TriCache: Providing three-tier caching for time series data in *serverless* healthcare services

Adriano Zavareze Righi
Universidade do Vale do Rio dos Sinos
São Leopoldo, Rio Grande do Sul, BR
contato@adrianorighi.com

Gabriel Souto Fischer
Universidade do Vale do Rio dos Sinos
São Leopoldo, Rio Grande do Sul, BR
gabriel.souto.fischer@gmail.com

Rodrigo da Rosa Righi
Universidade do Vale do Rio dos Sinos
São Leopoldo, Rio Grande do Sul, BR
rrrighi@unisinos.br

Cristiano André da Costa
Universidade do Vale do Rio dos Sinos
São Leopoldo, Rio Grande do Sul, BR
cac@unisinos.br

Alex Roehrs
Universidade do Vale do Rio dos Sinos
São Leopoldo, Rio Grande do Sul, BR
alexr@unisinos.br

ABSTRACT

Healthcare services and IoT, as highlighted by Hu et al. [9], generate enormous volumes of time series data. Using caching in *serverless* functions can significantly reduce latency and improve performance when storing frequently accessed data in memory. Although several approaches offer improvements, such as the use of in-memory caching, data prediction, and distributed systems, none of them fully addresses the need for a robust and efficient system for time series in healthcare, leaving a gap in necessary data availability and optimization. The TriCache model proposes a three-tier caching system to optimize storage and access to time series data in healthcare *serverless* functions, using a combination of memory in the *serverless* function, in-memory cache, and disk storage, in addition to predictive intelligence. The main contribution of the model is the significant reduction in latency and the improvement in the hit rate by efficiently predicting and allocating data across different cache layers. Experiments demonstrated a notable reduction in response time, with a 110 millisecond decrease in the 99th percentile. Additionally, the model performed significantly, achieving a 93% hit rate, compared to the 78% observed in the traditional model.

CCS CONCEPTS

• **Computing methodologies → Distributed computing methodologies**; • **Applied computing**;

KEYWORDS

cloud computing, cache, data time series, response time.

1 INTRODUÇÃO

Com o avanço da tecnologia e o crescimento exponencial dos dados gerados diariamente, o armazenamento e o processamento eficientes dessas informações se tornaram desafios significativos para as empresas. A computação *serverless* oferece benefícios que ajudam as organizações a inovar rapidamente, escalar eficientemente e reduzir custos [2]. Com ela, os desenvolvedores se concentram na criação de aplicativos sem se preocupar com a infraestrutura, liberando-os para serem mais criativos e inovadores [11]. Segundo Jazaeri et al. [10], os desafios na otimização dos recursos de rede de e-saúde incluem lidar com grandes quantidades de dados de dispositivos IoT (Internet of Things) [1] e IoMT (Internet of Medical Things) [4], garantir baixa latência e alta confiabilidade para aplicações críticas de saúde, e gerenciar eficientemente o armazenamento em cache para minimizar o tráfego de rede e maximizar o desempenho [22].

O uso de funções *serverless* para processamento de séries temporais requer otimização do desempenho de acesso aos dados e minimização do consumo de recursos, o que pode ser alcançado pelo uso de cache. É necessário investigar as complexidades técnicas envolvidas, como a seleção adequada das políticas de cache, a garantia da disponibilidade dos dados necessários e o tipo de armazenamento ideal. Assim, propõe-se a seguinte questão: Como otimizar o armazenamento de séries temporais de dados usando cache, garantindo o desempenho de acesso e minimizando o consumo de recursos em funções *serverless*?

Há diversos trabalhos científicos que exploram o cache, seja em funções *serverless* ou por meio de serviços terceiros [3, 6, 7, 10, 12, 14, 19]. A pesquisa sobre modelos de cache de séries temporais revela uma lacuna significativa na compreensão desse campo. Embora os sistemas de cache sejam eficazes no armazenamento de dados frequentemente acessados [21], sua aplicação específica em séries temporais tem sido pouco explorada. Isso é surpreendente, considerando a relevância crescente das séries temporais em áreas como finanças, meteorologia, IoT e saúde. Algoritmos preditivos são fundamentais para otimizar o desempenho de sistemas de séries temporais, permitindo prever tendências e melhorar a experiência do usuário. No entanto, a integração eficaz entre sistemas de cache e algoritmos preditivos nessas aplicações ainda carece de investigação, abrindo uma lacuna importante no conhecimento atual.

Nesse contexto, esse artigo apresenta o modelo TriCache, o qual utiliza séries temporais de dados de saúde e três camadas distintas de cache. Cada uma dessas camadas desempenha um papel pertinente no gerenciamento eficiente das informações, levando em conta espaço de armazenamento e tempo de resposta. O modelo será gerenciado por uma inteligência preditiva para determinar quais dados devem ser armazenados em cada camada. A principal contribuição do modelo TriCache pode ser vista na sentença abaixo:

- Dado um escopo de aplicações críticas na área da saúde, tem-se a proposição de modelo de cache baseado em séries temporais, predição de dados e arquitetura multi-camada para a execução eficiente de funções *serverless*.

Esse artigo está organizado em 6 seções. Após a Introdução, a Seção 2 descreve os trabalhos relacionados e enfatiza as lacunas existentes no estado-da-arte. A Seção 3 é a principal do artigo,

onde apresenta-se o modelo TriCache. A metodologia de avaliação do referido modelo está na Seção 4, enquanto que os resultados e a discussão estão apresentadas na Seção 5. Por fim, a Seção 6 apresenta as considerações finais, enfatizando como a contribuição foi atingida e tecendo trabalhos futuros para a continuidade dessa pesquisa.

2 TRABALHOS RELACIONADOS

Nesse seção serão apresentados os trabalhos relacionados ao tema deste estudo. O objetivo é fornecer uma visão geral da literatura existente sobre o assunto e destacar as principais contribuições e limitações dos estudos relevantes de fontes como IEEE, Springer, Google Scholar e ACM. A busca fez uso das condições ((*serverless* OR *function as a service*) AND *cache* AND *time series data*) como filtro, além da restrição de período iniciando em janeiro de 2020 até abril de 2023, resultando em 93 artigos. A partir dos resultados encontrados, foram aplicados critérios de inclusão, sendo eles a leitura dos títulos e resumos, buscando as palavras chave relacionadas ao presente trabalho e a posterior leitura da publicação completa para compreensão total. Por fim, foram selecionados 8 trabalhos que tratam do assunto, explorados a seguir.

2.1 Estado da arte

Ghosh et al. [6] comparam o uso de ambiente *serverless* e VM, medindo o tempo de resposta em aplicações de cadastro de usuários e pipelines de dados. Funções *serverless* apresentaram tempo de resposta 14 vezes maior devido à latência com o banco de dados. Eles propõem o uso de cache em memória, resultando em uma redução de 45ms, mas enfrentam limitações de consistência de dados. Göksel e Ovatman [7] desenvolveram uma solução de cache preditivo em máquinas de estado distribuídas. O algoritmo usa históricos de execução para decidir a pré-busca de dados, reduzindo falhas de cache. A quantidade de dados armazenados é baseada em um limite definido pelo número de vezes que determinados caminhos são seguidos.

Wang et al. [19] propõem o InfiniCache, que utiliza a memória das funções *serverless* para armazenamento de cache de baixo custo. A estratégia de estimativas probabilísticas determina quais dados manter ou descartar, resultando em uma alta taxa de acerto. Contudo, a natureza efêmera das funções *serverless* pode levar à perda de dados armazenados. Romero et al. [14] propõem o Faa$t, um sistema de cache distribuído para aplicações *serverless*, visando reduzir latência e melhorar a escalabilidade. Cada função possui seu próprio cache, com metadados históricos ajudando a pré-carregar objetos frequentemente acessados. Essa estratégia é executada quando a função não está em operação, otimizando o uso de recursos.

Boza et al. [3] identificaram problemas de particionamento estático e concorrência em sistemas de cache como o Redis. Eles desenvolveram o SPREDS, que usa funções *serverless* para monitorar a carga de trabalho e ajustar dinamicamente o particionamento de memória. A solução otimiza as partições de cache de acordo com um plano adaptativo gerado por amostras de uso. Tang e Yang [18] propõem o Lambdata, que introduz intenções de dados para otimizar o armazenamento em cache de funções *serverless*. O compilador extrai intenções de dados, o planejador otimiza a execução com base nessas intenções e o executor realiza a execução otimizada. Isso melhora a eficiência do acesso a dados e a execução das funções.

Li et al. [12] desenvolveram o Predictive Edge Caching (PEC) para prever a popularidade futura de conteúdos e pré-buscá-los, usando modelos de aprendizado. O sistema híbrido de cache reduz a latência em até 53,23%, especialmente fora dos horários de pico. A predição é baseada em comportamento histórico dos usuários e cálculos estatísticos. Jazaeri et al. [10] propõem um método de agrupamento para cache eficiente em tecnologias IoT na saúde, usando um controlador SDN. O controlador gerencia o cache baseado em critérios como tipo de dado e condições da rede, removendo dados antigos periodicamente. O algoritmo MFO-Edge agrupa dados de sensores médicos para otimização de cache.

2.2 Análise e oportunidades

Os artigos apresentados foram analisados com base nos critérios apresentados a seguir.

- Cache em cloud - serviços terceiros (C1): armazenamento temporário de dados frequentemente acessados em servidores localizados na nuvem, usando serviços como Redis, ElastiCache, Memorystore, etc.

- Cache em cloud - memória na função *serverless* (C2): é o armazenamento temporário de dados frequentemente acessados em servidores localizados na nuvem, usando memória da própria função *serverless*.

- Cache em cloud - disco na função *serverless* (C3): trata do armazenamento temporário de dados frequentemente acessados em servidores localizados na nuvem, usando arquivo escrito em diretório na própria função *serverless*.

- Cache em nós de borda (C4): trata de armazenar temporariamente dados em servidores localizados próximos aos dispositivos de usuários finais, nos chamados nós de borda ou *edge nodes*.

- Escalabilidade (C5): refere-se à capacidade de expandir ou ajustar o sistema de cache de forma a lidar com aumentos na demanda por dados e recursos.

- Performance (C6): refere-se à eficiência e velocidade com que os dados são armazenados em cache e recuperados a partir dele.

- Predição (C7): diz respeito a técnicas que permitem estimar quais dados ou recursos serão necessários em futuras execuções de uma função *serverless* e armazená-los em cache antecipadamente.

- Adaptação (C8): é a capacidade de ajustar dinamicamente as políticas de cache com base nas condições e demandas em tempo real, levando em consideração fatores como padrões de acesso, tamanho do cache disponível, características da carga de trabalho e metas de desempenho.

A Tabela 1 a seguir, apresenta uma comparação dos trabalhos com relação aos critérios definidos.

Em análise a tabela comparativa acima (Tabela 1), é possível identificar que os esforços se concentram muito na performance dos sistemas de cache e *caching* em memória. Esses esforços refletem uma importância dessas tecnologias com a finalidade de acelerar o acesso a dados, sejam em nós de borda ou na cloud, e melhorar a eficiência das funções *serverless*.

A pesquisa atual sobre modelos de cache de séries temporais revela uma lacuna notável na compreensão desse campo. Embora os sistemas de cache tenham se mostrado eficazes no armazenamento

Tabela 1: Tabela comparativa dos trabalhos selecionados

Artigo	C1	C2	C3	C4	C5	C6	C7	C8
[6]	S	S	N	N	N	S	N	N
[7]	N	S	N	N	S	S	S	N
[19]	N	S	N	N	N	S	S	N
[14]	N	S	N	N	N	S	S	N
[3]	S	N	N	N	S	S	S	S
[18]	N	S	N	N	S	S	S	S
[12]	N	N	N	S	N	S	S	S
[10]	N	N	N	S	S	S	S	S

de dados frequentemente acessados, sua aplicação específica em séries temporais tem sido pouco explorada. Isso é surpreendente, considerando a crescente relevância das séries temporais em áreas como finanças, meteorologia, IoT e saúde. Além disso, os algoritmos preditivos desempenham um papel fundamental na otimização do desempenho de sistemas de séries temporais, permitindo prever tendências futuras e melhorar a experiência do usuário. No entanto, a integração eficaz entre sistemas de cache e algoritmos preditivos nessas aplicações ainda carece de investigação adequada, abrindo uma lacuna importante no conhecimento atual.

3 MODELO TRICACHE

Nessa seção, será apresentado um modelo de solução de cache de séries temporais de dados em três camadas, chamado TriCache, que busca otimizar o desempenho de sistemas por meio do armazenamento de dados em diferentes níveis de cache. As decisões de projeto são abordadas, levando em consideração aspectos como prioridade de acesso aos dados, capacidade de armazenamento, políticas de substituição de dados e estratégias de comunicação entre as camadas. A arquitetura do modelo é detalhada, destacando as diferentes camadas de cache e como elas se interconectam para formar um sistema robusto e escalável.

A abordagem moderna para o desenvolvimento de sistemas permite que os desenvolvedores foquem apenas na lógica de negócio, minimizando as preocupações com infraestrutura e custos. Essa abordagem tem ganhado força nos últimos tempos, especialmente na área da saúde, onde é possível oferecer serviços inovadores e escaláveis com menor investimento [13]. Nesse contexto, definimos o escopo e as premissas para a implementação de uma arquitetura utilizando a infraestrutura de uma provedora de serviços em nuvem, escolhendo serviços específicos para execução de funções, banco de dados relacional e cache em memória, além do uso de um algoritmo de regressão linear para aprendizado de máquina.

Por fim, ao interpretar a arquitetura e o modelo propostos, o foco está em melhorar a velocidade no acesso a dados para o processamento síncrono de informações que utilizam séries temporais em ambiente de nuvem pública. A estratégia de intercomunicação adotada utiliza um formato padrão para troca de informações através de um protocolo amplamente aceito, facilitando a troca de dados entre os componentes e promovendo uma integração harmoniosa.

3.1 Decisões de projeto

Nesse contexto, iremos definir o escopo e as premissas para a implementação de uma arquitetura *serverless* usando a infraestrutura de cloud pública disponibilizada pela Amazon Web Services (AWS). Essa decisão tem como base o quadrante mágico da Gartner [5], que é uma metodologia para avaliar e posicionar empresas em diferentes setores de tecnologia, a qual apresenta a Amazon Web Services como líder no setor. A partir da escolha do fornecedor, definiu-se como serviços da plataforma, usados para o desenvolvimento do modelo de solução, o serviço de função *serverless* sendo o AWS Lambda e banco de dados relacional RDS - PostgreSQL. Outro serviço usado para cache em memória foi o Redis, executando em um cluster de Kubernetes, com três nós, na própria AWS.

Também, como parte do modelo proposto, optou-se pelo uso do algoritmo de regressão linear que traz bastante relevância e simplicidade na aplicação de *machine learning* para a solução do problema. A abordagem simples e a interpretabilidade da regressão linear, permitem a modelagem da relação linear entre as variáveis relevantes, fornecendo uma base para estimativas de valores futuros com base nos padrões identificados nos dados históricos. No que tange as funções Lambda, deve-se usar a linguagem Python por ser uma linguagem de programação amplamente adotada e possui uma vasta comunidade de desenvolvedores, o que significa que existe uma ampla gama de recursos, bibliotecas e *frameworks* disponíveis para facilitar o desenvolvimento e a manutenção das funções Lambda.

3.2 Arquitetura

A arquitetura do modelo TriCache proposto consiste no uso de cache em três camadas, sendo cada camada um módulo, a partir de uma estrutura hierárquica de armazenamento e consumo de dados, além do módulo de predição. Como base, usou-se o conceito arquitetural do DNS (*Domain Name System*) recursivo, descrito por Stanevsic et al. [16]. Em sua abordagem apresenta que o DNS recursivo é um tipo de consulta DNS em que o servidor, ao receber uma solicitação de um cliente para um domínio que não está em sua zona de autoridade, consulta outros servidores DNS em nome do cliente até encontrar a resposta correta. O servidor DNS recursivo, por sua vez, armazena em cache as respostas para consultas futuras, o que pode melhorar significativamente o desempenho e a eficiência do sistema.

Assim, a arquitetura TriCache é composta, na primeira camada, por uma função *serverless*, na segunda camada por um serviço de cache em memória e, na terceira camada, por um banco de dados relacional como cache em disco. Essa abordagem visa melhorar o desempenho e a eficiência de um sistema, minimizando a latência ao acessar dados frequentemente utilizados. A camada superior dessa arquitetura será a função *serverless*, executando seu código diretamente no ambiente de computação sem servidor. Essa função tem a capacidade de armazenar dados temporários em sua própria memória, fornecendo acesso de baixa latência a esses dados. É uma camada ideal para armazenar dados que são frequentemente acessados durante a execução da função, evitando consultas adicionais a camadas de cache externas ou ao banco de dados.

A segunda camada será o cache em memória provido por um sistema de armazenamento chave-valor altamente escalável e de alto desempenho. O serviço dessa camada será usado para armazenar dados em cache de forma persistente e deve oferecer uma ampla gama de recursos, como chaves personalizadas expiráveis,

estruturas de dados complexas e suporte a consultas por índices. Os dados armazenados podem ser acessados rapidamente, reduzindo a carga no banco de dados na camada subjacente. A terceira camada será o banco de dados em disco, responsável por armazenar os dados de forma persistente. Essa camada oferece durabilidade e recuperação de dados em caso de falhas ou reinicializações do sistema, além de escalabilidade necessária para grandes volumes de dados. Ele é projetado para armazenamento de forma eficiente e confiável, embora tenha um tempo de acesso maior em comparação com as camadas anteriores.

Além das três camadas de armazenamento, a arquitetura também inclui uma função de predição de cache *serverless*, que utiliza um algoritmo de regressão linear. A predição de cache é executada para antecipar quais dados serão necessários no futuro. O motor de predição analisa uma série de elementos históricos, como a frequência de acessos a determinados dados, padrões de uso ao longo do tempo, e a variação sazonal desses acessos. Esses dados históricos são utilizados como variáveis independentes na regressão linear. A partir dessa análise, o algoritmo projeta a quantidade e os tipos de dados que provavelmente serão acessados em um período futuro, permitindo que o sistema pré-carregue os dados previstos nas camadas de cache apropriadas.

Em arquiteturas atuais, que fazem o uso de cache em funções *serverless*, é bastante comum encontrar apenas um serviço como recurso de cache em memória. Dessa forma, antes de consultar um banco de dados persistente, é feita uma consulta ao cache. A Figura 1 demonstra essa arquitetura, normalmente usada, representada pelo diagrama do modelo tradicional. O modelo TriCache, representado ao lado na mesma figura, evidencia a arquitetura proposta para a solução conforme suas camada e serviços específicos. Com a finalidade de reduzir ainda mais a latência e melhorar o desempenho, é importante que todos os serviços estejam se comunicando internamente em uma mesma rede.

Em resumo, a arquitetura do cache em três camadas combina a velocidade de acesso à memória na função *serverless*, o desempenho e a flexibilidade do cache em memória e a durabilidade do cache em disco para criar uma estrutura eficiente de armazenamento de dados. A função de predição complementa a arquitetura, permitindo uma alocação otimizada de recursos de cache com base nos padrões de acesso aos dados, em cada uma das camadas. Essa abordagem pode melhorar o desempenho e a eficiência de um sistema, reduzindo a latência e minimizando a carga no banco de dados.

3.3 Funcionamento

Como base para implementação e funcionamento do modelo TriCache, é necessário a definição da estrutura de dados das séries temporais a serem processadas. É importante que os dados sejam enviados contendo o *timestamp* da leitura das informações de monitoramento e o identificador único do usuário monitorado, neste caso a propriedade *userId*. Essa estrutura de dados está exemplificada no pseudo-código JSON apresentado na Figura 2 a seguir.

No modelo TriCache, o fluxo de dados começa com a captura de dados de séries temporais através de dispositivos de monitoramento de saúde, que enviam essas informações em formato JSON para a função serverless na camada 1. Esta função realiza uma verificação inicial para determinar se os dados estão presentes em sua memória. Se não forem encontrados, a função faz uma chamada síncrona ao cache em memória na camada 2. Caso os dados ainda não estejam disponíveis, a busca se estende ao banco de dados relacional na camada 3. Uma vez localizados, os dados são retornados para a camada solicitante e, se necessário, armazenados nas camadas superiores para acesso mais rápido em futuras requisições. Além disso, o módulo de predição analisa continuamente os padrões de acesso e utiliza algoritmos de regressão linear para prever quais dados serão necessários, pré-carregando-os nas camadas de cache apropriadas. Na Figura 3 está um exemplo de pseudo-código que ilustra esse fluxo de dados.

A partir desse ponto, procede-se com a implementação das camadas do modelo, as quais são organizadas em quatro módulos distintos. Desses, três são encarregados de gerenciar os níveis de cache, enquanto o quarto módulo assume a responsabilidade de efetuar previsões e disponibilizar os dados essenciais. Os dados de séries temporais, que são disponibilizados na primeira camada pelo módulo de predição, serão armazenados em memória e preservados por um período prolongado na segunda camada. Essa solução de armazenamento garante que as informações temporais permaneçam acessíveis e disponíveis por um período de tempo estendido, permitindo um histórico mais completo e um funcionamento contínuo do sistema.

Além disso, os dados serão persistidos em disco na terceira camada do sistema, utilizando um banco de dados relacional. O armazenamento se dará de forma que uma coluna seja a chave para o registro e a outra seja o JSON com os dados de monitoramento, da mesma forma que os demais armazenamentos. Diferentemente das camadas anteriores, nessa etapa os dados não possuirão um tempo de expiração definido. Portanto, caso não sejam encontrados nas camadas anteriores, eles serão buscados nessa última camada, garantindo que as informações estejam sempre acessíveis e disponíveis para consulta. O módulo de predição consumirá dados históricos de processamento para gerar as necessidades de cache. Posterior a essa identificação, serão feitas as buscas dos dados para consumo, e armazenadas em memória na própria função de cache por um curto espaço de tempo devido a limitações de recursos, na primeira camada.

Como estratégia, os dados serão distribuídos em cada camada obtendo-se o total de registros únicos consultados e dividindo-se pelo número de camadas de cache, que são 3 (três). Dessa forma, os dados são distribuídos entre as camadas de cache de maneira estruturada. A primeira parte dos dados, representada pela variável "camada1"na Figura 4, é armazenada na primeira camada, que corresponde à memória da função serverless. Esta camada é usada para dados que requerem acesso mais rápido e são frequentemente consultados. A segunda parte dos registros, representada pela variável "camada2"na mesma figura, é direcionada para a segunda camada, que utiliza um serviço de cache em memória, como o Redis. Esta camada oferece um equilíbrio entre velocidade de acesso e capacidade de armazenamento. Finalmente, a terceira parte dos dados, representada pela variável "camada3"na figura, é enviada para a terceira camada, que é um banco de dados relacional. Esta última camada armazena os dados de forma persistente e é utilizada para consultas menos frequentes ou dados históricos que não precisam de acesso imediato. Esse esquema de distribuição garante que os

Figura 1: Comparação entre as arquiteturas envolvidas no modelo tradicional e no modelo TriCache

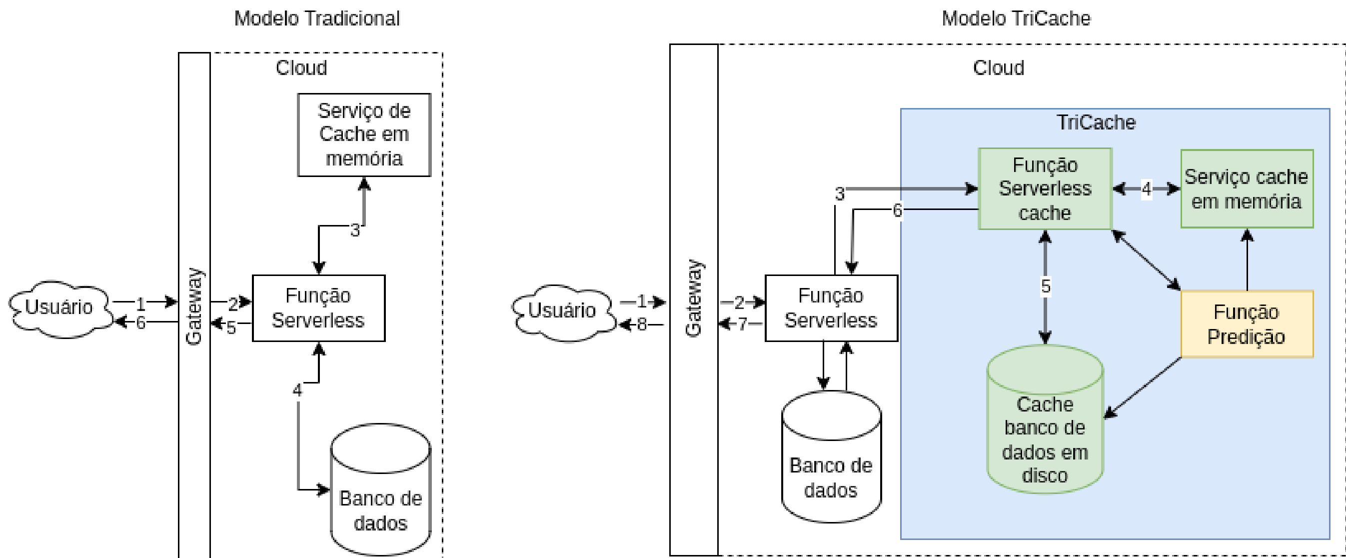


Figura 2: Pseudo-código JSON usado para envio das informações para a primeira camada do cache

```
{
  "source": "wereable-device",
  "vital_signs": {
    "heartbeat": 68,
    "spo2": 99,
    "temperature": 36.5,
    "blood_pressure": "175/80"
  },
  "timestamp": 1695520195,
  "userId": "3df4f9e3-f9e8-4cc1-9b18-7a8893a14838"
}
```

Figura 3: Pseudo-código de fluxo de cache nas camadas

```
# Função principal da camada 1

def funcao_principal(request):
    dados = buscar_cache_camada1(request)
    if not dados:
        dados = buscar_cache_camada2(request)
        if not dados:
            dados = buscar_cache_camada3(request)
            if not dados:
                dados = buscar_banco_de_dados(request)
                armazenar_cache_camada3(dados)
        armazenar_cache_camada2(dados)
    armazenar_cache_camada1(dados)
    return processar_dados(dados)

# Função de predição

def funcao_predicao():
    dados_historicos = coletar_dados_historicos()
    previsao = algoritmo_regressao_linear(dados_historicos)
    distribuir_dados(previsao)
```

Figura 4: Pseudo-código de distribuição de cache nas camadas

```
total_registros = len(registros_ordenados)
ponto_de_corte = total_registros//3

camada1 = registros_ordenados[:ponto_de_corte]
camada2 = registros_ordenados[ponto_de_corte:2 * ponto_de_corte]
camada3 = registros_ordenados[2 * ponto_de_corte:]
```

dados mais relevantes e frequentemente acessados estejam disponíveis rapidamente, enquanto dados menos críticos são armazenados de forma eficiente nas camadas subsequentes.

Com base nessas definições do funcionamento das camadas, estabelecemos o procedimento da função principal, implementada na função *serverless*, responsável pelo processamento das informações. Ao receber os dados contendo o período e a identificação dos dados a serem processados, ela inicia a busca em cache de forma síncrona. O primeiro passo consiste em consultar a função *serverless* de cache da primeira camada para verificar se os dados estão armazenados em memória. Caso estejam disponíveis, a função retorna os dados para o processamento necessário na função principal, que retorna a resposta correspondente. No entanto, se os dados não forem encontrados na camada de cache primária, a busca é estendida para a próxima camada, também de forma síncrona pela própria função de cache da primeira camada. Se os dados forem encontrados nessa camada, são devolvidos até a função onde o processamento é realizado e a resposta é enviada como resultado da requisição. Por fim, se os dados ainda não forem encontrados na camada anterior, a busca se estende para a terceira e última camada pela função *serverless* de cache, onde os dados podem estar acessíveis para consulta. Por fim, no caso de nenhuma das três camadas possuírem os dados necessários, é retornado para a função principal que a informação não está disponível, ficando de responsabilidade dessa função a

Figura 5: Fluxo de decisão para busca da informação no cache

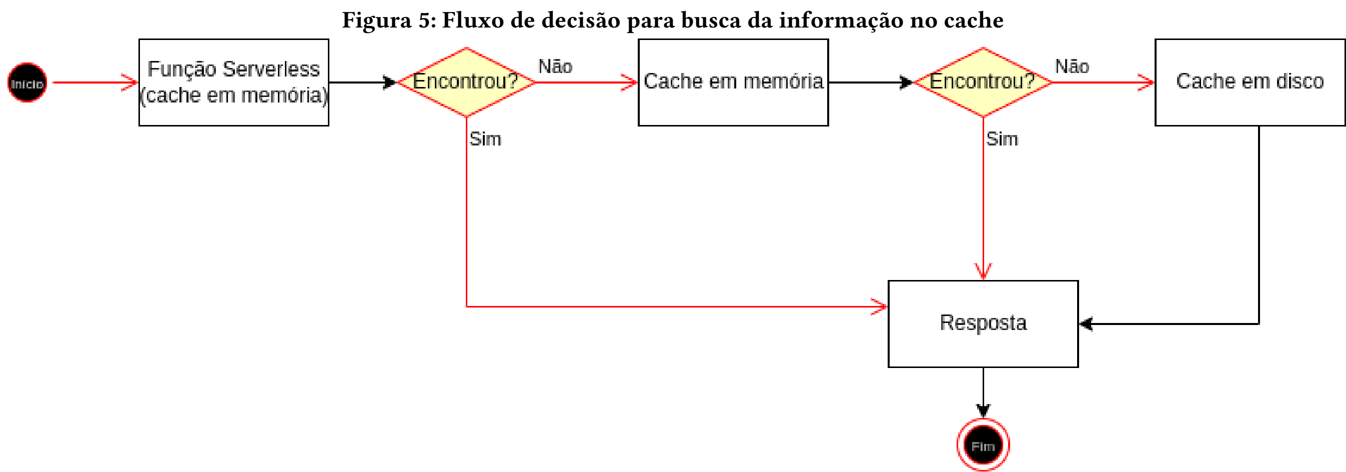
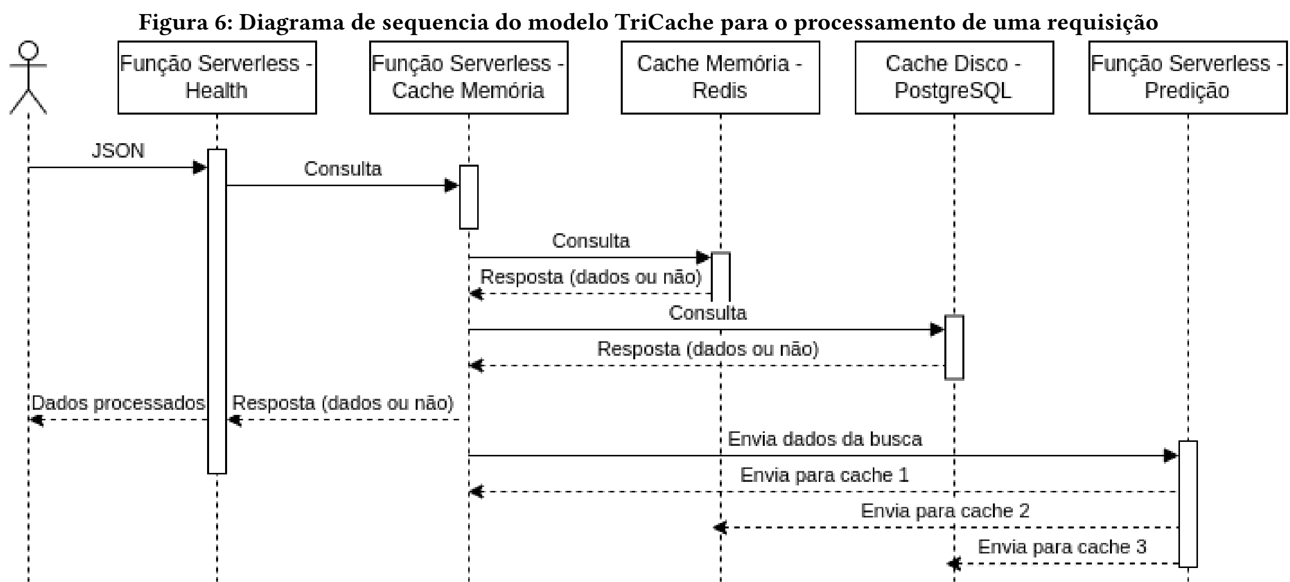


Figura 6: Diagrama de sequencia do modelo TriCache para o processamento de uma requisição



busca dos dados no local são persistidos em definitivo. A Figura 5 demonstra o fluxo de decisão para busca da informação no cache, com base no funcionamento descrito anteriormente.

O diagrama de sequência, apresentado na Figura 6, descreve o fluxo de processamento de uma solicitação de dados de séries temporais. A solicitação é recebida pela função *health*, que primeiro consulta a função de cache da primeira camada. Se não houver registros nessa camada, a consulta é estendida para a segunda camada. Caso não encontre dados na segunda camada, a função *serverless* da primeira camada consulta o banco de dados da terceira camada, onde os dados são persistidos em disco. Além disso, a função de cache primário deve submeter as informações de cada requisição para a função de predição, para que os dados sejam reprocessados e disponibilizados conforme o novo resultado de predição.

Um aspecto crucial da eficiência do TriCache é a política de substituição de cache utilizada para garantir que os dados mais relevantes estejam disponíveis nas camadas de cache de acesso mais rápido. A política adotada é baseada na estratégia de substituição do item mais antigo (*Least Recently Used* - LRU). Esse mecanismo funciona monitoramento o tempo de acesso aos dados e substituindo os dados que não foram acessados por mais tempo quando novos precisam ser inseridos no cache. Esse método é eficaz em ambientes de séries temporais, onde o padrão de acesso aos dados pode ser altamente dinâmico e variável. Além disso, a adoção da política LRU no TriCache garante que os dados mais frequentemente acessados permaneçam disponíveis na camada de acesso mais rápido, resultando em uma redução da latência e uma melhora na taxa de acerto, o que é crítico para aplicações em saúde que demandam respostas rápidas e confiáveis.

4 METODOLOGIA DE AVALIAÇÃO

Nessa seção será realizada uma avaliação do modelo proposto. Para atingir esse objetivo, elaboraremos um protótipo que servirá como base para a implementação dos cenários descritos, possibilitando assim a coleta das métricas definidas como parte do processo de avaliação.

4.1 Protótipo

A partir das diretrizes apresentadas na seção 4, foi desenvolvida uma função Lambda *health*. Seu propósito é receber requisições externas que estejam estruturadas conforme o formato JSON especificado na seção 4.3. Essa função foi construída usando a linguagem de programação Python e será executada em um ambiente *serverless* na AWS Lambda versão 3.12.0[1]. Além disso, cada módulo que compõe o modelo TriCache será construído utilizando a versão 3.10 da linguagem Python. Essa escolha se justifica em grande parte devido à natureza de código aberto (*open-source*) da linguagem, que oferece flexibilidade e uma vasta gama de bibliotecas que simplificam o desenvolvimento e a implementação dos módulos. A comunicação entre esses módulos será realizada por meio de requisições HTTP, permitindo uma integração eficaz.

Quando se trata de armazenamento de dados temporais, o PostgreSQL destaca-se como uma escolha robusta. Ele ganha destaque especialmente na preservação de dados biológicos, graças à sua sólida verificação de tipos de dados e ao sistema de restrições baseado em regras, que assegura a conformidade com ACID, consolidando assim sua posição como uma opção prevalente nesse contexto Sanderson et al. (2021) [15]. Além disso, aprofunda-se na comparação entre um banco de dados PostgreSQL convencional e um banco de dados híbrido para armazenamento de dados geoespaciais, revelando que o PostgreSQL apresenta um tempo de resposta mais curto em comparação com o banco de dados híbrido Herrera-Ramírez et al. (2021) [8]. Essas descobertas destacam a eficácia do PostgreSQL na gestão e armazenamento de dados de séries temporais, tornando-o uma escolha convincente em relação a outras alternativas, como o InfluxDB.

É importante notar que os módulos também serão implantados em ambientes *serverless*, aproveitando a infraestrutura flexível fornecida no serviço AWS Lambda. No contexto do módulo de predição, optou-se pelo uso do algoritmo de Regressão Linear. Esta escolha se baseia na ampla aceitação desse algoritmo, que é conhecido por sua simplicidade e eficiência computacional. A Regressão Linear é uma abordagem robusta e confiável para tarefas de previsão, tornando-a uma boa escolha para a implementação do módulo de predição no âmbito do modelo TriCache.

4.2 Cenários de avaliação

Os cenários propostos a seguir tem como objetivo avaliar a eficácia do modelo em situações de uso por aplicativos de monitoramento de saúde. Eles fornecerão resultados que demonstrarão o desempenho do sistema em termos de consumo de memória, escalabilidade, latência e taxa de acerto. Esses resultados permitirão avaliar a capacidade do modelo de lidar com cargas de trabalho variadas, garantir respostas rápidas, manter baixa latência e otimizar o uso de recursos, fornecendo *insights* críticos sobre sua adequação para aplicações práticas.

[1]https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html.

A frequência de medição é um fator importante a ser considerado em cenários de avaliação de funções *serverless* usadas para monitoramento de saúde. De acordo com os artigos de Xing et al. (2022) [20] e Zhang et al. (2021) [23], a frequência de medição varia de acordo com o tipo de sinal vital sendo monitorado e as necessidades do usuário. Para sinais vitais que precisam ser monitorados em tempo real, como a frequência cardíaca, a frequência de medição deve ser alta, de 1 a 2 vezes por minuto. Isso permite que mudanças sutis nos sinais vitais sejam detectadas rapidamente. Para sinais vitais que não precisam ser monitorados em tempo real, como a pressão arterial, a frequência de medição pode ser mais baixa, de 1 a 5 vezes por hora. Com base nisso, será usada a frequência de monitoramento de 2 vezes por minuto nos cenários.

- Cenário A - Modelo TriCache: Testar o desempenho do modelo TriCache em um consumo normal de monitoramento de 50 usuários durante 5 minutos. Isso será feito gerando um conjunto de dados de entrada de sinais vitais simulado com diferentes informações e frequências de acesso.

- Cenário B - Modelo tradicional e Modelo TriCache: Comparar o TriCache com o modelo tradicional de cache *serverless*, usando apenas uma camada. Isso será feito executando o modelo TriCache e o modelo de cache tradicional em um conjunto de dados de entrada de sinais vitais comum.

4.3 Métricas de avaliação

Esse capítulo aborda as métricas fundamentais de avaliação de um modelo de cache, explorando aspectos essenciais para seu desempenho e escalabilidade. Para Sun et al. (2023) [17] a escolha das métricas de avaliação de cache a serem usadas depende dos objetivos específicos do sistema. No entanto, as métricas fundamentais, como taxa de acerto, tempo de acesso e uso de memória, devem ser sempre consideradas. A seguir, estão os parâmetros levados em consideração para análise do modelo TriCache.

- Tempo de acesso: Avaliar o tempo decorrido entre a emissão da requisição para busca do cache e o recebimento da resposta. Um tempo de acesso baixo é essencial para garantir o funcionamento ideal de serviços críticos.

- Uso de memória: Avaliar o consumo de memória do sistema em diferentes momentos de carga de trabalho e cenários. Isso envolve a análise detalhada do uso de memória em cada camada, assegurando simultaneamente que o consumo esteja dentro dos limites aceitáveis.

- Taxa de acerto: Também é necessário avaliar a taxa de acerto do algoritmo de predição, evitando consumo de recursos desnecessários, o que também reduz o tempo de resposta da função *serverless*.

5 RESULTADOS

Essa seção compreende a apresentação dos resultados alcançados mediante a implementação do protótipo descrito no modelo, conforme discutido nas seções anteriores, e também no ambiente de testes previamente elaborado. Além disso, as análises serão fundamentadas nos cenários descritos no capítulo anterior e a sua simulação será realizada através da ferramenta *Locust* e, para a coleta de informações de consumo de recursos, utilizou o AWS

CloudWatch para funções Lambda e o Prometheus e Grafana para os serviços de Redis e PostgreSQL.

5.1 Cenário A - Modelo TriCache

A partir da execução do cenário A, analisaremos o desempenho do modelo TriCache sob condições normais de consumo, simulando o monitoramento de 50 usuários ao longo de 5 minutos. Para isso, geramos um conjunto de dados simulados incorporando diversas informações e frequências de acesso. Durante a execução do Cenário A, observamos um total de 85.949 pedidos processados. Dentre estes, 5.204 resultaram em não encontrados nas camadas de cache, indicando uma taxa de falha de 6,05%. Contrapondo essa métrica, o modelo apresentou uma notável taxa de acerto de 93,95%. As métricas de desempenho revelaram-se consistentes e promissoras. A taxa de falha de 6,05% sugere uma resistência notável em lidar com pedidos, enquanto a taxa de acerto de 93,95% ressalta a eficácia do modelo em atender as demandas dos usuários.

Analisando os percentis de 90 e 99, observamos tempos de resposta de 180 milissegundos e 230 milissegundos, respectivamente. Esses valores indicam uma consistência e eficiência notáveis do modelo TriCache em cenários de consumo normais. Além disso, a distribuição de registros entre as camadas foi calculada, revelando uma alocação equitativa de recursos. Aproximadamente 21,69% dos registros foram atribuídos à camada 1, 42,17% à camada 2 e 36,14% à camada 3, indicando um balanceamento eficiente durante a simulação de consumo. Além disso, a Figura 7 demonstra o consumo de memória da função de cache (camada 1), o que demonstrou variar entre 80MB e 300MB, de 1024MB disponíveis. A camada 2, representada pelo Redis, e a camada 3, representada pelo PostgreSQL, apresentaram um consumo médio de memória de 113 MiB e 39 MiB, respectivamente, conforme demonstrado na Figura 8. Em contraste, a função *serverless* de predição, obteve um consumo de memória mais alto, chegando a 2,5GB conforme apresentado na Figura 9, e com uma duração de até 32s de execução, sendo maior que os citados anteriormente.

5.2 Cenário B - Modelo Tradicional e Modelo TriCache

Nesse cenário comparativo, analisamos o desempenho de dois modelos distintos de cache, nomeadamente o TriCache e o modelo tradicional baseado em uma camada, sendo o Redis. Ambos os modelos foram avaliados utilizando um conjunto de dados de entrada comum. No contexto do modelo TriCache, observamos um total de 85.949 requisições processadas, com uma taxa de falha de 6,05% e uma notável taxa de acerto de 93,95%. Os percentis de 90 e 99 registraram tempos de resposta eficientes de 180 ms e 230 ms, respectivamente. Além disso, a distribuição equitativa de registros entre as camadas - aproximadamente 21,69% para a camada 1, 42,17% para a camada 2 e 36,14% para a camada 3 - destaca uma eficiente alocação de recursos.

Contrastando com o TriCache, o modelo tradicional processou um total de 77.566 requisições, apresentando uma taxa de falha significativamente mais elevada de 21,08%. A taxa de acerto foi de 78,92%, e os percentis de 90 e 99 resultaram em tempos de resposta de 190 ms e 340 ms, respectivamente. A comparação revela diferenças substanciais em termos de desempenho. O TriCache demonstrou uma superioridade, apresentando uma taxa de acerto mais elevada, uma taxa de falha menor e tempos de resposta mais rápidos quando comparado ao modelo tradicional. A distribuição equitativa de registros nas camadas do TriCache também ressalta uma gestão eficaz de recursos, contribuindo para a sua eficiência global. Em relação a escalabilidade, é possível identificar que o modelo TriCache possui uma taxa de aproximadamente 10% maior de requisições recebidas.

5.3 Avaliação qualitativa do TriCache em relação aos trabalhos relacionados

O modelo TriCache se destaca no cenário de cache para séries temporais em ambientes *serverless* ao introduzir uma arquitetura de três camadas combinada com predição de dados utilizando algoritmos de regressão linear. Em comparação com trabalhos relacionados, como os de Ghosh et al. [4] e Wang et al. [16], que focam principalmente no uso de cache em memória para reduzir a latência, o TriCache oferece uma abordagem mais robusta e escalável.

Desempenho e Eficiência: Ghosh et al. [4] propõem o uso de cache em memória para funções serverless, resultando em uma redução de 45ms na latência, mas enfrentam problemas de consistência de dados. Em contraste, o TriCache não só reduz significativamente a latência (com uma redução de até 230ms no 99º percentil), mas também melhora a taxa de acerto para 93,95%, superior aos 78% observados em modelos tradicionais. Isso é alcançado pela combinação de três camadas de cache e a predição de dados, que otimiza a alocação de recursos e garante a disponibilidade de dados frequentemente acessados.

Escalabilidade e Resiliência: Trabalhos como o de Göksel e Ovatman [5] e Romero et al. [11] exploram soluções de cache preditivo e distribuído para melhorar a escalabilidade e a eficiência. No entanto, o modelo TriCache vai além ao utilizar um motor de predição baseado em regressão linear que analisa padrões históricos e prevê necessidades futuras, permitindo uma distribuição inteligente dos dados entre as camadas. Isso não apenas melhora a eficiência do acesso, mas também assegura que o sistema possa escalar de forma dinâmica para atender a variações na carga de trabalho.

Flexibilidade e Aplicabilidade: Wang et al. [16] apresentam o InfiniCache, que utiliza funções *serverless* para um cache de baixo custo, mas enfrenta desafios devido à natureza efêmera dessas funções. O TriCache, por sua vez, mitiga esse problema ao integrar uma função de predição que distribui dados de maneira eficiente entre a memória volátil das funções *serverless*, cache em memória e armazenamento persistente em banco de dados relacional. Essa abordagem híbrida garante uma maior durabilidade dos dados e reduz a dependência de funções *serverless* voláteis, tornando o modelo uma solução mais flexível e aplicável a uma variedade de cenários, especialmente em ambientes críticos como a saúde.

Predição de Dados: Diferentemente de Boza et al. [2] e Li et al. [10], que utilizam métodos de predição baseados em aprendizado de máquina para otimizar o cache, o TriCache adota uma abordagem de regressão linear para manter a simplicidade e eficiência computacional. Essa escolha permite uma implementação mais direta e menos onerosa, ao mesmo tempo que proporciona melhorias significativas na predição de padrões de acesso e na alocação de dados.

Figura 7: Consumo de memória ao longo da execução dos cenários - Lambda cache (Camada 1)

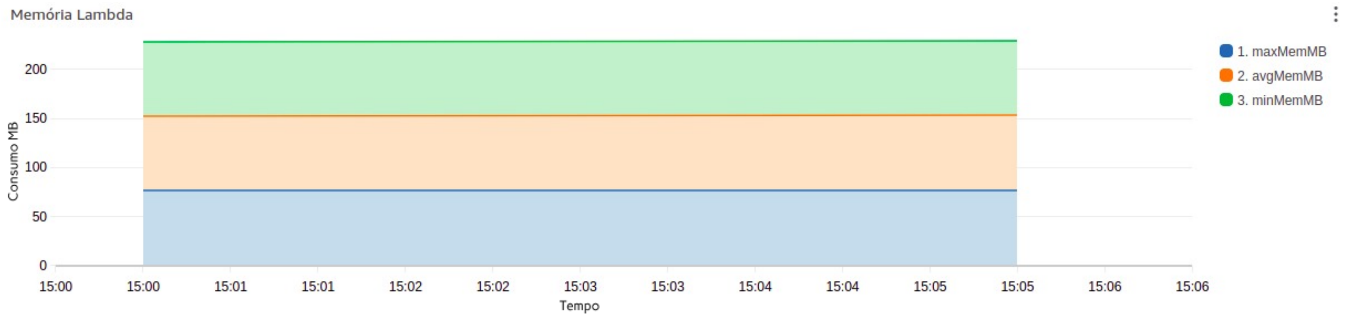


Figura 8: Consumo de memória ao longo da execução dos cenários - Redis (Camada 2) e PostgreSQL (Camada 3)

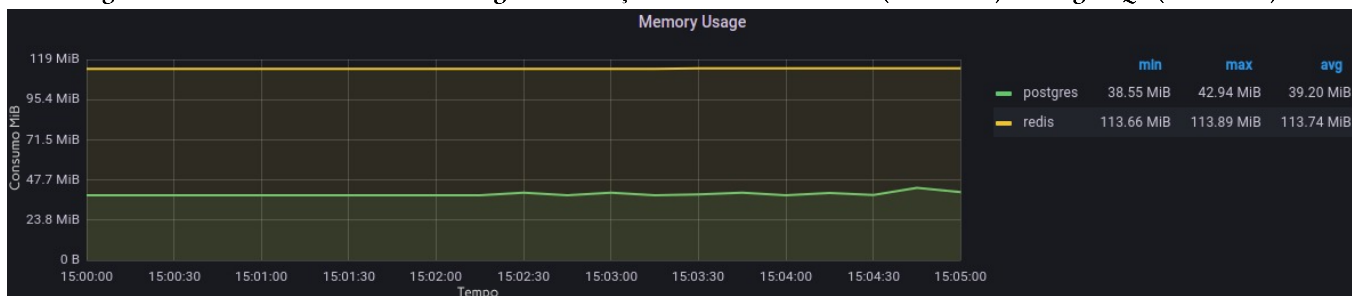
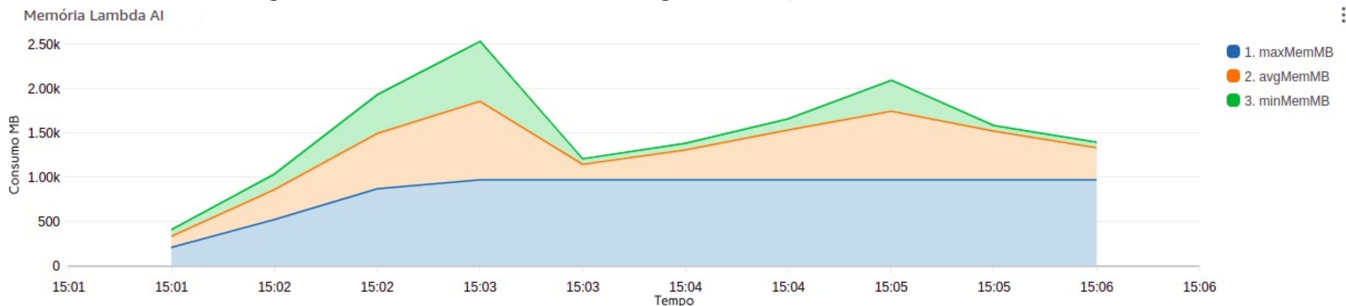


Figura 9: Consumo de memória ao longo da execução dos cenários - Lambda AI



O uso de regressão linear facilita a interpretação dos resultados e a adaptação do modelo a diferentes tipos de cargas de trabalho.

5.4 Limitações

Durante a execução dos testes do modelo TriCache, algumas limitações importantes foram observadas e, podem ter influenciado os resultados. Primeiramente, os testes foram conduzidos em um ambiente de simulação controlado, o que pode não refletir com precisão as condições reais de um ambiente de produção. Em um cenário de produção, variáveis como a variabilidade da carga de trabalho, falhas de rede, e a interação com outros sistemas e serviços podem afetar significativamente o desempenho do modelo. Além disso, a escala dos testes foi limitada a um número fixo de usuários e um período de tempo definido.

Embora os resultados tenham sido promissores, a capacidade do modelo de lidar com um número significativamente maior de usuários e um período de tempo mais prolongado precisa ser avaliada. Futuros trabalhos devem focar em realizar testes de estresse em larga escala e em ambientes de produção para validar a robustez e a escalabilidade do modelo em condições mais diversas e desafiadoras. Outra limitação foi a infraestrutura utilizada para os testes, que pode não ter explorado plenamente os recursos disponíveis em um ambiente de nuvem pública, como técnicas avançadas de balanceamento de carga e recuperação de desastres. Melhorias nesses aspectos podem fornecer uma visão mais abrangente e realista do desempenho do modelo TriCache.

6 CONCLUSÃO

Ao longo deste trabalho, foi explorada a problemática do uso de cache para armazenamento de séries temporais em ambientes *serverless*, com foco em dados de serviços de saúde. O uso de cache é uma solução promissora para otimizar o desempenho e minimizar o consumo de recursos, permitindo armazenar temporariamente resultados de consultas a banco de dados ou operações frequentes, resultando em acesso mais rápido aos dados e redução significativa da carga nos sistemas de armazenamento. Isso é crucial na área da saúde, onde diagnósticos rápidos podem evitar problemas sérios. Foram investigadas as características das séries temporais, funções *serverless*, cache em memória e disco, além das complexidades na seleção das políticas de cache e garantia da disponibilidade dos dados. Os resultados indicam que o modelo TriCache é mais eficiente e robusto em comparação a modelos tradicionais, mostrando uma redução de aproximadamente 32% no tempo de resposta e uma taxa de acerto 15% maior. Esses insights são valiosos para decisões sobre soluções de cache em ambientes variados e sugerem a necessidade de estudos mais aprofundados em cenários complexos para uma compreensão completa do desempenho desses modelos.

Considerando a arquitetura e o modelo desenvolvidos neste trabalho, existem áreas específicas que podem ser aprimoradas para melhorar a eficácia e o desempenho em trabalhos futuros. Reavaliar o uso de memória das funções *serverless* como cache, considerando sua efemeridade ou aplicar métodos para persistir a memória, testar a eficiência e eficácia de outros algoritmos de previsão, como ARIMA, regressão logística, redes neurais, entre outros, para determinar se eles podem oferecer resultados superiores ao atual modelo adotado. Também, considerar a substituição do banco de dados relacional PostgreSQL por um banco de dados específico para séries temporais como o InfluxDB, com o objetivo de identificar se essa troca traria benefícios significativos em termos de desempenho, escalabilidade e flexibilidade para a aplicação em questão. Além disso, espera-se realizar testes mais amplos no mundo real para ajudar a validar a escalabilidade e o desempenho do modelo sob condições operacionais e cargas de trabalho variadas. Por fim, vislumbra-se simplificar o gerenciamento de camadas de cache e automatizar o ajuste do sistema visando reduzir significativamente a complexidade do mesmo e aumentar a atratividade do modelo para utilização no mundo real.

ACKNOWLEDGMENTS

Os autores gostariam de agradecer os seguintes órgãos de fomento: CNPq, CAPES e FAPERGS.

REFERÊNCIAS

[1] Michel Albonico, Adair Rohling, Juliano Santos, and Paulo Varela. 2021. Mining Evidences of Internet of Robotic Things (IoRT) Software from Open Source Projects. In *Proceedings of the 15th Brazilian Symposium on Software Components, Architectures, and Reuse* (Joinville, Brazil) *(SBCARS '21)*. Association for Computing Machinery, New York, NY, USA, 71–79. https://doi.org/10.1145/3483899.3483900

[2] Rohan Basu Roy and Devesh Tiwari. 2024. StarShip: Mitigating I/O Bottlenecks in Serverless Computing for Scientific Workflows. *SIGMETRICS Perform. Eval. Rev.* 52, 1 (jun 2024), 79–80. https://doi.org/10.1145/3673660.3655082

[3] Edwin F Boza, Xavier Andrade, Jorge Cedeno, Jorge Murillo, Harold Aragon, Cristina L Abad, and Andres G Abad. 2020. On implementing autonomic systems with a serverless computing approach: The case of self-partitioning cloud caches. *Computers* 9, 1 (2020), 14.

[4] Gabriel Souto Fischer, Gabriel de Oliveira Ramos, Cristiano André da Costa, Antonio Marcos Alberti, Dalvan Griebler, Dhananjay Singh, and Rodrigo da Rosa Righi. 2024. Multi-Hospital Management: Combining Vital Signs IoT Data and the Elasticity Technique to Support Healthcare 4.0. *IoT* 5, 2 (2024), 381–408. https://doi.org/10.3390/iot5020019

[5] Gartner. 2022. Quadrante Mágico para infraestrutura em nuvem e serviços de plataforma. Disponível em: <https://www.gartner.com/technology/media-products/reprints/AWS/1-2AOZQARL-PTB.html>. Acesso em: 16 junho 2023.

[6] Bishakh Chandra Ghosh, Sourav Kanti Addya, Nishant Baranwal Somy, Shubha Brata Nath, Sandip Chakraborty, and Soumya K Ghosh. 2020. Caching techniques to improve latency in serverless architectures. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE, Bengaluru, India, 666–669.

[7] Onur Göksel and Tolga Ovatman. 2021. Collaborative Path Prediction in Cache Pre-fetching for Distributed State Machines. In *2021 6th International Conference on Computer Science and Engineering (UBMK)*. IEEE, Ankara, Turkey, 489–493.

[8] J. A. Herrera-Ramírez, M. Treviño-Villalobos, and L. Víquez-Acuña. 2021. Hybrid storage engine for geospatial data using nosql and sql paradigms. *Revista Tecnología en Marcha* 34, 1 (2021), 40 – 54. https://doi.org/10.18845/tm.v34i1.4822

[9] Chaochen Hu, Zihan Sun, Chao Li, Yong Zhang, and Chunxiao Xing. 2023. Survey of Time Series Data Generation in IoT. *Sensors* 23, 15 (2023), 19 pages. https://doi.org/10.3390/s23156976

[10] Seyedeh Shabnam Jazaeri, Parvaneh Asghari, Sam Jabbehdari, and Hamid Haj Seyyed Javadi. 2023. Composition of caching and classification in edge computing based on quality optimization for SDN-based IoT healthcare solutions. *The Journal of Supercomputing* 79, 15 (01 Oct 2023), 17619–17669. https://doi.org/10.1007/s11227-023-05332-x

[11] Rajalakshmi Krishnamurthi, Adarsh Kumar, Sukhpal Singh Gill, and Rajkumar Buyya. 2023. *Serverless Computing: Principles and Paradigms*. Vol. 162. Springer Nature, Berlim, German.

[12] Chen Li, Xiaoyu Wang, Tongyu Zong, Houwei Cao, and Yong Liu. 2023. Predictive edge caching through deep mining of sequential patterns in user content retrievals. *Computer Networks* 233 (2023), 109866.

[13] Luís Manuel Meruje Ferreira, Fabio Coelho, and José Pereira. 2024. Databases in Edge and Fog Environments: A Survey. *ACM Comput. Surv.* 56, 11, Article 285 (jul 2024), 40 pages. https://doi.org/10.1145/3666001

[14] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, Seattle, USA, 122–137.

[15] Lacey-Anne Sanderson, Carolyn T Caron, Reynold L Tan, and Kirstin E Bett. 2021. A PostgreSQL Tripal solution for large-scale genotypic and phenotypic data. *Database* 2021 (08 2021), baab051. https://doi.org/10.1093/database/baab051 arXiv:https://academic.oup.com/database/article-pdf/doi/10.1093/database/baab051/39737880/baab051.pdf

[16] Josip Stanešić, Zlatan Morić, Vedran Dakić, and Matej Bašić. 2023. PREVENTION OF DNS AMPLIFICATION ATTACKS. *34th DAAAM Symposium* 34, 1 (2023), 83 – 87.

[17] Jie Sun, Mengyao Wang, Xuesong Zhu, Xiwei Zhang, and Jinsong Xue. 2023. A Survey on Cache Evaluation Metrics. *Comput. Surveys* 56, 4 (2023), 1–37. https://doi.org/10.1145/3505168

[18] Yang Tang and Junfeng Yang. 2020. Lambdata: Optimizing serverless computing by making data intents explicit. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, Beijing, China, 294–303.

[19] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. INFINICACHE: exploiting ephemeral serverless functions to build a cost-effective memory cache. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST'20)*. USENIX Association, USA, 267–282.

[20] Xin Xing, Li Wang, Yi Wang, and Yang Liu. 2022. Frequency of data transmission in wearable healthcare devices: A systematic review. *Journal of Medical Systems* 46, 12 (2022), 513.

[21] L. Yang, C. Chi, C. Pan, and Y. Qi. 2021. An intelligent caching and replacement strategy based on cache profit model for space-ground integrated network. *Mobile Information Systems* 2021 (2021), 1–13. https://doi.org/10.1155/2021/7844929

[22] Matin Yarmand, Chen Chen, Michael V. Sherer, Yash N. Shah, Peter Liu, Borui Wang, Larry Hernandez, James D. Murphy, and Nadir Weibel. 2024. Enhancing Accuracy, Time Spent, and Ubiquity in Critical Healthcare Delineation via Cross-Device Contouring. In *Proceedings of the 2024 ACM Designing Interactive Systems Conference* (IT University of Copenhagen, Denmark) *(DIS '24)*. Association for Computing Machinery, New York, NY, USA, 905–919. https://doi.org/10.1145/3643834.3660718

[23] Yujiao Zhang, Jinghan Zhang, Yan Liu, Xiaodong Chen, and Xiangyang Liu. 2021. A survey on data collection and transmission in healthcare IoT. *IEEE Transactions on Industrial Informatics* 17, 12 (2021), 7627–7640.