário virtual, simulando picos de tráfego em momentos de alta demanda e permitindo avaliar o impacto no sistema de cache. Essa abordagem é inspirada nas variações propostas por Salhi *et al.* (2018), que ressaltam a importância de analisar o desempenho de caches distribuídos sob diferentes níveis de concorrência. Contudo, nossa metodologia adapta essa estratégia ao utilizar múltiplas conexões de clientes, proporcionando uma avaliação mais abrangente do throughput sob condições variadas.

Tabela 2 – Quantidade de Registros e Tamanho Total de Cache

Tamanho do Registro	Quantidade de Registros	Tamanho Total do Cache
1 KB	10.000	10 MB
1 KB	20.000	20 MB
1 KB	40.000	40 MB

Fonte: (Autoria Própria, 2024)

5.3.3 Produção e Inserção dos Dados no Banco de Dados

Os dados são gerados de forma automatizada, utilizando um *script* (`script.js`) que cria objetos JSON com campos predefinidos. Este *script* será responsável por:

1. Gerar objetos JSON que, quando serializados, atingirão o tamanho definido (1 KB).
2. Criar strings aleatórias de tamanho controlado para campos como `description` e outros atributos textuais, preenchendo o espaço necessário.
3. Garantir que os objetos gerados sejam representativos de produtos reais em um *e-commerce*.

O código para implementação e replicação da geração de produtos está disponível em um repositório do GitHub[1]

5.4 EXECUTAR TESTES

Os planos de teste são configurados para enviar múltiplas requisições GET ao *endpoint* `/products` (conforme descrito na seção 5.2). Esses testes são executados diretamente via terminal, enquanto os serviços (Redis, Memcached, API, e MongoDB) permanecem conteinerizados no Docker.

A cada requisição, o processo segue passos: (i) inicialmente, o Redis ou Memcached é consultado para verificar se a lista de produtos já está armazenada no *cache*; (ii) se os produtos forem encontrados no *cache*, eles são imediatamente recuperados e retornados na resposta; (iii) se os produtos não estiverem no *cache*, a API realiza a busca diretamente no banco de dados primário (MongoDB), e o *cache* é atualizado com essas informações. Durante os testes, os logs dos conteineres Docker permitem monitorar quando os produtos são recuperados do *cache* ou do banco de dados. A Figura 20 ilustra esse comportamento, no qual a primeira requisição acessa o banco de dados, enquanto as requisições subsequentes são atendidas diretamente pelo *cache*.

As *threads* enviam requisições ao longo de 5 minutos (300 segundos). A figura 21 do*script* a seguir ilustra um exemplo da simulação:

Esse *script* configura uma simulação de carga para 20 *threads* (usuários virtuais) ativas, enviando requisições HTTP para o endpoint `/products` durante um período de

[1]Repositório do código-fonte geração de produtos: https://github.com/KermanJR/generateProductsTCC

Figura 20 – Requisições do Log do servidor da API.

```
Recebida requisição: GET /api/products
Produtos recuperados do banco de dados e armazenados no cache
Recebida requisição: GET /api/products
Recebida requisição: GET /api/products
Produtos recuperados do cache
Recebida requisição: GET /api/products
Produtos recuperados do cache
Recebida requisição: GET /api/products
Produtos recuperados do cache
Recebida requisição: GET /api/products
Produtos recuperados do cache
Recebida requisição: GET /api/products
Produtos recuperados do cache
```

Fonte: (Autoria Própria, 2024).

Figura 21 – Plano de teste gerado com a Ferramenta Gatling.

```
import {
  simulation,
  scenario,
  exec,
  constantUsersPerSec
} from "@gatling.io/core";
import { http, status } from "@gatling.io/http";

export default simulation((setUp) => {
  const search = exec(
    // Nome da requisição para identificar nos relatórios
    http("GET PRODUCTS - 100% READ")
    // Requisição GET no endpoint "/products"
    .get("/products")
    // Verifica se o status de resposta é 200 (OK)
    .check(status().is(200))
  );

  const httpProtocol = http
    .baseUrl("http://localhost:3001/api")  // URL base da API
    .acceptHeader("application/json");  // Define o tipo de resposta esperada (JSON)

  // Cenário simulando 20 usuários simultâneos
  setUp(
    scenario("Máximo de requisições - 20 threads (UV's)")
    .exec(search)  // Executa a requisição acima definida
    .injectOpen(
      // Injeta 20 usuários por segundo durante 300 segundos
      constantUsersPerSec(20).during(300)
    )
  ).protocols(httpProtocol);  // Aplica o protocolo HTTP configurado
});
```

Fonte: (Autoria Própria, 2024).

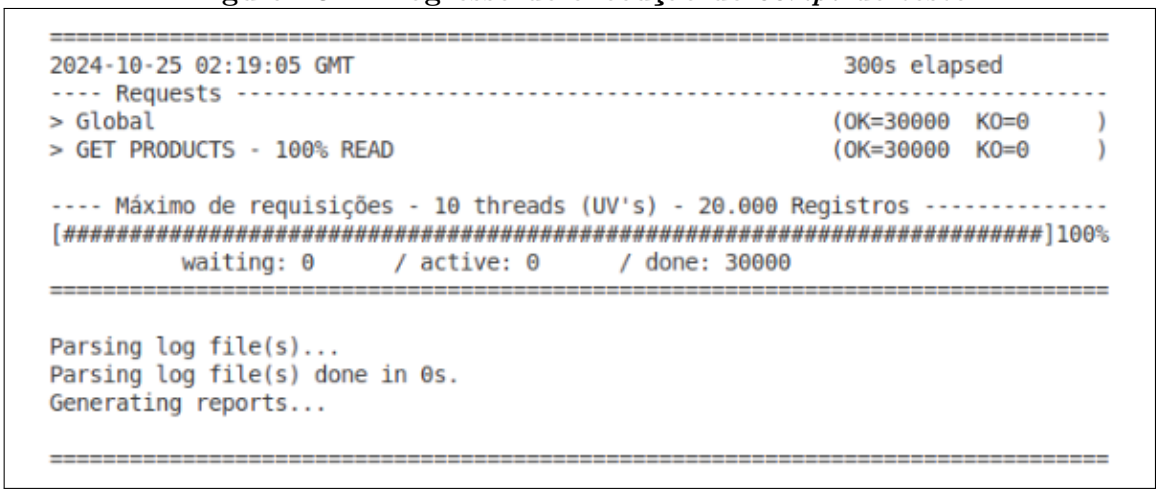
300 segundos. A requisição é uma operação de leitura (GET) e a simulação verifica se as respostas retornam o status HTTP 200 (OK). Esse cenário garante que as requisições sejam bem-sucedidas. O protocolo HTTP é configurado para se comunicar com a API localizada em http://localhost:3001/api e aceita respostas no formato `JSON`. A simulação injeta 20 *threads* (usuários virtuais) por segundo durante o período especificado. A execução do pode ser realizada via linha de comando de acordo com a figura 22 abaixo, utilizando o seguinte comando:

Figura 22 – Execução do *script* de teste via terminal.

Fonte: (Autoria Própria, 2024).

A linha de comando abaixo mostra o progresso dos testes:

Figura 23 – Progresso de execução do *script* de teste



Fonte: (Autoria Própria, 2024).

No exemplo da Figura 49, a linha de comando exibiu o progresso dos testes, indicando que 30.000 requisições foram processadas com sucesso, sem falhas. O teste completou a execução de 300 segundos com 100% das requisições concluídas. Ao final do teste, o Gatling gera relatórios com métricas detalhadas sobre o desempenho do sistema durante o teste de carga, incluindo as métricas de throughput e latência. A Figura 23 abaixo apresenta o relatório gerado com auxílio da ferramenta Gatling.

A linha de comando abaixo mostra o progresso dos testes:

5.5 LIMITAÇÕES DO AMBIENTE DE TESTE LOCAL

Os testes realizados neste trabalho foram conduzidos em um ambiente de teste local, utilizando uma máquina com especificações limitadas em termos de processamento, memória e capacidade de rede. Embora o ambiente tenha permitido a execução das simulações de carga, algumas limitações influenciaram os resultados obtidos. A falta de escalabilidade horizontal e o uso de um único nó para execução dos serviços limitam o volume de requisições que podem ser simuladas e introduzir gargalos em cenários de alta concorrência.

Outro fator limitante é a ausência de uma rede distribuída ou de um balanceamento de carga adequado. Em ambientes de produção, como clusters de Redis ou Memcached, a carga pode ser distribuída entre vários servidores, o que permite maior *throughput* e menor latência. Em contrapartida, em um ambiente local, todas as requisições são processadas por um único contêiner, o que impacta diretamente no desempenho sob cargas elevadas.

Portanto, os resultados apresentados podem não refletir o potencial máximo das tecnologias testadas, uma vez que o ambiente local introduziu limitações de *hardware* e infraestrutura. Para mitigar esses impactos, foram realizadas múltiplas execuções de cada teste, visando capturar variações de desempenho, que podem oferecer informações adicionais sobre a aplicação das tecnologias em contextos mais amplos.

O código-fonte da API de um *e-commerce*, desenvolvido utilizando a arquitetura de microsserviços em conjunto com a tecnologia Redis, pode ser encontrado em um repositório público no GitHub[2].

O código-fonte da implementação utilizando a tecnologia Memcached está disponível em outro repositório no GitHub[3].

[2]Repositório do código-fonte Redis: https://github.com/KermanJR/architecture-microsservices-nodejs-redis

[3]Repositório do código-fonte Memcached: https://github.com/KermanJR/architecture-memcached-nodejs

6 RESULTADOS E DISCUSSÕES

6.1 CONSIDERAÇÕES INICIAIS

Neste capítulo são apresentados os resultados obtidos a partir da execução de testes de carga com foco em operações intensivas de leitura utilizando as tecnologias Redis e Memcached. Os testes foram conduzidos em um ambiente local de microsserviços, utilizando o *workload* C do YCSB, que simula 100% de leitura intensiva, variando o número de *threads* (usuários virtuais) e o tamanho dos dados, conforme descrito no capítulo 5. A análise quantitativa/descritiva dos resultados obtidos visa identificar o comportamento de cada solução de *cache* frente a diferentes cargas de requisições em um ambiente de microsserviços. As métricas coletadas incluem o tempo de resposta médio e o *throughput* (requisições por segundo).

6.2 ANÁLISE DE DESEMPENHO DO REDIS E MEMCACHED

6.2.1 Descrição Geral do Comportamento de Redis e Memcached

É importante considerar a virtualização da API de *e-commerce* no ambiente Docker durante a execução desses testes, pois a utilização de contêineres pode introduzir latência adicional no ambiente de testes. A sobrecarga de iniciar o Docker e gerenciar os contêineres pode resultar em latência extra, o que pode afetar o desempenho geral, especialmente quando se utiliza contêineres aninhados, onde um contêiner pai hospeda outros contêineres filhos (AMARAL *et al.*, 2015).

6.2.2 Análise de 10 mil Registros

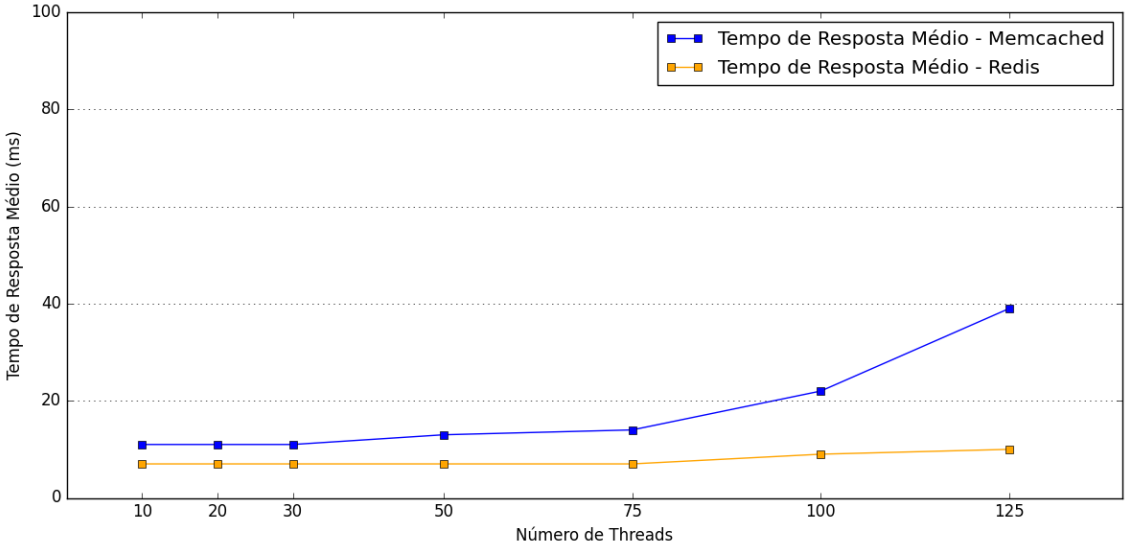
Para 10 mil registros de 1 KB, o Redis pode se destacar devido à sua arquitetura *single-threaded*. Ele usa *I/O multiplexing* para gerenciar várias conexões de forma eficiente, sem sobrecarregar o sistema com operações de troca de contexto entre *threads*. Esse modelo resulta em tempos de resposta baixos, com um tempo médio de 7 ms até 75 *threads*. Um fator que pode ser importante para esse desempenho é a implementação otimizada de *strings* no Redis. Ele utiliza a estrutura *Simple Dynamic String* (SDS), que mantém o comprimento da *string* em *cache* e faz pré-alocação de memória, evitando realocações frequentes (REDIS, 2023). Essa combinação de técnicas contribui para a

estabilidade nos tempos de resposta, especialmente em operações de leitura intensiva com strings de tamanho fixo, como os registros de 1 KB usados nos testes.

Em comparação, o Memcached adota uma arquitetura *multi-threaded*, o que pode ser mais adequado para cenários com um alto número de requisições. Inicialmente, os testes indicam um tempo de resposta médio de 13 ms com 10 threads, que aumenta progressivamente para 39 ms com 125 threads.

Em relação às *strings*, o Memcached trata *strings* de maneira mais simples do que o Redis, tratando-as como sequências de bytes armazenadas em *chunks* de memória gerenciados pelo *slab allocator*. Esse modelo, embora mais direto, é otimizado para o principal caso de uso do Memcached: o armazenamento e recuperação rápida de valores de tamanho fixo. No contexto de 1 KB de dados, o Memcached seleciona automaticamente a *slab class* adequada para acomodar o tamanho total da *string*, o que ajuda a minimizar o desperdício de memória (ATIGLOKU *et al.*, 2012).

Figura 24 – Tempo de Resposta Médio por Número de Threads para 10 mil registros



Fonte: (Autoria Própria, 2024).

6.2.3 Análise de 20 mil Registros

Para 20 mil registros, observa-se uma tendência intermediária entre os cenários de 10 mil e 40 mil registros, evidenciando como o aumento do volume de dados impacta gradualmente o desempenho de ambos os sistemas.

O Redis mantém um comportamento relativamente estável, com tempo de

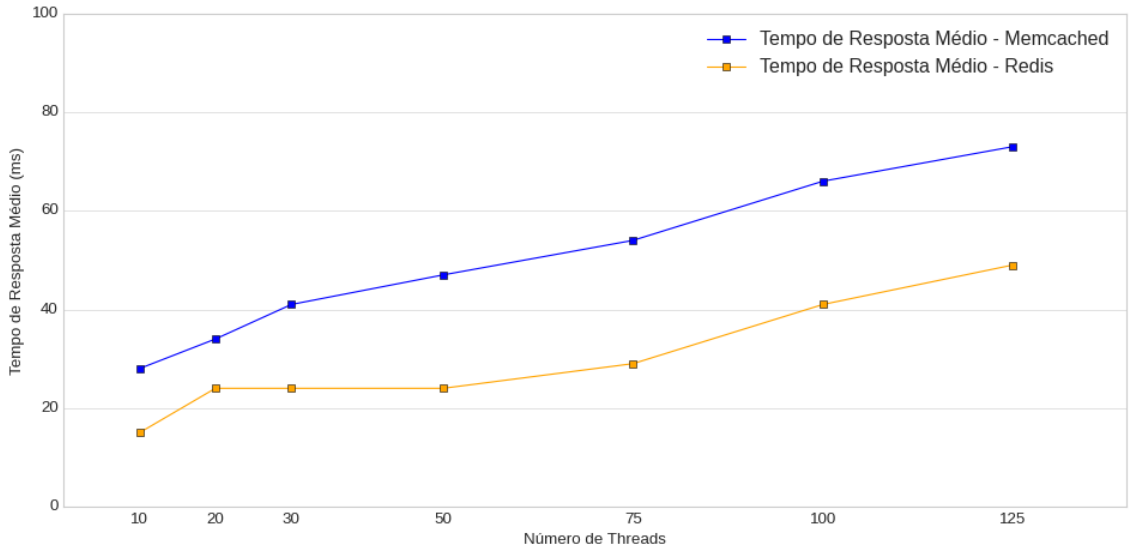
resposta inicial de aproximadamente 15 ms com 10 *threads*, aumentando moderadamente para 49 ms com 125 *threads*. Este padrão de degradação controlada sugere que sua arquitetura *single-threaded* continua eficiente mesmo com o dobro da carga de dados, possivelmente devido ao seu sistema de gerenciamento de memória otimizado e ao modelo de *I/O multiplexing* (ZHANG *et al.*, 2015). O mecanismo de persistência em memória do Redis, combinado com sua estratégia de gerenciamento de eventos, parece manter sua eficiência mesmo com o aumento do volume de dados. A utilização do jemalloc como alocador de memória continua demonstrando sua eficácia na prevenção de fragmentação de memória, contribuindo para a manutenção de tempos de resposta consistentes mesmo com o dobro da carga de dados (REDIS, 2023).

O Memcached, por sua vez, continua demonstrando indícios de uma degradação levemente acentuada em comparação com o cenário de 10 mil registros. Os tempos de resposta iniciam em aproximadamente 28 ms com 10 *threads* e escalam para cerca de 73 ms com 125 *threads*. Este comportamento intermediário sugere que o sistema começa a enfrentar desafios mais significativos com o aumento do volume de dados, possivelmente devido à maior pressão sobre seu mecanismo de gerenciamento de memória e sincronização entre *threads*.

A arquitetura *multi-threaded* do Memcached começa a mostrar sinais de sobrecarga mais evidentes neste volume de dados. O *slab allocator*, mesmo sendo eficiente na gestão de memória, parece enfrentar maior pressão com o aumento do volume de dados, resultando em tempos de resposta mais elevados em comparação com o Redis (ATIGLOKU *et al.*, 2012). Este comportamento pode ser atribuído aa sobrecarga adicional necessário para manter a consistência entre as múltiplas *threads* e gerenciar a alocação de memória para um volume maior de dados.

Estes resultados sugerem que, para cargas de trabalho intermediárias de 20 mil registros, ambos os sistemas ainda mantêm um desempenho aceitável, mas começam a demonstrar características distintas em suas capacidades de escala. O Redis continua apresentando indícios de maior estabilidade e tempos de resposta mais baixos, enquanto o Memcached começa a mostrar sinais dos desafios inerentes à sua arquitetura *multi-threaded* quando submetido a volumes crescentes de dados.

Figura 25 – Tempo de Resposta Médio por Número de *Threads* para 20 mil registros



Fonte: (Autoria Própria, 2024).

6.2.4 Análise de 40 mil Registros

Com 40 mil registros, o comportamento dos sistemas pode nos oferecer informações importantes sobre seu desempenho, especialmente em relação à eficiência no gerenciamento de memória com grandes volumes de dados.

O Redis demonstra uma degradação controlada, com tempo de reposta inicial de 49 ms com 10 *threads* aumentando para 91 ms em 125 *threads*. Percebe-se um aumento gradual, embora não muito expressivo, no volume de dados. Esse comportamento pode ser atribuído não apenas à sua arquitetura *single-threaded*, mas também à implementação eficiente da *Simple Dynamic String* (SDS).

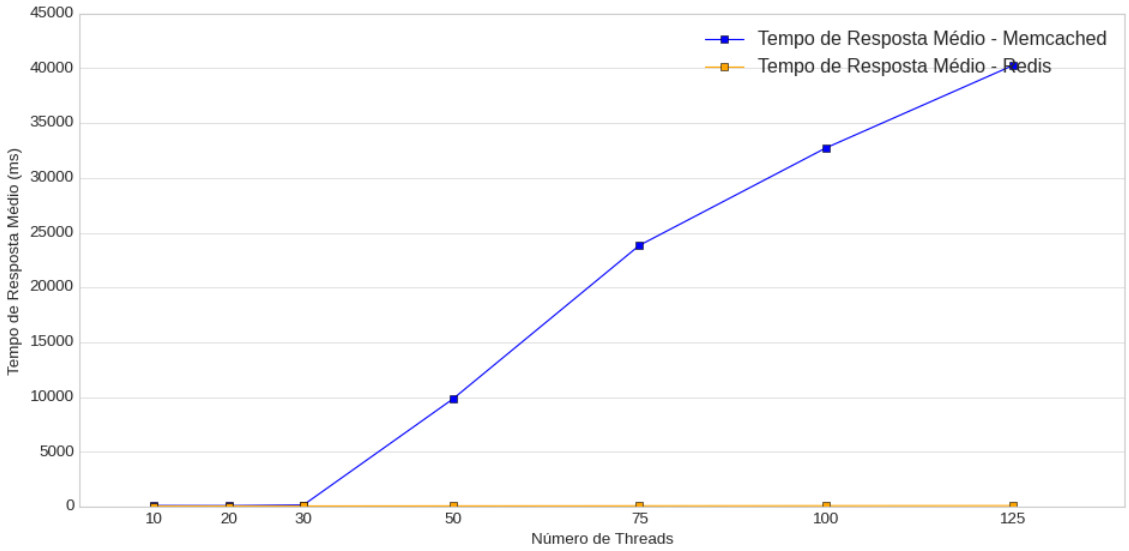
Em volumes maiores de dados, Zhang *et al.* (2014) observaram que a eficiência do SDS aparenta ser mais significativa, possivelmente devido à sua estratégia de pré-alocação e gerenciamento de memória, que tende a minimizar a fragmentação e as operações de realocação. A implementação do SDS no Redis mostra particular eficiência quando lidando com este volume de dados por vários motivos. Primeiro, o sistema de codificação automática do Redis permite otimizações específicas baseadas no conteúdo das *strings*, reduzindo a sobrecarga de memória. Segundo, o mecanismo de *lazy-free* do Redis permite que operações de remoção de grandes strings sejam processadas em background, evitando bloqueios longos na *thread* principal. Esta característica pode se tornar um fator relevante com 40 mil registros, onde operações de gerenciamento de memória são mais frequentes e intensivas (REDIS, 2023).

No entanto, mesmo com estas otimizações, observa-se um aumento no tempo de resposta que pode ser significativo para aplicações com requisitos rigorosos de tempo de resposta. De acordo com o estudo de Abubakar *et al.*, (2014), este comportamento pode ser atribuído ao crescimento mais acentuado da fila de processamento, onde, mesmo com o eficiente gerenciamento de *strings* do Redis, todas as operações necessitam ser serializadas através de uma única *thread*. Além disso, o estudo feito por Zhang *et al.*, (2014) indica que o modelo *single-threaded* do Redis limita a escalabilidade, pois o servidor pode não atender às solicitações de outros servidores, como por exemplo, durante a execução de um *script* de teste de longa duração, que é o caso deste trabalho.

Em contraste, há indícios de que o Memcached apresenta uma degradação muito mais acentuada neste cenário. Com tempos de resposta iniciando em 95 ms com 10 *threads* e escalando exponencialmente para mais de 32 mil milissegundos com 125 *threads*, o sistema pode indicar claras limitações em sua capacidade de gerenciar grandes volumes de dados.

A arquitetura *multi-threaded* do Memcached, que teoricamente deveria proporcionar melhor escalabilidade, pode acabar exacerbando os problemas de contenção quando combinada com seu sistema de gerenciamento de *strings*. A alocação de memória de *slabs* pode levar a um problema denominado "calcificação", onde a alocação de memória se torna ineficiente quando padrões de requisições mudam, como o número de *threads* e quantidade de registros (CARRA e MICHIARDI, 2019). Este efeito pode ser amplificado pelo fato de que o Memcached não implementa otimizações específicas para diferentes tipos de conteúdo de *string*, tratando todos os dados de maneira uniforme, assim como outros tipos de dados (FITZPATRICK, 2004).

Figura 26 – Tempo de Resposta Médio por Número de *Threads* para 40 mil registros



Fonte: (Autoria Própria, 2024).

6.3 ANÁLISE DE *THROUGHPUT* (REQUISIÇÕES POR SEGUNDO)

A capacidade de processamento de requisições por segundo (*throughput*) apresenta-se como uma métrica relevante para avaliar o desempenho dos sistemas de *cache* em ambiente de microsserviços.

Antes de realizar a análise dos dados, é importante entender o significado das colunas apresentadas na tabela 3. A coluna "Número de *Threads*"se refere à quantidade de processos/tarefas simultâneas que estão sendo executadas para fazer requisições aos sistemas de *cache*. Quando há 100 *threads*, significa que 100 solicitações estão sendo feitas ao mesmo tempo, de forma concorrente.

Já as colunas "Redis (Requisições por Segundo)"e "Memcached (Requisições por Segundo)"indicam quantas dessas requisições simultâneas cada sistema consegue processar e responder por segundo. Portanto, quando o valor é 100, significa que o Redis ou o Memcached é capaz de atender 100 requisições por segundo quando há 100 *threads* fazendo solicitações concorrentes.

Tabela 3 – Análise de Requisições por Segundo

Tamanho do Registro	Número de Threads	Redis (Requisições por Segundo)	Memcached (Requisições por Segundo)
10.000	100	100 ms	99.67 ms
10.000	125	125 ms	122.95 ms
20.000	100	100 ms	98 ms
20.000	125	125 ms	124 ms
40.000	100	98 ms	67 ms
40.000	125	122.69 ms	50 ms

Fonte: (Autoria Própria, 2024).

Para volumes de 10 mil e 20 mil registros, observa-se indícios de um comportamento similar entre Redis e Memcached. Com 10 mil registros, os dados indicam que o Redis alcança aproximadamente 100 req/s com 100 *threads* e cerca de 125 req/s com 125 *threads*, enquanto o Memcached aparentemente atinge 99,67 req/s e 122,95 req/s respectivamente. Para 20 mil registros, o Redis parece manter cerca de 100 req/s com 100 *threads* e aproximadamente 125 req/s com 125 threads, enquanto o Memcached apresenta indícios de 98 req/s e 124 req/s respectivamente.

Ao aumentar o volume para 40 mil registros, os resultados sugerem uma divergência levemente mais acentuada, onde o Redis aparentemente mantém sua performance com aproximadamente 98 req/s para 100 *threads* e cerca de 122,69 req/s para 125 *threads*, enquanto o Memcached apresenta indícios de uma queda expressiva para aproximadamente 67 req/s com 100 *threads* e 50 req/s com 125 *threads*. No caso do *throughput*, o uso do protocolo de comunicação TCP no Redis e Memcached pode impactar os resultados, especialmente em operações com objetos pequenos, devido à latência de comunicação associada ao protocolo. Durante a transferência de dados, o TCP exige múltiplas chamadas de sistema, aumentando o consumo de CPU e causando perdas de cache de instruções, o que impacta levemente a taxa de transferência.

Em um estudo conduzido por Zhang *et al.* (2014), foi observado que, para objetos entre 10 bytes e 1 KB, o *throughput* em requisições por segundo permanece constante para ambas as tecnologias de cache. Isso implica que a transferência de objetos de 10 bytes pode ser tão custosa quanto a de 10 KB, gerando uma ligeira queda na taxa de requisições para objetos entre 1 KB e 10 KB, e uma diminuição significativa para objetos maiores, indicando uma melhor utilização da largura de banda. Essa ineficiência do TCP em transferir pequenos objetos pode ser relevante em aplicações que manipulam grande volume de dados pequenos, como no caso deste trabalho, onde registros de aproximadamente 1 KB são utilizados.

Desta forma, ao analisar os resultados do *throughput*, há indícios de que Redis pode apresentar melhor adequação para serviços que necessitam escalar horizontalmente, ao se sobressair em volumes de dados maiores, enquanto o Memcached aparentemente demonstra maior impacto do protocolo TCP, o que pode introduzir latência adicional ao realizar requisições em um ambiente Docker. Além disso, um estudo realizado por Kausar et al. (2022) sugere que o Redis mantém um *throughput* estável, mesmo com o aumento no número de requisições.

Os resultados dos testes estão disponíveis em um repositório público no GitHub[1].

6.4 DISCUSSÃO DOS PRINCIPAIS RESULTADOS

Os resultados sugerem comportamentos distintos entre Redis e Memcached em um ambiente de microsserviços, especialmente em cargas de trabalho intensivas em leitura executadas em containers Docker. Indícios apontam que o Redis, com sua arquitetura *single-threaded*, poderia proporcionar uma eficiência superior na manutenção de baixos tempos de resposta, mesmo com o aumento gradual de *threads*. Essa observação pode estar relacionada ao uso de estruturas como *Simple Dynamic Strings* (SDS) e ao gerenciamento de memória otimizado pelo Jemalloc, que parecem reduzir a fragmentação e a sobrecarga em operações de leitura. Em contrapartida, o Memcached, com sua arquitetura *multi-threaded* e modelo de alocação de memória em slabs, apresentou indícios de bom desempenho inicial em volumes menores de dados e *threads* moderadas.

Profissionais podem usar os resultados para ajustar suas arquiteturas de microsserviços com base nas necessidades específicas de seus sistemas:

- Para *e-commerce* ou sistemas que lidam com grande volume de leituras rápidas, o Redis pode ser a melhor opção devido à sua baixa latência e alta performance em leituras.
- Em sistemas de alta concorrência ou com operações simples e dados pequenos, o Memcached pode ser preferível, embora com a consideração de que o volume de dados não deve ser muito alto para evitar degradação de desempenho.

Além disso, a análise de *throughput* pode sugerir que o uso do protocolo TCP em ambos os sistemas pode introduzir limitações, especialmente em operações com objetos

[1]Repositório de resultados: https://github.com/KermanJR/resultados-redis_vs_memcached

pequenos, resultando em pequenas perdas de desempenho. Os dados analisados parecem sugerir que o Redis mantém uma maior estabilidade no *throughput* com volumes de dados elevados, o que pode torná-lo mais adequado para aplicações que demandam escalabilidade horizontal em ambientes de microsserviços.

Em relação ao *feature models* do Redis e do Memcached há diferenças significativas no gerenciamento de *strings*, com implicações relevantes para o desenvolvimento de aplicações de microsserviços.

O modelo do Redis apresenta uma abordagem mais sofisticada e customizada para o armazenamento de *strings*, com um uso mais eficiente da memória, com strings pequenas (<44 bytes) sendo alocadas de forma compacta utilizando a codificação embstr, enquanto strings médias e grandes são alocadas dinamicamente usando o Jemalloc. O Jemalloc organiza as alocações em classes de tamanho específicas, visando minimizar a fragmentação de memória e otimizar o desempenho das operações sobre as strings.

Por outro lado, o modelo do Memcached adota uma abordagem baseada em slab, com a divisão do armazenamento em classes de tamanho (*small*, *medium*, *large*) gerenciadas de forma independente. Essa organização permite um gerenciamento mais granular da memória, reduzindo o desperdício e melhorando o desempenho das operações sobre as strings.

Essas características dos modelos de recursos têm implicações relevantes no contexto de aplicações de microsserviços e e-commerce, que lidam com grandes volumes de dados representados como strings. No cenário de microsserviços, os modelos de recursos podem auxiliar os engenheiros de software a:

- Selecionar a tecnologia (Redis ou Memcached) mais adequada para o armazenamento e recuperação de *strings*, com base nos padrões de uso da aplicação.
- Dimensionar de maneira mais eficiente a infraestrutura, alocando recursos de memória de acordo com as necessidades específicas de cada microsserviço.
- Projetar a interação entre microsserviços que dependem do armazenamento e manipulação de *strings* de forma otimizada.

Em aplicações *e-commerce*, os modelos de recursos podem ajudar os desenvolvedores a:

- Determinar a melhor abordagem para armazenamento e gerenciamento de *strings*

relacionadas a clientes, produtos, pedidos e outras entidades críticas para o desempenho da aplicação.

- Projetar sistemas de *cache* eficientes, aproveitando as características específicas de cada tecnologia (como as diferentes codificações de *strings* no Redis) para melhorar a resposta da aplicação.
- Garantir a escalabilidade e a disponibilidade da aplicação *e-commerce*, utilizando recursos como o Redis Cluster e a distribuição de carga do Memcached.

Em resumo, a comparação entre os feature models do Redis e do Memcached oferece informações importantes sobre como gerenciar *strings* de forma a obter um melhor desempenho. Isso ajuda engenheiros de software e desenvolvedores a escolher a tecnologia mais adequada para suas aplicações, especialmente em microsserviços e *e-commerce*, com o objetivo de melhorar o desempenho, a escalabilidade e a disponibilidade desses sistemas.

7 CONCLUSÃO

Este trabalho realizou uma análise comparativa entre as tecnologias de *cache* Redis e Memcached no contexto de uma arquitetura de microsserviços, avaliando o desempenho dessas tecnologias de *cache* na recuperação e armazenamento de dados em uma aplicação web. A implementação do protótipo para comparação das tecnologias de *cache* foi realizada em containers Docker, garantindo um ambiente controlado, padronizado e isolado para os testes de carga, facilitando a comparação justa e objetiva entre as duas soluções.

Foi adotada a representação de *feature models* para modelar, descrever e ilustrar as features específicas de cada tecnologia, com foco no uso de *strings* como estrutura de dados chave-valor. Essa abordagem forneceu orientações úteis para desenvolvedores e arquitetos de software sobre a utilização de *cache* no contexto de microsserviços e destacou a importância de soluções de *cache* para melhorar o tempo de resposta e reduzir a latência em sistemas web

Este trabalho contribui com:

- Uma avaliação comparativa de Redis e Memcached em uma arquitetura de microsserviços, utilizando métricas de desempenho como tempo de resposta, *throughput*;
- A criação de *feature models* que detalham as funcionalidades de Redis e Memcached, especialmente para o uso de *strings*, facilitando a visualização de suas capacidades e limitações;
- A implementação dos testes em conteineres Docker, o que promove a reprodutibilidade e a padronização dos experimentos em um ambiente controlado;
- Um conjunto de recomendações para a escolha de soluções de *cache* de acordo com o contexto da aplicação, considerando variáveis como volume de dados e *threads* (usuários virtuais).

Os *feature models* são uma forma de representação que organiza e visualiza as *features* das tecnologias de *cache*, como Redis e Memcached, servindo como uma ferramenta para compreender melhor as funcionalidades de cada tecnologia de *cache* e suas diferenças. Além disso, eles podem auxiliar na identificação de limitações e na

adaptação das soluções às necessidades específicas de sistemas web, proporcionando uma visão estruturada que apoia o processo de decisão e configuração.

O escopo deste trabalho apresenta algumas limitações:

- A análise comparativa das tecnologias de cache foi restrita ao uso de *strings* como estrutura de dados, sem explorar outras opções do Redis, como listas, conjuntos e *hashes*, nem suas possibilidades avançadas.
- Os testes ocorreram em um ambiente controlado, o que pode não refletir o desempenho real em infraestruturas mais complexas.
- O estudo também não incluiu outras soluções de *cache* que poderiam ser relevantes, nem avaliou configurações específicas como clusterização, replicação e persistência em disco, que são cruciais em cenários de alta disponibilidade e recuperação de falhas.
- Finalmente, a pesquisa focou exclusivamente em arquiteturas de microsserviços, deixando de considerar o desempenho das tecnologias em arquiteturas híbridas ou monolíticas.

Essas limitações indicam oportunidades para estudos futuros, que podem expandir e aprofundar a análise realizada.

No escopo deste trabalho, *feature models* oferecem potencial para futuras pesquisas e automações, especialmente no contexto de microsserviços. Para isso, recomenda-se como próximos passos:

- Ampliar os testes para incluir outros tipos de dados e casos de uso, como armazenamento de objetos JSON e listas, para explorar plenamente as funcionalidades do Redis e do Memcached;
- Desenvolver uma análise comparativa do impacto de soluções de *cache* no consumo de recursos, como memória e CPU, em diferentes configurações de infraestrutura, considerando variações no volume de dados e frequência de acesso em um ambiente de microsserviços.
- Explorar a adaptação dos *feature models* para incluir uma gama mais ampla de tecnologias de *cache*. Assim, os feature models poderiam fornecer recomendações personalizadas para diferentes requisitos de desempenho e escalabilidade.

Finalmente, esta pesquisa oferece uma primeira base para estudos futuros sobre a otimização de desempenho em arquiteturas de microsserviços, abordando o impacto do *cache* no desempenho de sistemas web. Com a implementação de práticas de *caching* e o uso de tecnologias como Docker, desenvolvedores podem criar sistemas web eficientes e com menor tempo de resposta, promovendo a escalabilidade e robustez de sistemas web modernos.

REFERÊNCIAS

ABUBAKAR, Yusuf et al. “Performance Evaluation of NoSQL Systems Using YCSB in a Resource Austere Environment.” *International Journal of Applied Information Systems*, vol. 7, no. 8, 4 Sept. 2014, pp. 23–27, https://doi.org/10.5120/ijais14-451229. Acesso em: 4 jul. 2024.

AMARAL, Marcelo et al. “Performance Evaluation of Microservices Architectures Using Containers.” 2015 IEEE 14th International Symposium on Network Computing and Applications, Sept. 2015, https://doi.org/10.1109/nca.2015.49.

ATIKOGLU, Berk et al. “Workload Analysis of a Large-Scale Key-Value Store.” *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, 7 June 2012, p. 53, https://doi.org/10.1145/2318857.2254766. Acesso em: 4 jul. 2024.

ALBIN, S. T. *The Art of Software Architecture*. [S.l.]: John Wiley & Sons, 2003.

APEL, Sven et al. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin, Heidelberg: Springer, 2013.

BARBACCI, Mario; KLEIN, Mark H.; LONGSTAFF, Thomas A.; WEINSTOCK, Charles B. Quality Attributes. Technical Report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, 1995.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. Addison-Wesley, 2012.

BELSHE, Mike; PEON, Roberto; THOMSON, Martin. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, Maio 2015. Disponível em: https://www.rfc-editor.org/rfc/rfc7540. Acesso em: 23 set. 2024.

BELL, M. *Service-oriented Modeling: Service Analysis, Design, and Architecture*. John Wiley & Sons, 2008.

BELLEMARE, A. *Building Event-Driven Microservices*. [S.l.]: O’Reilly Media, 2020.

BERNERS-LEE, Tim; FIELDING, Roy; FRYSTYK, Henrik. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, Maio 1996. Disponível em: https://www.rfc-editor.org/rfc/rfc1945. Acesso em: 23 set. 2024.

BUSCHMANN, Frank *et al*. *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester: J. Wiley, 2000.

CYSNEIROS, L. M.; LEITE, J. C. S. P. Nonfunctional requirements: from elicitation to conceptual models. *IEEE Transactions on Software Engineering*, v. 30, n. 5, p. 328-350, 2004.

CHUNG, Lawrence; NIXON, Brian A.; YU, Eric; MYLOPOULOS, John. *Non-Functional Requirements in Software Engineering*. Dordrecht: Springer, 2012.

CICERI, Claudio et al. *Software Architecture Metrics: Case Studies to Improve the Quality of Your Architecture*. Sebastopol: O'Reilly Media, Inc., 2022.

CLEMENTS, Paul; NORTHROP, Linda. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

COLE, Bryan. Performance Engineering for Retail and E-commerce Apps. 2024. Disponível em: https://www.tricentis.com/blog/performance-engineering-for-retail-and-e-commerce-apps. Acesso em: 18 maio 2024.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. Staged Configuration Using Feature Models. In: *Software Product Lines*. [S.l.: s.n.], 2004.

DAVISON, Brian D. A Web Caching Primer. *IEEE Internet Computing*, v. 5, n. 4, p. 38–45, 2001.

DECANDIA, Giuseppe; *et al.* Dynamo. *ACM SIGOPS Operating Systems Review*, v. 41, n. 6, p. 205, 14 out. 2007. Disponível em: https://doi.org/10.1145/1323293.1294281. Acesso em: 4 jul. 2024.

DRAGONI, Nicola *et al.* Microservices: Yesterday, Today, and Tomorrow. *Present and Ulterior Software Engineering*, p. 195-216, 2017. Disponível em: https://arxiv.org/pdf/1606.04036. Acesso em: 4 jul. 2024.

DUFFY, MEIER, Jim. *Performance Testing Guidance for Web Applications*. Microsoft Press, 2013.

FAIRBANKS, George. *Just Enough Software Architecture: A Risk-Driven Approach*. Boulder, Colo: Marshall & Brainerd, 2012.

FORD, N. et al. *Software Architecture: the Hard Parts*. [S.l.]: O'Reilly Media, Inc., 2021.

FIELDING, Roy; GETTYS, Jim; MOGUL, Jeffrey; FRYSTYK, Henrik; MASINTER, Larry; LEACH, Paul; BERNERS-LEE, Tim. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Junho 1999. Disponível em: https://www.rfc-editor.org/rfc/rfc2616. Acesso em: 23 set. 2024.

FIELDING, Roy; RESCHKE, Julian. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, IETF, 2014.

FOWLER, Martin; LEWIS, James. Microservices: A Definition of This New Architectural Term. *MartinFowler.com*, 2014. Disponível em: https://martinfowler.com/articles/microservices.html.

FOWLER, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

FOWLER, Martin. What do you mean by "Event-Driven"? *MartinFowler.com*, 2017. Disponível em: https://martinfowler.com/articles/201701-event-driven.html. Acesso em: 4 jul 2024.

GATLING. Gatling Documentation. Disponível em: https:// docs.gatling.io/ .Acessoem: 4jul.2024 .

GAMMA, Erich *et al. Design Patterns: Elements of Reusable Object-Oriented Software.* Boston: Addison-Wesley, 1994.

GERALDI, R. T. PROCESSO PARA MODELAGEM DE VARIABILIDADES DE SISTEMAS CIBER-FÍSICOS APOIADO POR FEATURE MODELS. 2022. Tese (Doutorado) — Pontifícia Universidade Católica do Paraná.

GERALDI, R. T.; REINEHR, S.; MALUCELLI, A. Software product line applied to the internet of things: A systematic literature review. *Information and Software Technology*, 2020.

GLINZ, Martin. On Non-Functional Requirements. In: *15th IEEE International Requirements Engineering Conference (RE 2007)*, Delhi, 2007. p. 21-26. Disponível em: https://doi.org/10.1109/RE.2007.45. Acesso em: 23 set. 2024.

GRIGORIK, Ilya. *High Performance Browser Networking.* O'Reilly Media, 2013.

HENNESSY, John L.; PATTERSON, David A. *Computer Architecture: A Quantitative Approach.* Elsevier, 2011.

JAIN, Raj. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley, 1991.

KAUSAR, Mohammad Abu et al. "A Study of Performance and Comparison of NoSQL Databases: MongoDB, Cassandra, and Redis Using YCSB." *Indian Journal of Science and Technology*, vol. 15, no. 31, 21 Aug. 2022, pp. 1532–1540, https://doi.org/10.17485/ijst/v15i31.1352.

KANG, Kyo C. et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report, CMU/SEI-90-TR-021, 1990.

KANIZAWA, Danilo Takeo; PINTO, Giuliano Scombatti. Arquitetura de microsserviços. Infa, [S. l.], v. 19, n. 2, p. 308-318, 20 dez. 2022.

KILLELEA, Patrick. *Web Performance Tuning.* [S.l.]: O'Reilly Media, Inc., 2002.

MENASCÉ, Daniel A. Load Testing of Web Sites. George Mason University, 2002. Disponível em: https://cs.gmu.edu/~menasce/papers/loadTesting.pdf. Acesso em: 17 ago. 2024.

MOLYNEAUX, Ian. *The Art of Application Performance Testing.* [S.l.]: O'Reilly Media, Inc., 2009.

NEWMAN, Sam. *Building Microservices.* 2. ed. Sebastopol: O'Reilly Media, 2015.

NOTTINGHAM, Mark. HTTP/2 Frequently Asked Questions. 2015. Disponível em: https://http2.github.io/faq/.

OSEMWEGIE, Omoruyi; *et al.* Performance Benchmarking of Key-Value Store NoSQL Databases. *International Journal of Electrical and Computer Engineering (IJECE)*, v. 8, n. 6, p. 5333-5341, 1 dez. 2018. Disponível em: https://doi.org/10.11591/ijece.v8i6.pp5333-5341. Acesso em: 4 jul 2024.

PEZZE, Mauro; YOUNG, Michal. *Software Testing and Analysis: Process, Principles, and Techniques*. Hoboken, N.J.: John Wiley & Sons Inc., 2007. ISBN 978-0-471-45593-6.

PODLIPNIG, Stefan; BÖSZÖRMENYI, Laszlo. A Survey of Web Cache Replacement Strategies. *ACM Computing Surveys*, v. 35, n. 4, p. 374-398, 2003.

POHL, Klaus; BÖCKLE, Günter; LINDEN, Frank J. van der. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

POHL, Klaus; RUPP, Chris. *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam – Foundation Level – IREB compliant*. 1. ed. Rocky Nook, 2015.

RAUL, Wagner. *Engenharia de Software: Conceitos e Práticas*. [S.l.]: Elsevier Editora Ltda., 2019.

REINEHR, S. *Engenharia de Requisitos*. [S.l.]: Grupo A, 2020.

RICHARDS, Mark. *Software Architecture Patterns*. [S.l.]: O'Reilly Media, Inc., 2022.

RICHARDS, Mark; FORD, Neal. *Fundamentals of Software Architecture*. Sebastopol, CA: O'Reilly Media, Inc., 2021.

RICHARDSON, Chris. *Microservices Patterns: with Examples in Java*. Shelter Island, New York: Manning Publications, 2018.

ROZANSKI, Nick; WOODS, Eoin. *Software Systems Architecture*. [S.l.]: Pearson Education, 2005.

SALHI, Haytham et al. "Benchmarking and Performance Analysis for Distributed Cache Systems: A Comparative Case Study." *Lecture Notes in Computer Science*, 28 Dec. 2017, pp. 147–163, https://doi.org/10.1007/978-3-319-72401-0_11. Acesso em: 9 Nov. 2024.

SILL, Alan. "The Design and Architecture of Microservices."*IEEE Cloud Computing*, v.3, n.5, p.76-80, 2016. Disponível em: https://ieeexplore.ieee.org/document/7742259. Acesso em: 4 jul. 2024.

SHAW, Mary; GARLAN, David. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

SMITH, Connie U.; WILLIAMS, Lloyd G. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Boston: Addison-Wesley, 2002.

SOMMERVILLE, Ian. *Software Engineering*. 10. ed. Harlow: Pearson Education, 2016.

SUBRAYA, B. M. *Integrated Approach to Web Performance Testing: A Practitioner's Guide*. Hershey: IRM Press, 2006.

THIEL, Steffen; HEIN, Andreas. Modeling and Using Product Line Variability in Automotive Systems. In: *Proceedings of the 2nd International Conference on Software Product Lines*, 2002. p. 1-12.

TAIBI, Davide; LENARDUZZI, Valentina; PAHL, Claus. Architectural Patterns for Microservices: A Systematic Mapping Study. *Proceedings of the 8th International Conference on Cloud Computing and Services Science*, 2018. Disponível em: https://www.scitepress.org/papers/2018/67983/67983.pdf. Acesso em: 4 jul. 2024.

UEDA, Takanori; et al. Workload Characterization for Microservices. In: 2016 IEEE International Symposium on Workload Characterization (IISWC), 2016. Disponível em: https://ieeexplore.ieee.org/document/7581269/. DOI: https://doi.org/10.1109/iiswc.2016.7581269. Acesso em: 4 jul. 2024.

VALENTE, M. T. *Engenharia de Software Moderna*. 2020.

VERNON, Vaughn. *Implementing Domain-Driven Design*. 1. ed. Addison-Wesley Professional, 2013.

WALLS, Craig. *Spring in Action*. 4. ed. Manning Publications, 2015.

WIEGERS, Karl; BEATTY, Joy. *Software Requirements*. Sydney: Pearson Education, 2013.

WIEGERS, Karl E.; BEATTY, Joycelyn I. *Software Requirements: Best Practices and Patterns*. 2. ed. Redmond, WA: Microsoft Press, 2013. ISBN 978-0-7356-7966-5.

WITHALL, Stephen. *Software Requirements Patterns*. Redmond, WA: Microsoft Press, 2007.

WIEGERS, Karl; BEATTY, Joy. *Software Requirements*. 3. ed. Redmond, WA: Microsoft Press, 2013. 672 p.

WANG, Xiao Sophia et al. Demystifying Page Load Performance with WProf. In: *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, 2013.

ZIMMERMANN, Olaf. Microservices Tenets. *Computer Science - Research and Development*, v. 32, n. 3-4, p. 301-310, 16 nov. 2016. Disponível em: https://doi.org/10.1007/s00450-016-0337-0.

ZHANG, Hao et al. "In-Memory Big Data Management and Processing: A Survey." *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 7, 1 July 2015, pp. 1920–1948, https://doi.org/10.1109/tkde.2015.2427795.