Vitor Trindade Rocha Ribeiro

Avaliação e Implementação de Arquitetura Distribuída para Disponibilização de Serviços de Telemedicina em Computação em Borda

Campina Grande, PB

2024

Vitor Trindade Rocha Ribeiro

Avaliação e Implementação de Arquitetura Distribuída para Disponibilização de Serviços de Telemedicina em Computação em Borda

Trabalho de Conclusão de Curso (TCC) submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Universidade Federal de Campina Grande – UFCG

Centro de Engenharia Elétrica e Informática

Departamento de Engenharia Elétrica

Orientador: Marcus Marinho Bezerra

Campina Grande, PB

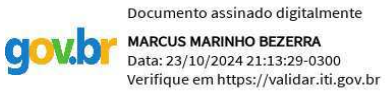
2024

Vitor Trindade Rocha Ribeiro

Avaliação e Implementação de Arquitetura Distribuída para Disponibilização de Serviços de Telemedicina em Computação em Borda

Trabalho de Conclusão de Curso (TCC) submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Trabalho aprovado. Campina Grande, PB, 23/10/2024.



Marcus Marinho Bezerra
Orientador

Danilo Freire de Souza Santos
Professor Avaliador Convidado

Campina Grande, PB
2024

Este trabalho é um tributo a tudo o que me moldou e a todos que cruzaram o meu caminho, deixando marcas profundas e inesquecíveis.

Agradecimentos

Gostaria de expressar minha mais profunda gratidão aos meus pais, Ana Paula e Flávio, pelo amor incondicional e suporte incansável que me proporcionaram ao longo de toda a minha vida. Foram os pilares que me sustentaram e, com seus ensinamentos e cuidados, me guiaram até este momento de conquista.

Aos meus irmãos, Vinicius e Laís, agradeço por me inspirarem diariamente a ser uma pessoa melhor. Nunca esquecerei o dia em que, ao iniciar o curso, recebi de Laís minha primeira calculadora científica e lapiseira, assim como o apoio constante de Vinicius em diversos momentos desafiadores. Sem vocês, esta vitória não teria o mesmo significado.

Minha gratidão também se estende à minha amada Clara, que esteve ao meu lado nos últimos anos, tornando-se uma presença essencial em minha vida. Seu apoio incondicional e conselhos foram inestimáveis para que eu pudesse alcançar esta etapa.

Aos amigos que estiveram comigo nessa jornada, agradeço pelas palavras de incentivo e pelas conversas que trouxeram leveza e alegria aos dias difíceis. Vocês tornaram este caminho mais especial e significativo.

Não poderia deixar de expressar meu sincero agradecimento ao Professor Marcus Marinho, meu orientador, cuja presença e apoio foram indispensáveis para a realização deste trabalho. Sua dedicação e disponibilidade foram fundamentais, mostrando que eu sempre poderia contar com sua orientação em cada etapa.

Agradeço, ainda, à Universidade Federal de Campina Grande pelas oportunidades oferecidas e a todos os professores e funcionários do Departamento de Engenharia Elétrica, cujo apoio e dedicação foram essenciais para minha formação acadêmica e pessoal.

"Não é sobre o que a tecnologia faz.
É sobre o que você faz com ela."

Tim Cook

Lista de ilustrações

Lista de abreviaturas e siglas

IA	Inteligência Artificial
TIC	Tecnologias de Informação e Comunicação
LGPD	Lei Geral de Proteção de Dados Pessoais
GDPR	General Data Protection Regulation
IoT	Internet das Coisas
VMs	Máquinas Virtuais
API	Application Programming Interface
CNCF	Cloud Native Computing Foundation
ECG	Eletrocardiograma
EEG	Eletroencefalograma
ML	Machine Learning
ANNs	Artificial Neural Networks
CNNs	Convolutional Neural Networks
RNNs	Recurrent Neural Networks
DNNs	Deep Neural Networks
LSTM	Long Short-Term Memory
GRUs	Gated Recurrent Units
AVX	Advanced Vector Extensions

Resumo

A telemedicina tem emergido como uma alternativa essencial para a prestação de cuidados de saúde a distância, especialmente durante crises sanitárias, como a pandemia de COVID-19. No entanto, a infraestrutura tradicional enfrenta desafios para lidar com a crescente demanda, exigindo soluções que ofereçam baixa latência e alta disponibilidade. A computação em borda surge como uma solução promissora ao aproximar o processamento dos usuários finais, melhorando a eficiência e reduzindo a latência. Desse modo, neste trabalho estão apresentadas a avaliação e a implementação de uma arquitetura distribuída baseada em computação em borda para a disponibilização dinâmica de serviços de telemedicina, com foco em consultas virtuais e diagnósticos assistidos por Inteligência Artificial (IA). A arquitetura proposta integra tecnologias de orquestração de serviços virtualizados, facilitando a gestão de microserviços e a alocação dinâmica de recursos. A implementação e avaliação da prova de conceito são realizadas utilizando *Kubernetes* para gerenciar os serviços, garantindo um sistema escalável e resiliente. Os resultados demonstram que a computação em borda pode otimizar o desempenho dos serviços de telemedicina, promovendo um atendimento de maior qualidade.

Palavras chave: Telemedicina, Computação em Borda, Arquitetura Distribuída, Kubernetes, Inteligência Artificial, Orquestração de Microserviços.

Abstract

Telemedicine has emerged as an essential alternative for delivering healthcare services remotely, particularly during health crises such as the COVID-19 pandemic. However, traditional infrastructure faces challenges in managing the increasing demand, requiring solutions that provide low latency and high availability. Edge computing offers a promising approach by bringing processing closer to end users, enhancing efficiency and reducing latency. This work presents the evaluation and implementation of a distributed architecture based on edge computing to enable the dynamic provisioning of telemedicine services, focusing on virtual consultations and AI-assisted diagnostics. The proposed architecture integrates virtualized service orchestration technologies, facilitating the management of microservices and the dynamic allocation of resources. A proof-of-concept implementation and evaluation are carried out using Kubernetes to manage the services, ensuring a scalable and resilient system. The results demonstrate that edge computing can optimize telemedicine services, delivering higher-quality care.

Keywords: Telemedicine, Edge Computing, Distributed Architecture, Kubernetes, Artificial Intelligence, Microservices Orchestration.

Sumário

1 Introdução

A telemedicina tem emergido como uma ferramenta essencial para o atendimento à saúde, especialmente em momentos de crise sanitária global, como a pandemia de COVID-19, que impulsionou a adoção de tecnologias digitais para a prestação de cuidados de saúde a distância (EHEALTH, 2010). Este modelo, que utiliza tecnologias de informação e comunicação para conectar pacientes e profissionais de saúde, oferece uma série de benefícios, incluindo maior acesso aos serviços de saúde, redução de custos e melhoria na qualidade do atendimento (EHEALTH, 2010). Contudo, o crescimento exponencial da telemedicina expôs limitações significativas nas infraestruturas atuais, que nem sempre conseguem suportar a carga crescente de usuários e garantir a qualidade do serviço. Entre os principais desafios estão a necessidade de baixa latência, alta disponibilidade e uma arquitetura capaz de escalar de forma eficiente (KEESARA; JONAS; SCHULMAN, 2020).

Para enfrentar esses desafios, a computação em borda, do inglês *edge computing*, tem se destacado como uma abordagem promissora. Diferente da computação em nuvem tradicional, onde o processamento é realizado em servidores centralizados, a computação em borda traz o processamento para mais perto do usuário final, reduzindo a latência e aumentando a eficiência no uso da largura de banda (SHI et al., 2016). Essa arquitetura distribuída é particularmente relevante para a telemedicina, onde o tempo de resposta é crítico para o diagnóstico preciso e para a interação em tempo real entre paciente e médico. Além disso, a computação em borda possibilita uma resposta mais rápida às variações na demanda, garantindo um serviço mais resiliente e de alta disponibilidade (SATYANARAYANAN, 2017).

Neste contexto, este trabalho propõe a avaliação de uma arquitetura distribuída baseada em computação em borda para a disponibilização de serviços de telemedicina, com ênfase em consultas virtuais e diagnósticos assistidos por Inteligência Artificial (IA). A solução proposta integra tecnologias de gestão e orquestração de serviços virtualizados. A utilização de uma ferramenta de orquestração facilita a gestão e a escalabilidade de microserviços, promovendo a alocação dinâmica de recursos, o que é essencial para garantir a continuidade dos serviços em situações de alta demanda (BURNS; BEDA; HIGHTOWER, 2016). A aplicação dessa arquitetura na computação em borda tem o potencial de otimizar o desempenho dos serviços de telemedicina, reduzindo a latência e melhorando a qualidade do atendimento ao paciente.

1.1 Objetivo Geral

Avaliar e implementar uma arquitetura distribuída utilizando computação em borda para a disponibilização dinâmica de serviços de telemedicina, com foco em consultas virtuais e diagnósticos assistidos por Inteligência Artificial (IA).

1.2 Objetivos Específicos

- Investigar arquiteturas distribuídas aplicáveis ao cenário de disponibilização de serviços médicos remotos;
- Projetar uma arquitetura distribuída utilizando microserviços e computação em borda;
- Implementar um protótipo da arquitetura utilizando *Kubernetes* para orquestração dos serviços.

1.3 Organização do Trabalho

Este trabalho está estruturado em 7 capítulos, incluindo este capítulo introdutório (**Capítulo 1**), organizados de forma a oferecer uma compreensão clara e progressiva dos conceitos, métodos e resultados apresentados.

No **Capítulo 2**, são explorados os fundamentos teóricos necessários para a compreensão do tema abordado, incluindo conceitos de Computação em Borda, Arquiteturas Distribuídas, e os desafios e oportunidades da Telemedicina. São discutidas também tecnologias relevantes, como Kubernetes, microserviços e Inteligência Artificial, fornecendo uma base teórica para o desenvolvimento da solução proposta.

Já no **Capítulo 3** está apresentada a metodologia utilizada para a concepção, desenvolvimento e avaliação da arquitetura distribuída proposta para a disponibilização de serviços de telemedicina. Este capítulo detalha as etapas do estudo, desde a definição dos requisitos até a implementação e os critérios de avaliação.

No **Capítulo 4**, são descritos os materiais e ferramentas utilizados no desenvolvimento da arquitetura, incluindo o ambiente de testes, as tecnologias selecionadas e as plataformas de hardware e software empregadas.

No **Capítulo 5** é detalhado o processo de implementação da solução, descrevendo a configuração do Kubernetes, o desenvolvimento e a integração dos microserviços, bem como a distribuição das cargas de trabalho na arquitetura de computação em borda. Este capítulo também discute os desafios encontrados durante a implementação e as soluções aplicadas.

No **Capítulo 6**, são apresentados e analisados os resultados obtidos com a implementação da arquitetura proposta.

Por fim, no **Capítulo 7** estão apresentados as conclusões do trabalho, destacando as principais contribuições, limitações e implicações práticas dos resultados alcançados. Além disso, são sugeridas direções para pesquisas futuras, visando o aprimoramento da arquitetura proposta e a ampliação do seu uso em outros contextos de saúde digital.

2 Fundamentação Teórica

Neste capítulo estão descritos conceitos chave relacionados à computação em borda, telemedicina e arquiteturas distribuídas, com o objetivo de embasar a proposta de uma solução tecnológica eficiente para a disponibilização de serviços de saúde a distância. São discutidas tecnologias emergentes, como *Kubernetes* e microserviços, que permitem uma infraestrutura escalável e resiliente, assim como as vantagens e os desafios da implementação de inteligência artificial na telemedicina.

2.1 Telemedicina

A telemedicina, como um avanço no setor de saúde, busca romper as barreiras físicas entre pacientes e profissionais de saúde, possibilitando cuidados médicos a distância por meio de tecnologias digitais. Essa seção descreve pontos importantes de telemedicina relacionados com o trabalho.

2.1.1 Conceitos e Definições

A telemedicina refere-se ao uso de Tecnologias de Informação e Comunicação (TIC) para oferecer serviços de saúde a distância, englobando atividades como diagnóstico, tratamento, prevenção de doenças e educação de profissionais da saúde. Esse conceito visa ultrapassar barreiras geográficas e temporais, promovendo acesso equitativo e eficiente aos cuidados médicos, independentemente da localização do paciente (EHEALTH, 2010). Ao conectar pacientes e profissionais de saúde por meio de plataformas digitais, a telemedicina possibilita a continuidade do atendimento de forma remota, o que se mostrou especialmente crucial em períodos críticos, como a pandemia de COVID-19. No entanto, sua implementação também trouxe à tona desafios como a garantia de segurança e privacidade dos dados, a manutenção de baixa latência nas conexões e a alta disponibilidade dos serviços (DINESEN et al., 2016; KEESARA; JONAS; SCHULMAN, 2020).

2.1.2 O Impacto da Pandemia de COVID-19

A pandemia de COVID-19 acelerou a adoção da telemedicina ao redor do mundo, transformando-a de uma prática complementar em uma necessidade fundamental para garantir a continuidade dos cuidados de saúde. Com as restrições de mobilidade e o distanciamento social, tornou-se evidente que muitos sistemas de saúde não estavam preparados para lidar com a sobrecarga dos hospitais, mantendo, ao mesmo tempo, o atendimento de pacientes com doenças crônicas e outras condições não relacionadas ao

COVID-19 (KEESARA; JONAS; SCHULMAN, 2020). Nesse cenário, a telemedicina emergiu como uma solução prática, proporcionando consultas remotas, monitoramento de sintomas e gestão de tratamentos, especialmente para populações vulneráveis e em áreas remotas, onde o acesso aos serviços de saúde é limitado (GREENHALGH et al., 2020).

Além disso, o contexto da pandemia impulsionou avanços tecnológicos na área, como o uso de inteligência artificial para triagem e diagnósticos preliminares, e o desenvolvimento de plataformas de comunicação seguras para garantir a privacidade dos pacientes(WHITELAW et al., 2020). Apesar desses progressos, a rápida expansão da telemedicina evidenciou a necessidade de uma infraestrutura tecnológica mais robusta e regulamentações apropriadas para minimizar desigualdades de acesso e garantir a eficácia dos serviços oferecidos (BOKOLO, 2021).

2.1.3 Desafios na Telemedicina

Apesar dos avanços recentes e fundamental relevância durante a pandemia, a telemedicina ainda enfrenta desafios significativos, especialmente relacionados à infraestrutura tecnológica e arquiteturas de suporte. A necessidade de garantir alta disponibilidade e baixa latência para serviços de saúde digital impõe requisitos rigorosos à infraestrutura de comunicação e computação. A computação em borda surge como uma solução promissora ao aproximar o processamento dos dados do local de origem, reduzindo a latência e otimizando a utilização de largura de banda. Contudo, a implementação de uma arquitetura distribuída em borda demanda um gerenciamento eficiente de recursos e a capacidade de escalar rapidamente para lidar com picos de demanda, o que requer plataformas robustas para orquestrar e distribuir serviços de forma dinâmica e resiliente (KRUSE et al., 2017).

Ademais, garantir a resiliência e a escalabilidade da infraestrutura é um desafio constante, especialmente em cenários de alta carga e quando se lida com grandes volumes de dados médicos sensíveis. É necessário assegurar que as arquiteturas distribuídas consigam atender requisitos de resiliência sem comprometer a continuidade dos serviços. A escolha adequada de tecnologias de orquestração e balanceamento de carga é crucial para atingir esse objetivo e garantir a experiência do usuário final (WIIG et al., 2020).

Outros desafios, como a segurança e privacidade dos dados dos pacientes, são críticos, pois a transmissão de informações médicas sensíveis entre ambientes de computação distribuídos exige conformidade com regulamentações como a LGPD (Lei Geral de Proteção de Dados) e o GDPR (Regulamento Geral de Proteção de Dados). A LGPD é uma lei brasileira que visa proteger os dados pessoais dos cidadãos, estabelecendo diretrizes sobre como empresas e instituições devem coletar, armazenar e tratar essas informações. Da mesma forma, o GDPR é uma regulamentação da União Europeia que padroniza a proteção de dados pessoais, impondo requisitos rigorosos para garantir a privacidade e os direitos dos indivíduos. Além disso, existem desafios relacionados à aceitação e adaptação dos usuários

às novas tecnologias, bem como questões éticas e legais que precisam ser consideradas, muitas vezes envolvendo regulamentações específicas e a capacitação adequada dos usuários (GAJARAWALA; PELKOWSKI, 2021; GREENHALGH et al., 2020; MEHROTRA et al., 2020).

2.2 Computação em Borda

A computação em borda, como uma inovação tecnológica, visa aproximar o processamento de dados do local onde eles são gerados, reduzindo a latência e melhorando a eficiência dos sistemas. Essa abordagem permite a execução de serviços críticos mais próximos dos usuários e dispositivos, aumentando a velocidade de resposta e a confiabilidade. Esta seção aborda aspectos essenciais da computação em borda que se conectam diretamente com o escopo deste trabalho.

2.2.1 Conceito de Computação em Borda

A computação em borda é um paradigma de processamento de dados que desloca o processamento e o armazenamento da informação para locais mais próximos do usuário final, em vez de centralizar esses processos em servidores remotos na nuvem. Este conceito visa reduzir a latência, melhorar a eficiência do uso de banda e aumentar a disponibilidade dos serviços, especialmente em aplicações que requerem respostas com baixa latência, como a telemedicina, Internet das Coisas, do inglês *Internet Of Things* (IoT), veículos autônomos e aplicações industriais (SHI et al., 2016).

Ao processar os dados na borda da rede, ou seja, perto da origem ou do ponto de coleta, a computação em borda permite uma resposta mais rápida e eficiente às necessidades dos usuários. Isso é particularmente relevante em ambientes onde o tempo de resposta é crítico, como no diagnóstico remoto em telemedicina, onde atrasos podem afetar significativamente a qualidade do atendimento ao paciente (SATYANARAYANAN, 2017). Ainda, a computação em borda ajuda a reduzir o tráfego de dados para servidores de computação em nuvem, diminuindo custos operacionais e melhorando a escalabilidade dos serviços.

2.2.2 Vantagens da Computação em Borda

A computação em borda oferece diversas vantagens em relação ao modelo tradicional de computação centralizada em nuvem, especialmente em contextos onde a latência, a largura de banda e a privacidade são cruciais. Uma das principais vantagens é a redução da latência. Como o processamento e a análise dos dados ocorrem mais próximos dos dispositivos, a computação em borda reduz o tempo de resposta, o que é essencial em

aplicações que demandam decisões rápidas, como em telemedicina, sistemas de veículos autônomos e monitoramento industrial (SHI et al., 2016).

Outra vantagem significativa é a eficiência no uso da largura de banda. Em vez de enviar grandes volumes de dados brutos para processamento em servidores em nuvem, a computação em borda permite que esses dados sejam processados localmente, enviando para servidores em nuvem apenas informações filtradas ou relevantes. Isso não apenas reduz o tráfego de rede, mas também diminui os custos associados à transmissão de dados e ao armazenamento na nuvem (SATYANARAYANAN, 2017).

A computação em borda também contribui para a melhoria da privacidade e segurança dos dados. Como os dados são processados localmente, há menos necessidade de transferi-los através de redes públicas ou armazená-los em servidores remotos, o que reduz os riscos de exposição a ataques cibernéticos. Este aspecto é particularmente importante em setores que utilizam dados sensíveis, como a saúde, onde a proteção de informações pessoais dos pacientes é uma prioridade (VARGHESE; BUYYA, 2016).

Outras duas características importantes são a resiliência e disponibilidade são outras vantagens importantes. Em uma arquitetura de computação em borda, mesmo que a conectividade com a nuvem seja perdida, os dispositivos locais ainda podem continuar operando e processando dados, garantindo a continuidade dos serviços. Isso torna a computação em borda uma solução robusta para ambientes onde a confiabilidade é um fator crítico (SATYANARAYANAN, 2017).

2.2.3 Aplicações da Computação em Borda na Telemedicina

A computação em borda desempenha um papel crucial na telemedicina ao permitir o processamento eficiente de grandes volumes de dados gerados por dispositivos médicos e sensores remotos. Uma das principais aplicações é no monitoramento contínuo de pacientes, onde dispositivos conectados, como monitores de sinais vitais, realizam a coleta e análise de dados localmente. Ao processar esses dados na borda, é possível detectar anomalias rapidamente e alertar rapidamente profissionais de saúde sobre possíveis emergências, reduzindo o tempo de resposta e melhorando os desfechos clínicos (RAHMANI et al., 2018).

Outra aplicação relevante da computação em borda no contexto de telemedicina é no suporte a diagnósticos assistidos por inteligência artificial (IA). A computação em borda permite que algoritmos de IA executem análises de imagens médicas, como radiografias e tomografias, diretamente onde os dados são capturados, minimizando o atraso no diagnóstico. Esse processamento local é especialmente útil em áreas remotas ou com conectividade limitada, onde o envio de grandes arquivos de imagem para a nuvem pode ser impraticável (WANG et al., 2019).

A computação em borda também é usada para gestão de recursos e otimização de rede em ambientes hospitalares inteligentes. Hospitais podem utilizar essa tecnologia para gerenciar dinamicamente a distribuição de carga de trabalho entre servidores locais e a nuvem, garantindo que os serviços de telemedicina permaneçam disponíveis e funcionais mesmo durante picos de demanda. Essa abordagem ajuda a manter a qualidade do atendimento, otimizando o uso de recursos e reduzindo custos operacionais (VARGHESE; BUYYA, 2016).

Outra aplicação importante da computação em borda aplicada no contexto da telemedicina é a proteção da privacidade dos dados dos pacientes. Ao processar informações sensíveis localmente, em vez de transmiti-las integralmente para a nuvem, os riscos de exposição e vazamento de dados são significativamente reduzidos. Isso é especialmente importante para garantir conformidade com regulamentações de proteção de dados, como a LGPD e o GDPR, ao mesmo tempo em que se mantém a confiança do paciente na segurança dos seus dados médicos (ROMAN; LOPEZ; MAMBO, 2018).

2.3 Arquiteturas Distribuídas

As arquiteturas distribuídas, como um modelo avançado de organização de sistemas, dividem o processamento e o armazenamento de dados entre múltiplas máquinas interconectadas, promovendo maior escalabilidade, resiliência e flexibilidade. Essa abordagem permite lidar com grandes volumes de dados e fornecer serviços de forma contínua e eficiente. Esta seção explora os principais aspectos das arquiteturas distribuídas que se relacionam com o objetivo deste trabalho.

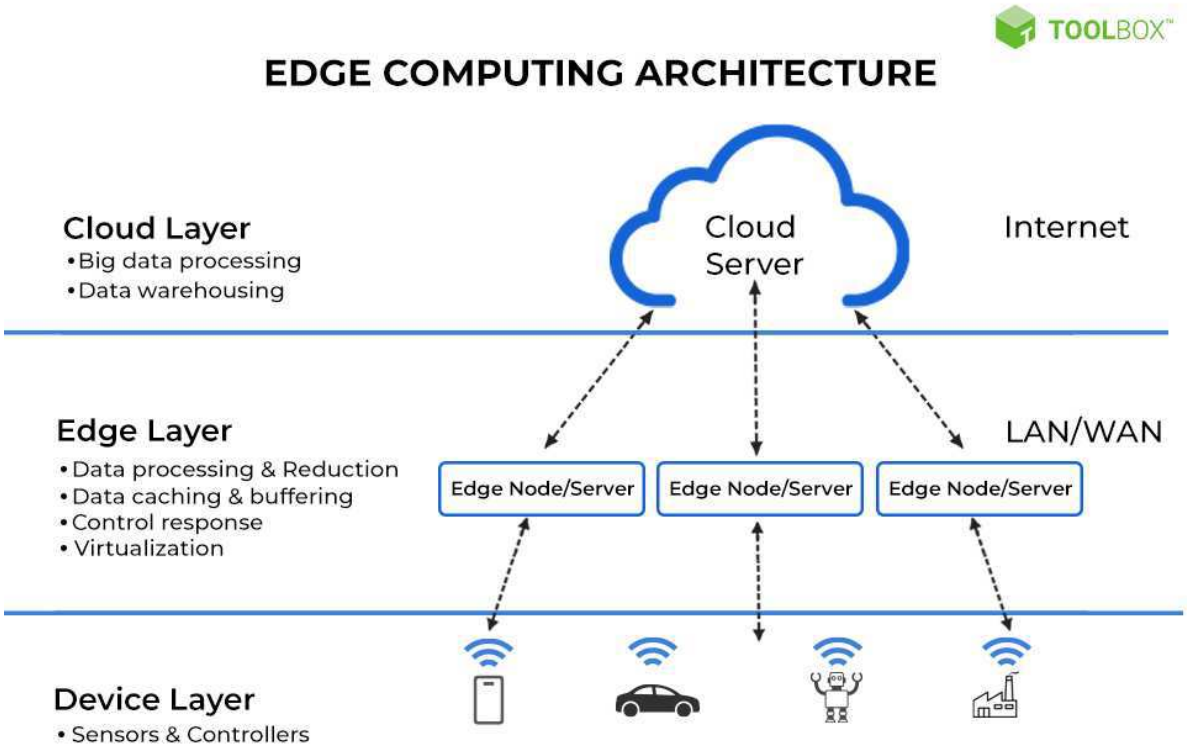
2.3.1 Definições e Tipos de Arquiteturas

As arquiteturas de computação em borda são projetadas para descentralizar o processamento de dados, distribuindo-o em diferentes níveis da rede, desde dispositivos de borda até servidores intermediários e centrais. Esse modelo difere da abordagem tradicional de computação em nuvem, onde o processamento é amplamente centralizado em grandes *data centers*. Na computação em borda, o objetivo é realizar o processamento o mais próximo possível da origem dos dados, minimizando a latência, economizando largura de banda e garantindo uma resposta mais rápida e eficiente (SHI et al., 2016).

Existem diferentes tipos de arquiteturas que podem ser aplicadas em computação em borda. Uma das mais comuns é a arquitetura de três camadas, que consiste em dispositivos de borda (como sensores e dispositivos IoT), gateways de borda (nós intermediários que agregam e processam dados localmente), e servidores de nuvem (responsáveis pelo armazenamento de longo prazo e processamento intensivo de dados). Uma ilustração dessa arquitetura pode ser vista na Figura 1. Essa arquitetura permite uma distribuição eficiente

de carga, onde tarefas críticas são executadas nas camadas mais próximas do usuário, enquanto operações menos sensíveis ao tempo podem ser enviadas para a nuvem (YI; LI; LI, 2015).

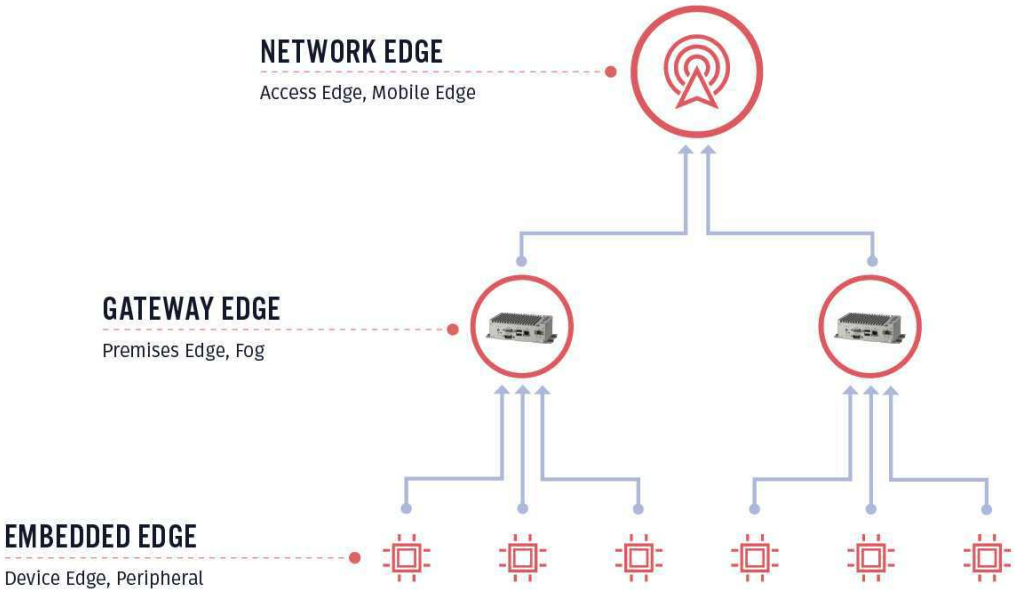
Figura 1 – Arquitetura de três camadas



Fonte: (Spiceworks, 2024).

Outra abordagem é a arquitetura hierárquica, que pode ser vista na Figura 2, onde múltiplas camadas de nós de borda são organizadas em níveis de acordo com suas capacidades de processamento e armazenamento. Cada nó de borda pode interagir com outros nós de níveis superiores ou inferiores, criando uma rede altamente resiliente e adaptável. Essa arquitetura é especialmente vantajosa em cenários de grande escala, como cidades inteligentes e redes de saúde pública, onde a flexibilidade e a redundância são essenciais para garantir a continuidade dos serviços (ROMAN; LOPEZ; MAMBO, 2018).

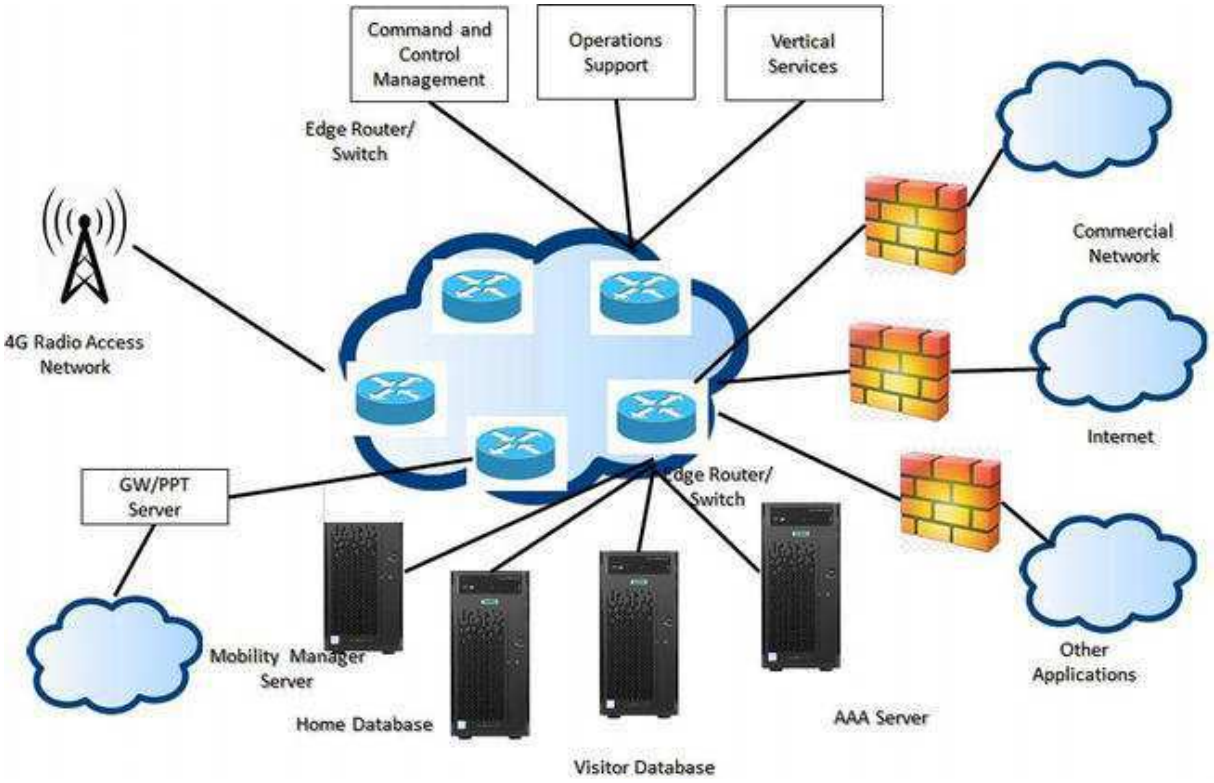
Figura 2 – Arquitetura Hierárquica



Fonte: (Losant, 2024).

Além disso, há a arquitetura de malha *(mesh architecture)*, na qual todos os dispositivos de borda e gateways estão conectados uns aos outros em uma rede distribuída. Essa abordagem melhora a resiliência da rede, pois permite que os dados sejam roteados através de diferentes caminhos em caso de falhas. A arquitetura de malha é particularmente útil em ambientes com conectividade intermitente ou onde a robustez da rede é uma prioridade (DASTJERDI; BUYYA, 2016). Essa arquitetura está ilustrada na Figura 3.

Figura 3 – Arquitetura de Malha



Fonte: (Intechopen, 2024).

Cada tipo de arquitetura tem suas vantagens e limitações, e a escolha adequada depende dos requisitos específicos da aplicação. Na telemedicina, por exemplo, onde é crucial garantir baixa latência, alta disponibilidade e segurança dos dados, a combinação de arquiteturas de borda e nuvem, com foco em otimização de recursos e gerenciamento dinâmico, pode oferecer a solução mais eficaz.

2.3.2 Comparação com Arquiteturas Centralizadas

A computação em borda difere significativamente das arquiteturas centralizadas, como a computação em nuvem tradicional, ao trazer o processamento e o armazenamento de dados para mais próximo da origem dos dados, em vez de depender de *data centers* centralizados. Uma das principais vantagens dessa abordagem é a redução da latência. Em arquiteturas centralizadas, todos os dados devem ser enviados para um servidor remoto para processamento, o que pode causar atrasos significativos, especialmente quando grandes volumes de dados estão envolvidos ou quando a conectividade de rede é limitada. Por outro lado, na computação em borda, o processamento é feito localmente, resultando em tempos de resposta mais rápidos, essenciais para aplicações que exigem decisões rápidas, como em telemedicina (SHI et al., 2016).

Além disso, a eficiência no uso de largura de banda é uma vantagem importante das arquiteturas de borda. Enquanto as arquiteturas centralizadas exigem que todos os dados sejam transmitidos para a nuvem, a computação em borda permite que apenas os dados relevantes ou processados sejam enviados, reduzindo significativamente o tráfego de rede e os custos associados (SATYANARAYANAN, 2017). Isso é particularmente benéfico em ambientes onde a largura de banda é escassa ou cara, como em áreas rurais ou remotas.

Outra diferença notável é a resiliência e disponibilidade. Nas arquiteturas centralizadas, a falha de um *data center* pode interromper completamente o acesso aos serviços. Em contraste, as arquiteturas distribuídas, como a computação em borda, oferecem maior resiliência, pois os dados e o processamento estão espalhados por vários nós na rede. Isso significa que, mesmo que um nó falhe, outros podem continuar operando, garantindo a continuidade dos serviços (DASTJERDI; BUYYA, 2016).

Por outro lado, as arquiteturas centralizadas podem ser mais fáceis de gerenciar e monitorar, pois todos os dados e processos estão em um local centralizado, permitindo uma gestão mais simplificada de segurança, atualizações e manutenção. No entanto, essa centralização também representa um ponto único de falha e pode tornar o sistema mais vulnerável a ataques cibernéticos ou falhas de infraestrutura (YI; LI; LI, 2015).

A escolha entre uma arquitetura centralizada e uma arquitetura de borda depende das necessidades específicas da aplicação. Para serviços que exigem baixa latência, alta resiliência e uso eficiente de largura de banda, como na telemedicina, a computação em borda oferece vantagens claras sobre as soluções centralizadas. No entanto, em contextos onde a centralização de dados e o gerenciamento simplificado são mais importantes, uma abordagem centralizada pode ser mais apropriada.

2.3.3 Resiliência e Escalabilidade em Arquiteturas Distribuídas

Arquiteturas distribuídas são projetadas para fornecer alta resiliência e escalabilidade da capacidade total de processamento, do armazenamento ou do *throughput*. Sendo esses atributos essenciais para aplicações críticas, como a telemedicina. A resiliência refere-se à capacidade do sistema de continuar operando corretamente mesmo em caso de falhas parciais, como a perda de um nó de processamento ou falhas de rede. Em arquiteturas distribuídas, os dados e processos são replicados e distribuídos entre vários nós, de modo que a falha de um componente não comprometa todo o sistema. Essa redundância aumenta a disponibilidade e a confiabilidade dos serviços, garantindo a continuidade do atendimento ao paciente mesmo em condições adversas (WEATHERSPOON; KUBIATOWICZ, 2002).

A escalabilidade, por outro lado, permite que o sistema se adapte a cargas variáveis de trabalho, adicionando ou removendo recursos conforme necessário. Em arquiteturas distribuídas, a capacidade de processamento pode ser aumentada simplesmente adicionando

mais nós à rede, sem a necessidade de grandes revisões na infraestrutura existente. Isso é particularmente importante em ambientes de telemedicina, onde a demanda pode flutuar significativamente, como durante surtos de doenças ou emergências de saúde pública (DEAN; GHEMAWAT, 2008).

Um caso real que exemplifica a resiliência e escalabilidade de arquiteturas distribuídas na prática é o sistema de telemedicina do Hospital Mount Sinai, em Nova York, durante a pandemia de COVID-19. Com a súbita necessidade de realizar um grande número de consultas virtuais, o hospital enfrentou um aumento exponencial na demanda por serviços de telemedicina. Para atender a essa demanda, o Mount Sinai implementou uma arquitetura distribuída utilizando computação em borda e serviços baseados em nuvem para gerenciar a carga de trabalho. A computação em borda permitiu o processamento local de dados de pacientes, como imagens médicas e sinais vitais, enquanto a nuvem forneceu capacidade adicional de processamento e armazenamento (Microsoft, 2023).

2.4 Métodos tradicionais de Implatação de Aplicações

Os métodos tradicionais de implantação de aplicações envolvem a utilização de infraestruturas físicas, conhecidas como baremetal, e a virtualização por meio de máquinas virtuais. Esses métodos proporcionam controle direto sobre os recursos de hardware e permitem a execução de várias instâncias de aplicações em ambientes isolados, atendendo a requisitos específicos de desempenho e segurança. Esta seção explora os principais aspectos desses métodos tradicionais de implantação e sua relevância para o contexto deste trabalho.

2.4.1 *Bare Metal*

Antes de abordar tecnologias mais modernas de infraestrutura escalável, é fundamental compreender como elas diferem dos métodos tradicionais de implantação. Um dos métodos mais convencionais é conhecido como *bare metal*, que se refere à execução de aplicações diretamente em servidores físicos, sem a intermediação de hipervisores ou sistemas de virtualização. Neste contexto, a aplicação interage diretamente com o hardware subjacente, o que elimina a sobrecarga gerada por camadas intermediárias e, teoricamente, permite um desempenho otimizado em comparação a ambientes virtualizados (ROSADO; AL., 2014).

No modelo bare metal, o sistema operacional é instalado diretamente sobre o hardware, sem a presença de um hipervisor ou mecanismo de virtualização. Este sistema operacional pode variar entre distribuições *Linux*, *Windows*, *FreeBSD* e outros, dependendo dos requisitos da aplicação. Após a configuração do sistema, todas as bibliotecas e dependências necessárias são instaladas, seguido pela implantação da própria aplicação (HE; WILSON, 2015). Este processo é ilustrado na Figura 4.

Figura 4 – *Diagrama de arquitetura Bare Metal*

Fonte: Adaptado de (SINGH; AL., 2019).

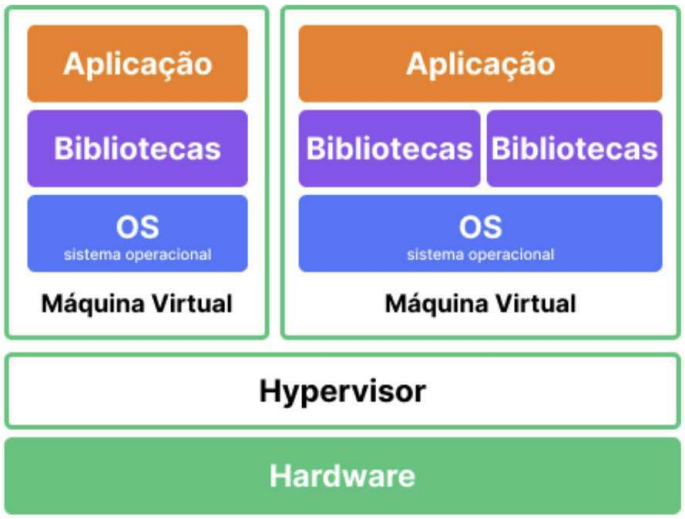
Apesar das vantagens de desempenho, a abordagem bare metal apresenta desafios quando se trata de flexibilidade e escalabilidade. A adição de novos recursos, como memória RAM ou capacidade de processamento, requer modificações físicas no servidor existente ou a aquisição de novos equipamentos. Isso torna a escalabilidade uma tarefa complexa e custosa, principalmente para empresas que necessitam de aumento rápido de capacidade computacional (ALICHERRY; LAKSHMAN, 2009). Além disso, a ausência de isolamento entre aplicações pode levar a conflitos de recursos e problemas de segurança (SHAH; AL., 2020).

2.4.2 Máquinas Virtuais

Ao nos afastarmos do modelo de *bare metal*, adentramos no conceito de máquinas virtuais, uma abordagem amplamente utilizada para superar as limitações impostas pela execução direta sobre o hardware. Nesse modelo, os recursos físicos são abstraídos e virtualizados, permitindo a criação de múltiplas instâncias de sistemas operacionais independentes sobre o mesmo hardware físico. A virtualização é facilitada por um software especializado chamado hipervisor, que gerencia e particiona os recursos da máquina física em várias máquinas virtuais (*VMs*) (SMITH; NAIR, 2005).

Cada máquina virtual opera como se fosse um sistema independente, com seu próprio sistema operacional, bibliotecas e uma instância isolada da aplicação. Isso garante um isolamento entre as VMs, o que é benéfico tanto em termos de segurança quanto de gerenciamento de recursos. O hipervisor pode ser de dois tipos: tipo 1 (*bare metal*) e tipo 2 (*hosted*). O primeiro é instalado diretamente sobre o hardware, enquanto o segundo roda sobre um sistema operacional já existente (ROSENBLUM; GARFINKEL, 1995). A Figura 5 ilustra a arquitetura de uma solução baseada em máquinas virtuais.

Figura 5 – Diagrama de arquitetura de Máquinas Virtuais



Fonte: Adaptado de (SINGH; AL., 2019).

A virtualização facilita a escalabilidade, permitindo que novas máquinas virtuais sejam criadas sob demanda, sem a necessidade de adquirir mais hardware físico. Esse processo é altamente flexível, visto que o hipervisor gerencia eficientemente a alocação de recursos, como CPU, memória e armazenamento, entre as VMs existentes (BARHAM; AL., 2003). No entanto, a abordagem de máquinas virtuais ainda enfrenta desafios quando se trata de agilidade nas implantações, uma vez que cada VM requer sua própria instância de sistema operacional, kernel e outros componentes essenciais (GOVIL; AL., 1999).

Embora as VMs compartilhem o hardware físico subjacente, o *overhead* gerado pela replicação de sistemas operacionais pode impactar a performance. Esse é um fator crucial ao se considerar ambientes de alta densidade de VMs ou quando se busca maximizar a utilização de recursos de maneira mais eficiente (CHADHA; AL., 2015). Dessa forma, enquanto as máquinas virtuais oferecem um grau significativo de isolamento e flexibilidade, surgiram tecnologias mais recentes, como contêineres, que visam reduzir a sobrecarga ao compartilhar o mesmo *kernel* do sistema operacional entre múltiplas instâncias de aplicações (MERKEL, 2014).

2.5 Tecnologias Emergentes para Infraestrutura Escalável

As tecnologias emergentes para infraestrutura escalável, como containers, microsserviços e Kubernetes, estão transformando a maneira como as aplicações são desenvolvidas, implantadas e gerenciadas. Containers permitem empacotar e isolar aplicações de forma leve e eficiente, enquanto microsserviços promovem uma arquitetura modular, facilitando o desenvolvimento ágil e a escalabilidade. O Kubernetes, por sua vez, orquestra e gerencia

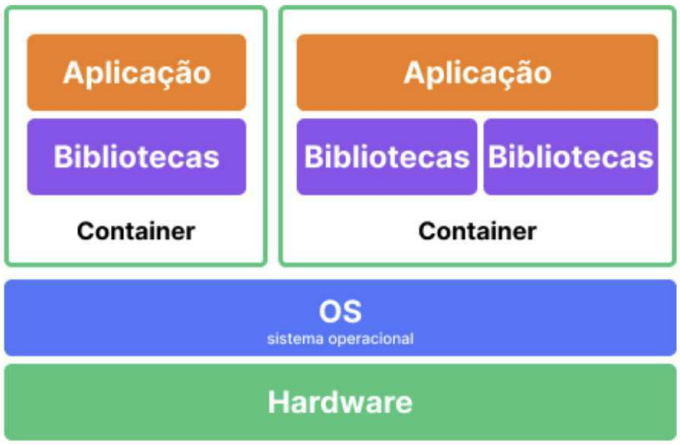
esses containers em ambientes distribuídos, garantindo alta disponibilidade e escalabilidade automática. Esta seção explora como essas tecnologias emergentes contribuem para a criação de uma infraestrutura mais flexível e escalável, em sintonia com os objetivos deste trabalho.

2.5.1 Contêineres e o Surgimento de Microsserviços

Nos últimos anos, os contêineres emergiram como uma tecnologia disruptiva para o desenvolvimento e a implantação de aplicações, permitindo uma abordagem mais eficiente de virtualização. Diferente do modelo tradicional de máquinas virtuais, que replicam o sistema operacional inteiro, os contêineres virtualizam apenas o sistema operacional subjacente, compartilhando o kernel com as aplicações implantadas. Essa característica resulta em um consumo reduzido de recursos, visto que cada contêiner não necessita de um sistema operacional completo, mas apenas das bibliotecas e dependências essenciais para sua execução (MERKEL, 2014).

A plataforma de contêineres é executada sobre um sistema operacional hospedeiro, utilizando o kernel já existente para suportar as operações dos contêineres. Esse mecanismo permite que múltiplos contêineres sejam executados em paralelo, mantendo um alto grau de isolamento entre as instâncias. Embora os contêineres compartilhem o kernel do sistema operacional subjacente, cada contêiner possui seu próprio espaço de usuário, bibliotecas e uma camada de sistema de arquivos independente (TURNBULL, 2014). A Figura 6 ilustra a arquitetura de uma solução baseada em contêineres.

Figura 6 – *Diagrama de arquitetura de Contêineres*



Fonte: Adaptado de (SINGH; AL., 2019).

Uma das principais vantagens dos contêineres é a agilidade na implantação e no escalonamento. Uma vez que uma imagem de contêiner é configurada e testada, ela pode ser replicada de maneira consistente em diferentes ambientes, garantindo que a aplicação

funcione da mesma forma independentemente do contexto em que está sendo executada (BERNSTEIN, 2014). Esse comportamento elimina o problema clássico de inconsistência entre ambientes de desenvolvimento e produção.

Além disso, a facilidade de isolamento dos contêineres os torna ideais para a implementação de microsserviços. Um microsserviço é uma pequena aplicação independente que desempenha uma única tarefa dentro de um sistema mais amplo. Em contraste com as arquiteturas monolíticas, que concentram todas as funcionalidades em um único pacote, a arquitetura baseada em microsserviços divide a aplicação em múltiplas partes menores, cada uma implantada como um contêiner separado (DRAGONI et al., 2017). Isso permite que cada contêiner/microsserviço seja escalado, atualizado e gerenciado de forma independente, resultando em maior flexibilidade e resiliência do sistema como um todo (NEWMAN, 2015).

Essa abordagem também simplifica a manutenção e a evolução de grandes sistemas, uma vez que novos contêineres podem ser introduzidos ou removidos sem impacto significativo no restante do sistema. Além disso, a comunicação entre contêineres ocorre por meio de APIs bem definidas, facilitando a integração e a interoperabilidade (TURNBULL, 2014). Abaixo vemos algumas das principais vantagens da utilização de contêineres em comparação com outros métodos de virtualização.

Vantagem	**Descrição**
Rapidez	Implantação rápida sem necessidade de instalar um sistema operacional completo, economizando tempo e recursos.
Consistência	Todas as bibliotecas e requisitos da aplicação são empacotados no contêiner, garantindo que a aplicação funcione de maneira consistente em qualquer ambiente.
Simplicidade	O processo de implantação é simplificado, uma vez que a imagem do contêiner é testada e compartilhada, eliminando a necessidade de configurações adicionais.
Escalabilidade	Facilidade em escalar aplicações adicionando mais contêineres em vez de configurar novos servidores físicos ou máquinas virtuais.
Isolamento	Cada contêiner é isolado, garantindo que as aplicações não interfiram umas com as outras, mesmo estando no mesmo servidor.
Eficiência de Recursos	Aproveita o kernel do sistema operacional subjacente, utilizando menos recursos do que as máquinas virtuais.

Com a combinação de alto desempenho, escalabilidade e flexibilidade, os contêineres se tornaram a base para a arquitetura de microsserviços, que, por sua vez, possibilita

o desenvolvimento ágil e o crescimento dinâmico de aplicações complexas e distribuídas (DRAGONI et al., 2017).

2.5.2 Microsserviços

Os microserviços são uma abordagem de arquitetura de software que estrutura uma aplicação como um conjunto de serviços pequenos, independentes e orientados a negócios. Cada microserviço é responsável por uma função específica, comunicando-se com outros serviços por meio de APIs bem definidas. Essa arquitetura oferece uma série de vantagens, como maior modularidade, facilidade de manutenção e capacidade de escalar componentes individualmente conforme necessário, sem afetar o sistema como um todo (NEWMAN, 2015).

Na telemedicina, os microserviços são particularmente úteis para construir sistemas flexíveis e escaláveis, onde cada serviço pode ser desenvolvido, implantado e atualizado independentemente. Por exemplo, um sistema de telemedicina pode ser dividido em serviços separados para gerenciamento de consultas, armazenamento de registros médicos, autenticação de usuários, comunicação por vídeo, entre outros. Isso permite que cada serviço seja otimizado e escalado de acordo com sua carga específica, garantindo uma distribuição eficiente de recursos e mantendo o desempenho ideal mesmo durante picos de demanda (FOWLER; LEWIS, 2014).

Uma das principais vantagens dos microserviços é a resiliência. Em uma arquitetura monolítica, uma falha em um único componente pode comprometer todo o sistema. Já em uma arquitetura de microserviços, falhas são isoladas, de modo que um problema em um serviço específico não afeta diretamente os demais. Isso é particularmente importante em contextos críticos, como a telemedicina, onde a continuidade do serviço é essencial para o atendimento ao paciente (THöNES, 2015).

Os microserviços também facilitam a implantação contínua e a agilidade no desenvolvimento. Equipes de desenvolvimento podem trabalhar simultaneamente em diferentes serviços, implementando novas funcionalidades e corrigindo bugs sem a necessidade de paralisar todo o sistema. Essa abordagem permite uma resposta mais rápida às necessidades dos usuários e uma maior adaptabilidade às mudanças de mercado ou regulamentação, o que é crucial no setor de saúde, onde os requisitos podem evoluir rapidamente (DRAGONI et al., 2017).

Além disso, a arquitetura de microserviços promove a portabilidade e a interoperabilidade entre diferentes sistemas e plataformas, uma vez que os serviços podem ser implementados em diferentes linguagens de programação e executados em diversos ambientes de infraestrutura. Isso é especialmente vantajoso em ambientes de telemedicina que requerem a integração de vários componentes, como sistemas de informação hospitalar,

dispositivos médicos e plataformas de comunicação (LEWIS; FOWLER, 2015).

Portanto, a adoção de microserviços permite a criação de sistemas de telemedicina mais robustos, escaláveis e eficientes, melhorando a qualidade do atendimento ao paciente e garantindo a continuidade dos serviços mesmo em situações adversas.

2.5.3 Kubernetes e Orquestração de Contêineres

Kubernetes é uma plataforma de código aberto para orquestração de contêineres que automatiza o gerenciamento, escalonamento e operação de aplicações em contêineres. Criado originalmente pelo Google e agora mantido pela Cloud Native Computing Foundation (CNCF), o Kubernetes se tornou a solução de referência para a implantação e gestão de aplicações distribuídas em larga escala. A orquestração de contêineres refere-se ao processo de automatizar a implantação, o gerenciamento, o dimensionamento e a rede de contêineres, garantindo alta disponibilidade, resiliência e eficiência operacional (BURNS; BEDA; HIGHTOWER, 2016).

A principal vantagem do Kubernetes é sua capacidade de gerenciar a infraestrutura de forma eficiente por meio da orquestração de contêineres, permitindo que os desenvolvedores se concentrem no desenvolvimento de aplicativos, enquanto o Kubernetes lida com a complexidade da infraestrutura subjacente. Isso é particularmente importante em arquiteturas distribuídas, onde os contêineres precisam ser escalados dinamicamente, monitorados e gerenciados em diferentes nós de computação. O Kubernetes fornece recursos como autoescalamento, balanceamento de carga e autorrecuperação, que garantem que as aplicações permaneçam altamente disponíveis e performantes (HIGHTOWER; BURNS; BEDA, 2018).

No contexto da telemedicina, Kubernetes permite a implementação de uma infraestrutura escalável e resiliente. À medida que a demanda por serviços de telemedicina varia, especialmente em resposta a eventos imprevisíveis, como pandemias, o Kubernetes facilita a adição ou remoção de contêineres de forma dinâmica, garantindo que os recursos de computação sejam utilizados de maneira eficiente. Isso é crucial para manter a qualidade do serviço, minimizando a latência e garantindo que as aplicações críticas estejam sempre disponíveis (VASSOLER, 2020).

O Kubernetes facilita a portabilidade das aplicações em diferentes ambientes de nuvem ou infraestrutura local, permitindo que as organizações de saúde implementem estratégias híbridas de nuvem e borda. Essa flexibilidade é importante para garantir a continuidade dos serviços de telemedicina, mesmo em condições de rede desafiadoras ou com requisitos regulatórios específicos. Com o uso de ferramentas de gerenciamento, como Helm e Istio, Kubernetes também simplifica a implantação de microserviços e a aplicação de políticas de segurança, garantindo que os dados dos pacientes sejam protegidos de

acordo com as regulamentações de privacidade (BURNS; OPPENHEIMER; BREWER, 2019).

2.5.4 Principais Objetos Kubernetes

Dentro do Kubernetes, os objetos representam as entidades persistentes no sistema e descrevem o estado desejado do cluster. Esses objetos, como *Pods*, *Services* e *Ingress*, são fundamentais para a operação do Kubernetes, permitindo a definição e a gerência das cargas de trabalho de forma eficiente e escalável.

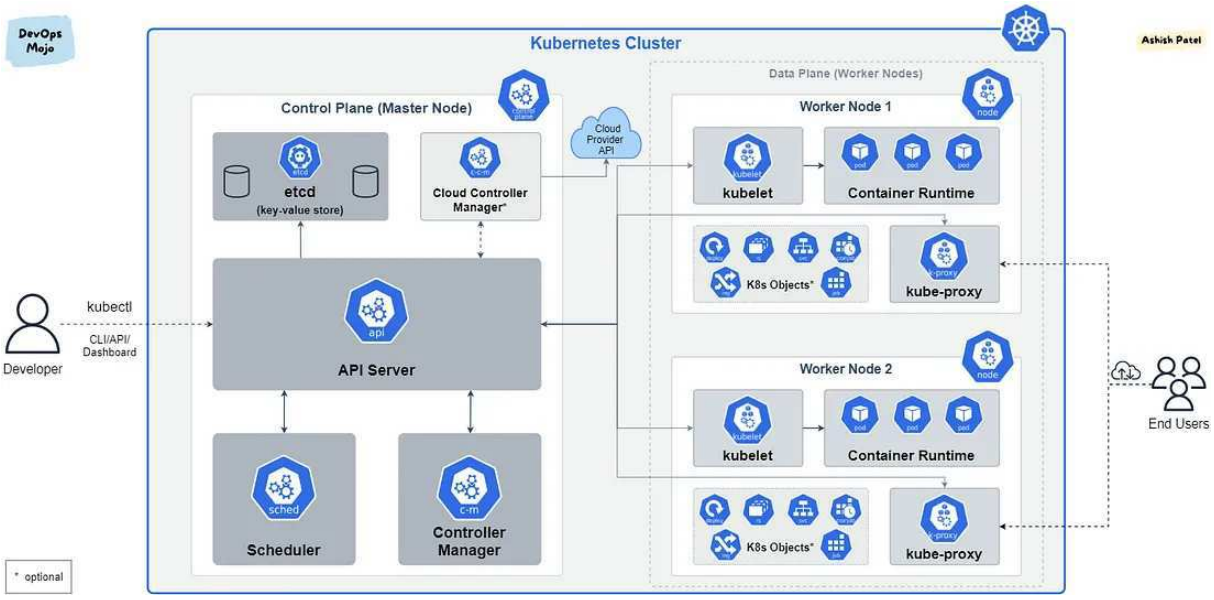
2.5.4.1 Pods

O **Pod** é a menor unidade computacional do Kubernetes, responsável por encapsular um ou mais contêineres que compartilham o mesmo espaço de rede e armazenamento. Cada pod possui um endereço IP único dentro do cluster e é projetado para executar um único processo ou uma carga de trabalho pequena. Pods são efêmeros por natureza e, quando são destruídos ou falham, são substituídos automaticamente.

2.5.4.2 Clusters

O **Cluster** é a instância completa do Kubernetes, consistindo em um conjunto de **Nodes** que trabalham juntos para executar as aplicações. Um cluster Kubernetes é dividido em dois componentes principais: *Control Plane* e *Nodes*. O Control Plane gerencia o estado global do cluster e toma decisões como escalonamento e balanceamento de carga, enquanto os Nodes executam os contêineres.

Figura 7 – *Cluster Kubernetes*



Fonte: (PATEL, 2021).

2.5.4.3 Nodes

O **Node** é a unidade física ou virtual que compõe o cluster. Cada Node possui um agente chamado *kubelet*, que se comunica com o Control Plane para garantir que os contêineres estejam executando conforme o especificado. Além disso, cada Node possui um runtime de contêiner (como Docker ou containerd) e componentes para monitoramento e comunicação em rede.

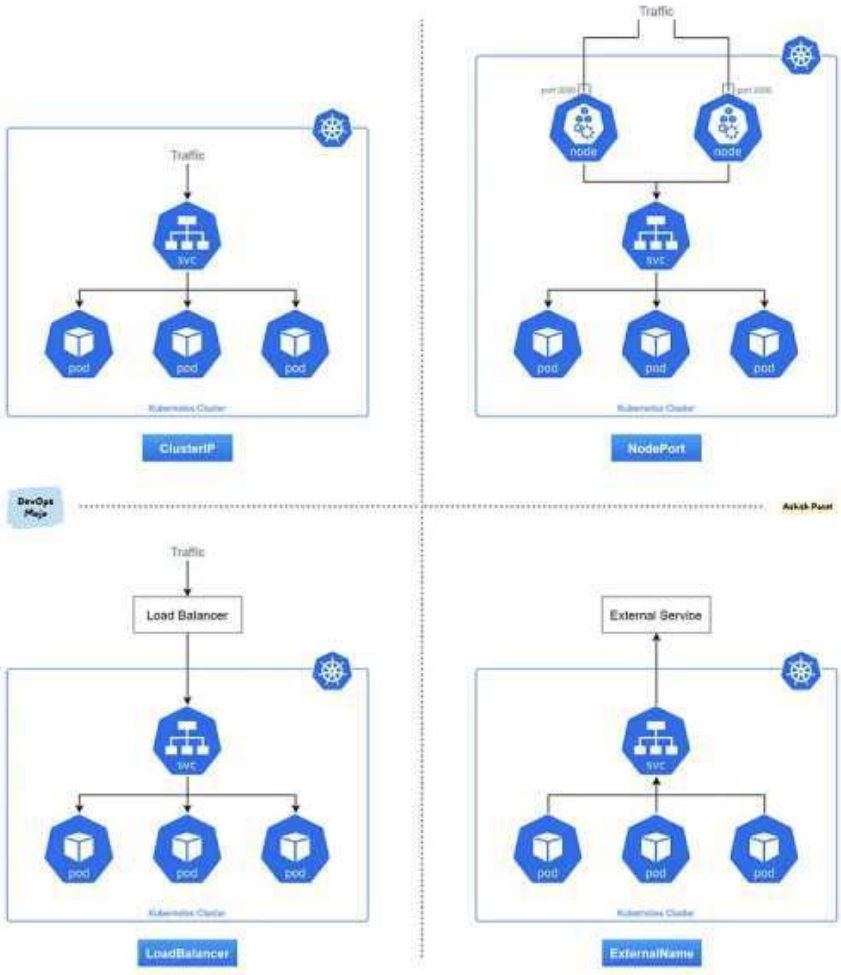
2.5.4.4 Deployments

O **Deployment** define o estado desejado para um conjunto de Pods, permitindo gerenciar atualizações de forma declarativa. Um Deployment é utilizado para criar e atualizar múltiplas réplicas de Pods e garantir que o número correto de réplicas esteja sempre em execução. Isso facilita a escalabilidade e a resiliência das aplicações.

2.5.4.5 Services

O **Service** é um objeto Kubernetes que define um ponto de acesso estável e confiável para um conjunto dinâmico de Pods. Ele possibilita a descoberta e a comunicação entre os componentes da aplicação, independentemente dos IPs individuais dos pods. Um Service pode ser de tipo *ClusterIP*, *NodePort* ou *LoadBalancer*, dependendo do escopo e das necessidades da aplicação.

Figura 8 – *Services Kubernetes*



Fonte: (PATEL, 2021).

2.5.4.6 Ingress

O **Ingress** é um objeto que gerencia o acesso externo aos serviços dentro de um cluster, oferecendo regras baseadas em host ou rota HTTP. Ele permite definir um único ponto de entrada para múltiplos serviços, facilitando o gerenciamento de tráfego e a configuração de SSL.

2.5.4.7 Load Balancer

O **Load Balancer** é um tipo de Service que distribui o tráfego de rede externo entre os diferentes Pods no cluster, garantindo alta disponibilidade e balanceamento de carga. Em ambientes de nuvem, a configuração de um Load Balancer geralmente resulta na criação automática de um balanceador de carga externo, permitindo exposição pública do serviço.

Esses objetos são componentes essenciais no Kubernetes, permitindo a construção de

sistemas distribuídos e escaláveis de maneira eficiente. O entendimento e uso correto desses recursos possibilitam a orquestração de aplicações complexas com alta disponibilidade, resiliência e simplicidade na gestão.

2.5.5 Integração de Inteligência Artificial em Ambientes Distribuídos

A integração de Inteligência Artificial (IA) em ambientes distribuídos, como aqueles que utilizam computação em borda e arquiteturas de microserviços, oferece novas possibilidades para o desenvolvimento de sistemas de telemedicina mais eficientes e funcionais. A IA permite que tarefas complexas, como diagnóstico assistido, análise de imagens médicas e monitoramento de sinais vitais, sejam realizadas de maneira automatizada, reduzindo a carga de trabalho dos profissionais de saúde e melhorando a qualidade do atendimento ao paciente (ESTEVA et al., 2019).

Em ambientes distribuídos, a IA pode ser implementada de forma descentralizada, onde algoritmos de aprendizado de máquina são executados diretamente em dispositivos de borda, como sensores e dispositivos médicos, ou em nós intermediários da rede. Essa abordagem permite que os dados sejam processados localmente, minimizando a necessidade de enviar grandes volumes de dados para servidores centrais na nuvem, o que reduz a latência e aumenta a velocidade de resposta (XU et al., 2020). Por exemplo, algoritmos de IA podem ser utilizados para analisar em tempo real dados de eletrocardiogramas (ECG) ou imagens de raios-X em clínicas remotas, permitindo diagnósticos rápidos e intervenções imediatas quando necessário.

Outra vantagem da integração de IA em ambientes distribuídos é a escalabilidade. Como os modelos de aprendizado de máquina podem ser treinados de forma centralizada e, posteriormente, implantados em vários nós da borda, é possível escalar o uso de IA em grande escala sem a necessidade de uma infraestrutura centralizada massiva. Isso é especialmente útil em cenários de saúde pública, como pandemias, onde há uma demanda repentina por recursos médicos e diagnósticos em massa (CONSORTIUM, 2021).

A segurança e privacidade dos dados são aprimoradas quando a IA é integrada a ambientes distribuídos. Em vez de enviar todos os dados para um servidor central, apenas os resultados processados ou dados anonimizados são transmitidos, minimizando o risco de exposição de dados sensíveis. Essa abordagem é fundamental para atender às regulamentações de proteção de dados, como a LGPD e o GDPR, garantindo que as informações dos pacientes sejam gerenciadas de forma segura (ROMAN; LOPEZ; MAMBO, 2018).

2.6 Redes Neurais e *Machine Learning*

Nos últimos anos, as redes neurais e o *Machine Learning* (ML) têm se consolidado como abordagens fundamentais para resolver problemas complexos de análise de dados, reconhecimento de padrões e previsões. Inspiradas na estrutura e funcionamento do cérebro humano, as redes neurais consistem em uma rede interconectada de neurônios artificiais que processam informações em várias camadas para realizar tarefas como classificação, regressão e agrupamento (GOODFELLOW; BENGIO; COURVILLE, 2016).

O campo de *Machine Learning* é vasto e abrange uma variedade de métodos que permitem que os computadores aprendam a partir dos dados, sem serem explicitamente programados para executar tarefas específicas (BISHOP, 2006). Dentre os modelos mais utilizados estão as redes neurais artificiais (*Artificial Neural Networks* - ANNs), as redes neurais convolucionais (*Convolutional Neural Networks* - CNNs), as redes neurais recorrentes (*Recurrent Neural Networks* - RNNs), e as redes neurais profundas (*Deep Neural Networks* - DNNs). Cada uma dessas arquiteturas possui características específicas que as tornam mais apropriadas para diferentes tipos de problemas.

No contexto médico, as redes neurais e os algoritmos de *Machine Learning* têm sido aplicados de maneira inovadora para diagnóstico automatizado, análise de imagens médicas, predição de doenças e monitoramento de pacientes em tempo real (SHICKEL; AL., 2017). A capacidade de aprender a partir de grandes volumes de dados clínicos e imagens médicas possibilita a criação de sistemas que auxiliam profissionais de saúde na tomada de decisões críticas.

2.6.1 Classificação de Séries Temporais

A classificação de séries temporais é uma área de crescente importância dentro do *Machine Learning*, especialmente para aplicações médicas. As séries temporais representam dados coletados ao longo do tempo, sendo utilizadas para monitorar padrões e tendências. Exemplos de séries temporais no contexto médico incluem sinais vitais (frequência cardíaca, pressão arterial), registros de eletrocardiograma (ECG) e dados de sensores utilizados no monitoramento remoto de pacientes (FAWAZ; AL., 2019).

Para a classificação de séries temporais, as redes neurais recorrentes (RNNs) e as suas variantes, como *Long Short-Term Memory* (LSTM) e *Gated Recurrent Units* (GRUs), se destacam pela capacidade de capturar dependências temporais de curto e longo prazo presentes nos dados (GREFF; AL., 2017). Em termos práticos, isso significa que esses modelos conseguem detectar padrões que se repetem ao longo do tempo, como anomalias no ritmo cardíaco ou mudanças significativas na respiração de um paciente.

Outra abordagem para a classificação de séries temporais inclui as redes neurais convolucionais (CNNs), que têm sido amplamente utilizadas para a análise de imagens e,

mais recentemente, para a análise de sinais temporais (ZHAO; AL., 2017). O uso de CNNs em séries temporais permite a identificação de padrões locais e características importantes, como picos de amplitude ou variações bruscas, que podem indicar anomalias ou eventos críticos no estado de saúde de um paciente.

A combinação de RNNs e CNNs tem se mostrado eficaz na classificação de séries temporais complexas. Esses modelos híbridos são capazes de capturar características tanto locais quanto globais dos dados, proporcionando um alto desempenho em tarefas como detecção de arritmias cardíacas a partir de sinais de ECG (RAJPURKAR; AL., 2017) e predição de crises epilépticas a partir de sinais de eletroencefalograma (EEG) (HOSSEINI; AL., 2016).

2.6.2 Aplicações Médicas de Redes Neurais

O uso de redes neurais e *Machine Learning* no campo da saúde tem sido um dos principais catalisadores para o desenvolvimento de tecnologias de suporte à decisão clínica. A seguir, são descritas algumas das principais aplicações:

- **Diagnóstico Automatizado:** Redes neurais convolucionais (CNNs) têm sido amplamente empregadas para a análise de imagens médicas, como raios-X, tomografias e ressonâncias magnéticas. Esses modelos conseguem identificar automaticamente anomalias e padrões associados a diferentes condições médicas, como tumores, fraturas e doenças pulmonares (ESTEVA; AL., 2017).
- **Análise de Sinais Fisiológicos:** Sinais de ECG e EEG são comumente analisados utilizando RNNs e LSTMs para detectar padrões irregulares, como arritmias cardíacas ou crises epilépticas, com um alto grau de precisão (HANNUN; AL., 2019).
- **Preditivos de Doenças:** Modelos de *Deep Learning* têm sido aplicados para prever a probabilidade de desenvolvimento de doenças crônicas com base em fatores como idade, histórico médico e dados genéticos (MIOTTO; AL., 2016). Esses modelos ajudam a identificar pacientes em risco e a implementar estratégias de intervenção precoce.
- **Monitoramento Remoto de Pacientes:** Sensores vestíveis e dispositivos de Internet das Coisas (IoT) coletam dados em tempo real sobre o estado de saúde dos pacientes. Esses dados são então analisados por redes neurais para identificar eventos críticos, como quedas, mudanças de postura ou deterioração dos sinais vitais (SHICKEL; AL., 2017).

O avanço contínuo em redes neurais e *Machine Learning* tem o potencial de transformar significativamente o campo da saúde, proporcionando uma medicina mais

precisa e personalizada. A capacidade dessas tecnologias de aprender com dados e se adaptar a novos contextos clínicos promete um futuro onde diagnósticos mais rápidos e precisos e tratamentos mais eficazes estarão ao alcance dos profissionais de saúde e pacientes.

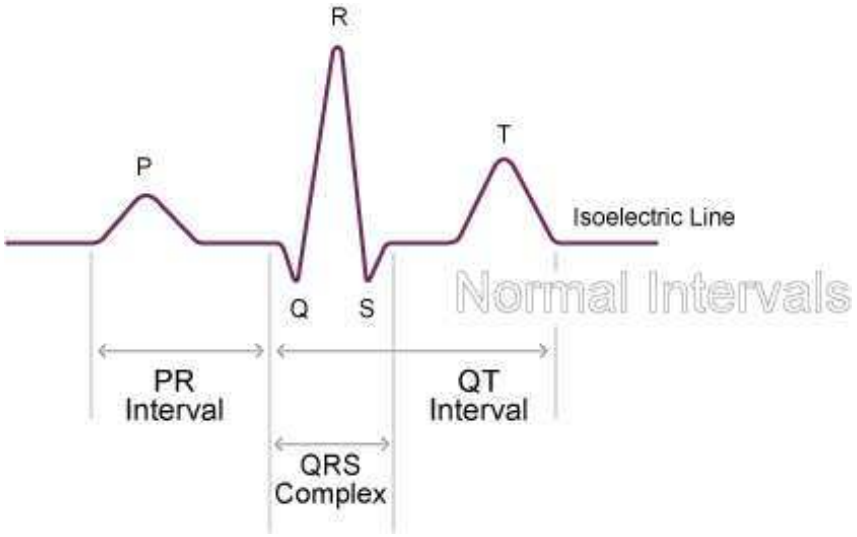
2.7 ECG e Anomalias

O eletrocardiograma (ECG) é uma técnica amplamente utilizada na prática médica para registrar a atividade elétrica do coração ao longo do tempo. O ECG fornece informações essenciais sobre a condição cardíaca de um paciente, permitindo a identificação de distúrbios como arritmias, bloqueios cardíacos e isquemias (ACHARYA; AL., 2007). As ondas registradas no ECG são categorizadas em várias componentes, incluindo ondas P, QRS e T, cada uma correspondendo a diferentes fases do ciclo cardíaco. A análise dessas ondas e seus intervalos permite aos cardiologistas inferir a funcionalidade e a saúde do coração.

2.7.1 Características do Sinal de ECG

O sinal de ECG é composto por várias ondas distintas, cada uma representando uma fase do ciclo de contração e relaxamento cardíaco, que podem ser vistas na Figura 9. A seguir, são descritas as principais ondas e intervalos do ECG (KLIGFIELD; AL., 2007):

Figura 9 – Fases ECG



Fonte: (NOTTINGHAM, 2024).

- **Onda P:** Representa a despolarização atrial, que é o processo de ativação elétrica dos átrios.

- **Complexo QRS:** Corresponde à despolarização dos ventrículos, indicando o início da contração ventricular. Este é o segmento mais significativo do ECG, pois reflete a maior parte da atividade elétrica do coração.
- **Onda T:** Indica a repolarização dos ventrículos, ou seja, a recuperação elétrica que prepara o coração para a próxima contração.
- **Intervalo PR:** Representa o tempo que o impulso elétrico leva para viajar dos átrios aos ventrículos.
- **Intervalo QT:** É o tempo necessário para que os ventrículos se despolarizem e se repolarizem.

Essas ondas e intervalos fornecem informações valiosas para o diagnóstico de anomalias cardíacas. Um desvio nos valores esperados para qualquer um desses elementos pode indicar uma condição clínica que requer atenção.

2.7.2 Detecção de Anomalias no ECG

Anomalias no ECG, como arritmias, são marcadores importantes para a detecção precoce de doenças cardiovasculares. A arritmia é caracterizada por um ritmo cardíaco irregular, e pode ser classificada como taquicardia (ritmo rápido), bradicardia (ritmo lento) ou fibrilação (ritmo desorganizado) (GOLDBERGER; AL., 2008). A identificação manual dessas anomalias, especialmente em longos registros de ECG, pode ser demorada e propensa a erros. Nesse contexto, técnicas automatizadas baseadas em aprendizado de máquina e redes neurais profundas têm demonstrado um alto grau de acurácia na detecção e classificação de arritmias e outras anomalias cardíacas (HANNUN; AL., 2019).

Figura 10 – Exemplo de um sinal de ECG com uma arritmia (fibrilação atrial)

Fonte: Adaptado de (RAJPURKAR; AL., 2017).

As redes neurais convolucionais (CNNs) e as redes neurais recorrentes (RNNs) têm sido amplamente empregadas para a análise de sinais de ECG. As CNNs são utilizadas para extrair características importantes do sinal, como variações de amplitude e frequências específicas que indicam a presença de anomalias (ACHARYA; AL., 2017). Por outro lado, as RNNs, incluindo LSTMs e GRUs, são particularmente eficazes em capturar dependências temporais e padrões recorrentes em sinais de longa duração (MARTIS; AL., 2013).

Esses modelos têm sido aplicados para detectar uma variedade de anomalias, incluindo:

- **Fibrilação Atrial:** Caracterizada por um ritmo desorganizado e irregular. É uma das causas mais comuns de acidente vascular cerebral (AVC) (LAU; AL., 2013).
- **Taquicardia Ventricular:** Um ritmo cardíaco acelerado originado nos ventrículos, que pode levar a uma parada cardíaca súbita.
- **Bloqueios Cardíacos:** Ocorrem quando há um atraso ou bloqueio na condução do impulso elétrico entre os átrios e ventrículos. Podem ser observados por alterações no intervalo PR.
- **Síndrome do QT Longo:** Uma condição genética ou adquirida que prolonga o intervalo QT, aumentando o risco de arritmias perigosas (GOLDENBERG; AL., 2008).

2.7.3 Aplicações de *Machine Learning* para Detecção de Anomalias no ECG

Com a crescente disponibilidade de grandes bases de dados de ECG e avanços em *Machine Learning*, técnicas baseadas em aprendizado profundo estão sendo amplamente empregadas para a detecção automatizada de anomalias no ECG (RAJPURKAR; AL., 2017). Modelos de *Deep Learning* têm se mostrado superiores às técnicas tradicionais devido à capacidade de aprender automaticamente características discriminantes a partir dos dados, sem a necessidade de intervenção manual.

Um exemplo notável é o uso de CNNs para a detecção de fibrilação atrial e outras arritmias, atingindo um desempenho comparável ao de cardiologistas experientes (HANNUN; AL., 2019). Outro exemplo é a utilização de redes LSTM para previsão de eventos cardíacos adversos, como a ocorrência de taquicardia ventricular, com alta acurácia e sensibilidade (ZHAO; AL., 2017).

Essas técnicas não apenas aceleram o processo de análise, mas também contribuem para a redução de erros humanos e a identificação precoce de condições críticas, melhorando a eficácia do diagnóstico e do tratamento em contextos clínicos (ZHANG; AL., 2019).

3 Metodologia

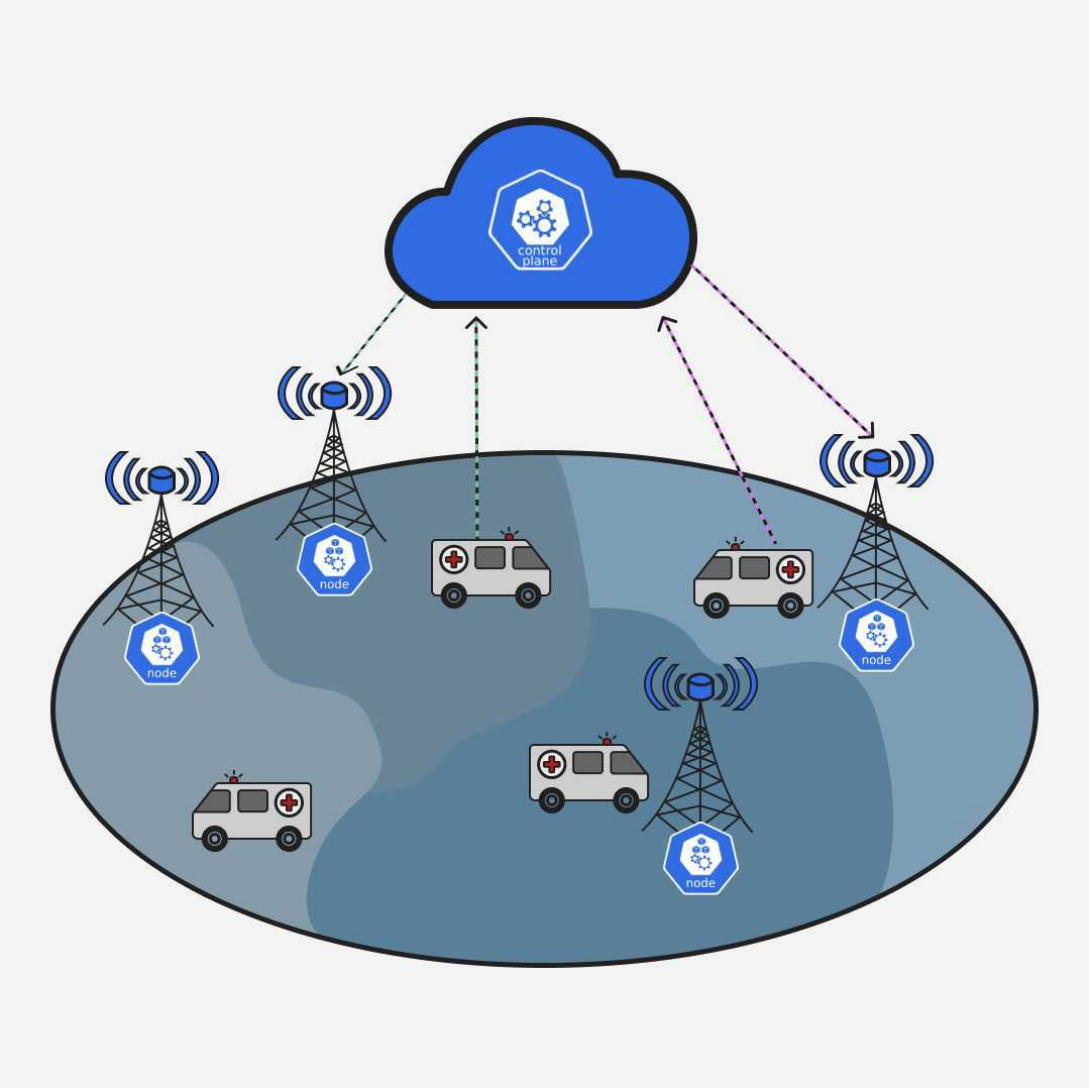
Considerando as necessidades emergentes de serviços de telemedicina, conforme apresentado anteriormente, foi identificada a demanda por uma arquitetura distribuída capaz de proporcionar consultas virtuais eficientes e diagnósticos assistidos por inteligência artificial. Assim, o foco principal deste trabalho constituiu em propor e implementar uma solução que permita a integração de serviços médicos virtualizados, utilizando técnicas de orquestração de contêineres e computação.

A arquitetura proposta neste trabalho foi desenvolvida com base na combinação de tecnologias de computação distribuída e orquestração de contêineres, que permite a criação de um sistema dinâmico e escalável. Essa abordagem teve como foco otimizar a distribuição de recursos computacionais e garantir que os serviços, como consultas virtuais e diagnósticos por IA, sejam entregues com baixa latência e alta disponibilidade, especialmente em ambientes distribuídos.

Nesse contexto, foi realizado uma revisão bibliográfica visando aprofundar o entendimento sobre sistemas de orquestração, como o Kubernetes, que se mostraram altamente eficazes na gestão de clusters distribuídos e na garantia da continuidade dos serviços em diversos ambientes. Além disso, foram analisadas as aplicações desses sistemas no setor de saúde, avaliando como eles têm sido utilizados e como poderiam ser adaptados para este projeto. Também, foi investigada a viabilidade de implementar essas arquiteturas em um ambiente local, simulado para reproduzir condições reais de operação, sem incorrer nos custos elevados de um ambiente de produção.

Foi desenvolvida uma arquitetura distribuída em um ambiente local para simular um sistema inteligente de ambulâncias, capaz de utilizar serviços de Inteligência Artificial (IA) para otimizar o atendimento ao paciente. A ilustração do problema simulado é apresentada na Figura 11. Nesse cenário, as ambulâncias comunicavam-se com uma aplicação central que, por meio de um mecanismo de orquestração, direcionava as solicitações ao servidor mais adequado, garantindo uma resposta eficiente e rápida.

Figura 11 – Ambiente Simulado Localmente



Fonte: Produzida pelo autor

Para implementar essa solução, foi configurado um cluster Kubernetes em máquinas virtuais, onde cada máquina virtual representava um servidor distinto dentro do sistema. O cluster incluía um broker MQTT para gerenciar a recepção e a distribuição dos dados recebidos, além das aplicações baseadas em inteligência artificial para processamento e tomada de decisão.

Como parte da simulação de um ambiente médico, foi desenvolvida uma aplicação para a classificação de sinais de ECG (eletrocardiograma) em tempo real, utilizando técnicas de *machine learning*. Assim, o fluxo completo de simulação consistia no envio dos dados de ECG para o cluster Kubernetes, que processava as informações através da aplicação de IA e respondia com a classificação apropriada dos sinais, representando o diagnóstico automatizado do paciente.

4 Materiais e Ferramentas

Neste capítulo, são descritos os materiais e ferramentas empregadas no desenvolvimento da arquitetura distribuída para o cenário de telemedicina, que visa proporcionar consultas virtuais e diagnósticos assistidos por inteligência artificial. O foco deste capítulo é justificar as tecnologias e plataformas escolhidas, descrevendo suas funcionalidades e adequação ao contexto do projeto.

4.1 Ambiente de Testes

Para simular o ambiente de produção, foi configurado um conjunto de cinco máquinas virtuais utilizando o *VirtualBox*. Essas máquinas formaram um cluster, permitindo que os componentes da arquitetura fossem distribuídos e orquestrados em um ambiente controlado. O uso de máquinas virtuais se mostrou adequado para replicar um cenário realista de computação distribuída, permitindo o controle total dos recursos alocados sem os custos adicionais de um ambiente físico (PAHL et al., 2019).

4.2 Orquestração com Kubernetes

O Kubernetes foi escolhido como a principal ferramenta de orquestração de contêineres no cluster. A escolha do Kubernetes se baseia em sua capacidade amplamente reconhecida de gerenciar cargas de trabalho distribuídas, escalabilidade automática e resiliência em ambientes críticos, como os exigidos por sistemas de telemedicina (BURNS; BEDA; HIGHTOWER, 2016). Além disso, sua ampla adoção na indústria de tecnologia garante um ecossistema maduro e estável para o gerenciamento de contêineres.

A arquitetura do Kubernetes inclui diversos componentes essenciais para o gerenciamento eficiente de microserviços distribuídos, como o *etcd* (para armazenamento de configurações), o *kube-apiserver*, *kube-scheduler* e *kube-controller-manager*, que garantem a distribuição e execução otimizada das cargas de trabalho no cluster (HIGHTOWER; BURNS; BEDA, 2020).

4.3 MetalLB e NGINX

Para o balanceamento de carga e gestão de entradas e saídas no cluster, foram utilizadas as tecnologias MetalLB e NGINX. O MetalLB foi configurado como *load balancer*, permitindo a distribuição eficiente do tráfego entre os nós do cluster. Essa escolha foi feita por sua facilidade de integração com clusters baseados em Kubernetes, além de oferecer

suporte em ambientes que não possuem balanceadores de carga nativos, como é o caso do ambiente local utilizado (METALLB. . . , 2021).

O NGINX, por sua vez, foi utilizado como controlador de *ingress*, permitindo a gestão de acessos externos ao cluster, facilitando a comunicação entre os serviços da arquitetura e os dispositivos dos usuários finais. O NGINX é amplamente reconhecido por sua alta performance e baixa latência, características essenciais para garantir uma experiência fluida em consultas virtuais e diagnósticos em tempo real (FRAMPTON, 2018).

4.4 Broker MQTT com Mosquitto

Para gerenciar a comunicação de dados entre os dispositivos de telemetria e o sistema de diagnóstico assistido por IA, foi utilizado o *broker* MQTT Mosquitto. O MQTT é um protocolo leve, ideal para cenários onde é necessária a transmissão de grandes quantidades de dados em tempo real, como sinais de ECG no contexto de telemedicina (GUTH et al., 2016). O Mosquitto, como um *broker* de código aberto, se destaca pela sua eficiência e suporte a dispositivos com recursos limitados, além de ser facilmente integrável ao Kubernetes (LIGHT, 2017).

4.5 Classificação de ECG com TensorFlow

Para a classificação em tempo real dos sinais de ECG, foi desenvolvido um modelo de rede neural utilizando o *TensorFlow*, uma biblioteca amplamente utilizada para aprendizado de máquina e redes neurais. A arquitetura do modelo foi baseada em técnicas de classificação de séries temporais, adequadas para a análise de sinais biomédicos (SANTOS et al., 2019). O modelo foi implementado em um contêiner Docker e distribuído no cluster Kubernetes, garantindo que a classificação fosse realizada de maneira escalável e eficiente (ABADI et al., 2016).

4.6 Docker Hub e Deployments no Cluster

O Docker Hub foi utilizado como repositório para a imagem do contêiner que executa o código de classificação de ECG. A escolha pelo Docker Hub se justifica pela facilidade de distribuição de contêineres em diferentes ambientes e pela compatibilidade direta com o Kubernetes. O *deployment* foi configurado para garantir a escalabilidade horizontal, ou seja, a criação de réplicas do serviço de classificação conforme a demanda por consultas virtuais aumentasse, mantendo a qualidade do serviço prestado (MERKEL, 2014).

4.7 Instalação do VirtualBox e Configuração de Rede

O primeiro passo foi a instalação do *VirtualBox*, uma ferramenta de virtualização que permite a criação e execução de múltiplos sistemas operacionais em um único computador. O *VirtualBox* é amplamente utilizado em ambientes de teste e desenvolvimento devido à sua flexibilidade e suporte a diferentes sistemas operacionais.

Para compor o cluster Kubernetes, foram criadas cinco máquinas virtuais (VMs) usando o *VirtualBox*. A imagem do Ubuntu 20.04.6 LTS foi escolhida como sistema operacional para todas as VMs devido à sua estabilidade e popularidade em ambientes de produção. A primeira máquina virtual foi configurada como *master-node*, utilizando a interface de rede em modo *bridge*, o que permite que a VM se conecte diretamente à rede física do host e obtenha um endereço IP na mesma sub-rede, essencial para a comunicação entre os nós do cluster.

4.8 Configuração e Preparação da Máquina Virtual

Após a criação da VM *master-node*, o ambiente foi preparado utilizando os seguintes comandos:

```
sudo apt update -y: Atualiza a lista de pacotes disponiveis.
sudo apt install ssh: Instala o servidor SSH .
systemctl status ssh: Verifica se o servico SSH esta em execucao.
```

Além disso, o arquivo `/etc/hosts` foi editado para incluir os endereços IP e nomes das VMs, facilitando a comunicação entre os nós do cluster. Isso foi necessário para permitir que os nós se referissem uns aos outros por nomes amigáveis, em vez de endereços IP.

```
/etc/hosts:

192.168.15.40    master-node
192.168.15.41    worker-node-1
192.168.15.42    worker-node-2
192.168.15.43    worker-node-3
192.168.15.44    worker-node-4
```

4.9 Configuração dos Nós: Kubeadm, Kubelet e Kubectl

O próximo passo foi configurar os nós do cluster, utilizando as ferramentas *containerd*, *kubeadm*, *kubelet* e *kubectl*, que são essenciais para a operação do Kubernetes. A instalação seguiu o guia do repositório *usianej/kubernetes*, que fornece um passo a passo para configurar um ambiente Kubernetes a partir do zero.

Após a configuração inicial do *master-node*, foi criado um *snapshot* da máquina, permitindo que ela fosse clonada quatro vezes para formar os nós *worker*. Cada nó foi ajustado com um novo endereço IP e *hostname*, conforme necessário.

4.10 Configuração do Control Plane

Para configurar o *control plane* do cluster, foi necessário acessar o *master-node* via SSH e seguir as instruções do guia mencionado anteriormente. Após a configuração, o *kubeadm* gerou um comando de *join*, que foi utilizado em cada um dos *worker nodes* para conectá-los ao cluster.

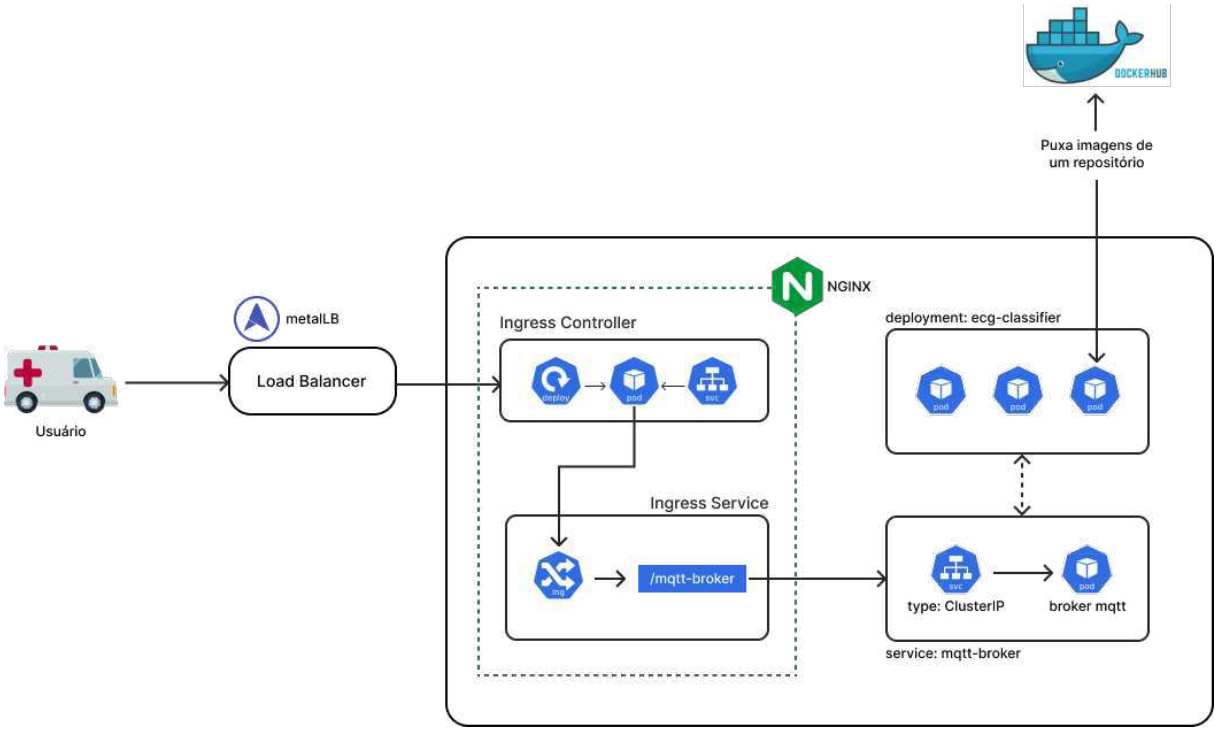
5 Arquitetura

Este capítulo detalha a arquitetura proposta que teve como objetivo uma comco-municação organizada e segura entre os componentes, utilizando balanceamento de carga, controle de acesso via NGINX e mensageria MQTT.

5.1 Arquitetura Proposta

A arquitetura implementada (Figura 12) envolve a configuração de um ambiente Kubernetes (K8s) para uma aplicação baseada em serviços. Nela, o usuário acessa o sistema através de um balanceador de carga externo, configurado com o MetalLB, que distribui o tráfego de entrada entre os componentes internos do cluster. Esse balanceador direciona o tráfego para o Ingress Controller, que atua como ponto de entrada central para a comunicação externa, possibilitando o roteamento das requisições para diferentes serviços internos do cluster.

Figura 12 – Arquitetura Implementada



Fonte: Produzida pelo autor

O Ingress Controller, utilizando *NGINX*, gerencia o tráfego de maneira organizada.

Dentro do cluster, há um serviço de *Ingress* configurado para definir regras específicas de roteamento. Por exemplo, uma rota específica foi configurada para um serviço MQTT, acessível através do caminho "/mqtt-broker". Esse serviço de Ingress permite que o tráfego seja direcionado de forma apropriada para os serviços internos, garantindo uma comunicação eficiente.

Além disso, a arquitetura inclui um *broker MQTT*, que está configurado como um serviço do tipo *ClusterIP*, o que significa que ele é acessível apenas internamente no *cluster*. Esse *broker* é utilizado para comunicação assíncrona de mensagens, facilitando a troca de dados entre componentes. A aplicação principal, chamada "ecg-classifier", é implementada como um *deployment* no *Kubernetes* e tem como objetivo realizar a classificação de sinais de eletrocardiograma (ECG). As imagens de contêiner necessárias para esses componentes são obtidas de um repositório Docker, garantindo que as versões e dependências sejam gerenciadas de forma consistente.

5.2 Configuração do Load Balancer MetalLB

Uma vez configurado o cluster Kubernetes, foi necessário adicionar um *load balancer* para balancear o tráfego entre os nós. Como o ambiente utilizado é *bare-metal* (sem um *load balancer* fornecido por provedores de nuvem), foi utilizado o *MetalLB*, que é uma solução de software para essa função.

A configuração do *MetalLB* envolveu a criação de um *pool* de endereços IP disponíveis para o balanceamento. Um arquivo YAML denominado `pool-1.yml` foi criado com o seguinte conteúdo:

```
1 apiVersion: metallb.io/v1beta1
2 kind: IPAddressPool
3 metadata:
4    name: first-pool
5    namespace: metallb-system
6 spec:
7     addresses:
8     - 192.168.15.140-192.168.15.150
```

Esse arquivo foi aplicado ao cluster utilizando o comando `kubectl apply`. Em seguida, foi criado um arquivo YAML para anunciar o uso do *pool* de IPs no modo L2:

```
1 apiVersion: metallb.io/v1beta1
2 kind: L2Advertisement
3 metadata:
4    name: homelab-l2
5    namespace: metallb-system
```

```
spec:
    ipAddressPools:
    - first-pool
```

Este arquivo também foi aplicado com `kubectl apply`, configurando o *load balancer* para distribuir o tráfego de forma eficiente entre os nós.

5.3 Instalação e Configuração do NGINX Ingress Controller

O *NGINX Ingress Controller* foi instalado para gerenciar as entradas e saídas de tráfego no cluster. A instalação seguiu a documentação oficial do NGINX por meio do método de *installation by manifest*. O *Ingress Controller* permite que serviços externos se conectem ao cluster, facilitando o roteamento de requisições aos serviços corretos.

5.4 Adaptando o NGINX para Funcionar com Protocolo TCP

Um dos maiores desafios encontrados foi adaptar o *NGINX Ingress* para trabalhar com serviços que utilizam o protocolo TCP, como o *broker* MQTT. Por padrão, o *NGINX Ingress* é configurado para lidar com tráfego HTTP, o que exigiu a criação de um *TransportServer* para suportar TCP.

Foi criado o arquivo `transport-server-mqtt.yaml`, conforme mostrado abaixo:

```
apiVersion: k8s.nginx.org/v1alpha1
kind: TransportServer
metadata:
  name: mqtt-tcp
  namespace: default
spec:
  listener:
    name: mqtt-tcp
    protocol: TCP
    upstreams:
      name: mqtt-broker
      service: mosquitto
      port: 1883
      action:
        pass: mqtt-broker
```

Além disso, foi necessário criar uma configuração global para o *NGINX*, especificando o uso do protocolo TCP na porta 1883. Esse arquivo foi denominado global-configuration.yaml:

```
apiVersion: k8s.nginx.org/v1
kind: GlobalConfiguration
metadata:
  name: nginx-configuration
  namespace: nginx-ingress
spec:
  listeners:
  - name: mqtt-tcp
    port: 1883
    protocol: TCP
```

Ambos os arquivos foram aplicados ao cluster usando `kubectl apply`.

5.5 Criação do Service para o Broker MQTT

O *broker* MQTT foi configurado como um serviço dentro do cluster. Para isso, foi criado o arquivo YAML `mosquitto.yaml`, que define um *service*, um *ConfigMap* e um *deployment* para o Mosquitto:

```
apiVersion: v1
kind: Service
metadata:
  name: mosquitto
spec:
  type: ClusterIP
  selector:
    app: mosquitto
  ports:
  - port: 1883
    targetPort: 1883
    protocol: TCP
    name: mqtt
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: mosquitto-config
data:
  mosquitto.conf: |
    allow_anonymous true

    listener 1883
```

```
24      protocol mqtt
25  ---
26  apiVersion: apps/v1
27  kind: Deployment
28  metadata:
29    name: mosquitto
30  spec:
31    replicas: 1
32    selector:
33      matchLabels:
34        app: mosquitto
35    template:
36      metadata:
37        labels:
38          app: mosquitto
39      spec:
40        containers:
41        - name: mosquitto
42          image: eclipse-mosquitto
43          ports:
44          - containerPort: 1883
45          volumeMounts:
46          - mountPath: /mosquitto/config/mosquitto.conf
47            subPath: mosquitto.conf
48            name: config
49        volumes:
50        - name: config
51          configMap:
52            name: mosquitto-config
```

5.6 Aplicação de Classificação de ECG

Com o *broker* MQTT exposto pelo *NGINX Ingress* e validado o próximo passo foi fazer a aplicação em si, que dependia do *broker*. Assim, foi decidido que a aplicação seria a classificação de sinais de ECG. Assim, o primeiro passo foi encontrar um dataset que seria usado como base para o treinamento de um modelo. Foi então usado o *MIT-BIH Arrhythmia Database* que contém 48 trechos de 30 minutos de gravações de ECG de dois canais, coletadas de 47 pacientes entre 1975 e 1979. (MOODY; MARK, 2001)

Os dados foram separados em 109446 trechos de 188 colunas, cada coluna representando um dado coletado do ECG. importante ressaltar que os dados possuem uma

frequencia de amostragem de 125Hz. Após isso, Aconteceu o treinamento do modelo. Inicialmente, os dados de treino e teste foram carregados a partir de arquivos CSV, com a normalização dos valores de entrada e a conversão dos rótulos para o formato categórico. Em seguida, um modelo CNN foi construído com camadas de convolução, pooling e densas, otimizando a saída para 5 classes utilizando uma função de ativação softmax. Os dados de entrada foram preparados para a estrutura correta de convolução, com a adição de um canal extra, e foram divididos entre os conjuntos de treino e validação. O modelo foi treinado usando o otimizador Adam e a função de perda categorical crossentropy. Após o treinamento, o modelo foi avaliado no conjunto de teste, e a precisão foi exibida. Por fim, o modelo treinado foi salvo para uso posterior. Abaixo podemos visualizar o código utilizado:

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv1D, MaxPooling1D, Flatten
    , Dense
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

def load_data(train_file, test_file):
    train_data = pd.read_csv(train_file, header=None).values
    test_data = pd.read_csv(test_file, header=None).values

    X_train, y_train = train_data[:, :-1], train_data[:, -1].
        astype(int)
    X_test, y_test = test_data[:, :-1], test_data[:, -1].astype(
        int)

    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    y_train = to_categorical(y_train, num_classes=5)
    y_test = to_categorical(y_test, num_classes=5)

    return (X_train, y_train), (X_test, y_test)

def create_model(input_shape):
    model = Sequential()
```

```
28      model.add(Conv1D(filters=32, kernel_size=5,
29      activation='relu', input_shape=input_shape))
30      model.add(MaxPooling1D(pool_size=2))
31      model.add(Conv1D(filters=64, kernel_size=5, activation='relu'
            ))
32      model.add(MaxPooling1D(pool_size=2))
33      model.add(Flatten())
34      model.add(Dense(128, activation='relu'))
35      model.add(Dense(5, activation='softmax'))
36
37      model.compile(optimizer='adam', loss='
            categorical_crossentropy',
38      metrics=['accuracy'])
39
40      return model
41
42  def prepare_data(X, y):
43      X = np.expand_dims(X, axis=2)
44      return X, y
45
46  def train_model(train_data, test_data, batch_size=64, epochs=10):
47
48      X_train, y_train = prepare_data(*train_data)
49      X_test, y_test = prepare_data(*test_data)
50
51      X_train, X_val, y_train, y_val = train_test_split(X_train,
            y_train,
52      test_size=0.2, random_state=42)
53
54      model = create_model(input_shape=(X_train.shape[1], 1))
55
56      history = model.fit(X_train, y_train, epochs=epochs,
57      batch_size=batch_size, validation_data=(X_val, y_val))
58
59      test_loss, test_accuracy = model.evaluate(X_test, y_test,
            verbose=0)
60      print(f"Test accuracy: {test_accuracy * 100:.2f}%")
61
62      return model
63
64  def save_model(model, filename='ecg_classifier.h5'):
65      model.save(filename)
```

```
66
67 def main():
68
69     train_file = 'mitbih_train.csv'
70     test_file = 'mitbih_test.csv'
71
72     train_data, test_data = load_data(train_file, test_file)
73
74     model = train_model(train_data, test_data, batch_size=64,
           epochs=10)
75
76     save_model(model, filename='/app/model/ecg_classifier.h5')
77
78 if __name__ == '__main__':
79     main()
```

Com o modelo gerado, o próximo passo foi elabora de fato a aplicação. Um modelo de rede neural pré-treinado foi utilizado para classificar dados de ECG em tempo real, recebidos via o protocolo MQTT. O modelo foi carregado inicialmente e utilizado para prever a classe dos dados de ECG processados em lotes. A aplicação conectou-se a um broker MQTT, recebendo mensagens de dados em um tópico específico. Esses dados foram armazenados até que um lote suficiente fosse formado, momento em que o lote foi processado, e a previsão de classe (como "Normal", "Supraventricular", etc.) foi publicada em um tópico de resposta. A conexão com o broker foi realizada com tentativas automáticas de reconexão em caso de falhas, garantindo a resiliência da aplicação. Abaixo podemos visualizar o código utilizado:

```
1  import paho.mqtt.client as mqtt
2  import numpy as np
3  import tensorflow as tf
4  import time
5
6  def load_keras_model(filename='ecg_classifier.h5'):
7      model = tf.keras.models.load_model(filename)
8      return model
9
10 def predict_single_input(model, input_data):
11     input_data = np.reshape(input_data,
12     (1, input_data.shape[0], 1)).astype(np.float32)
13     predictions = model.predict(input_data, verbose=0)
14     predicted_class = np.argmax(predictions, axis=1)[0]
15     return predicted_class
```

```
16
17 class_labels = {
18     0: "Normal",
19     1: "Supraventricular",
20     2: "Ventricular",
21     3: "Fusion",
22     4: "Unknown"
23 }
24
25 def process_batch(batch):
26     input_array = np.array(batch).astype(np.float32)
27     predicted_class = predict_single_input(model=loaded_model,
28     input_data=input_array)
29     class_label = class_labels.get(predicted_class, "Unknown")
30     return class_label
31
32 def on_message(client_, userdata, message):
33     global ecg_data
34
35     try:
36         topic = message.topic
37         client_id = topic.split('/')[1]
38
39         payload = message.payload.decode()
40         print(f"Payload recebido: {payload}")
41
42         try:
43             new_data = float(payload)
44             print(f"Convertido para float: {new_data}")
45             ecg_data.append(new_data)
46
47             if len(ecg_data) >= batch_size:
48                 current_batch = ecg_data[-batch_size:]
49                 class_label = process_batch(current_batch)
50
51                 response_topic = f"resposta/{client_id}"
52                 print(f"Classificacao: {class_label}")
53                 client.publish(response_topic, class_label)
54
55                 ecg_data = []
56
57         except ValueError as ve:
```

```
58              print(f"Erro ao converter o payload para float: {
                    payload}.
59              Erro: {ve}")
60
61      except Exception as e:
62          print(f"Erro na mensagem recebida: {e}")
63
64  def on_connect(client, userdata, flags, reason_code, properties=
        None):
65      if reason_code == 0:
66          print("Conectado ao broker MQTT com sucesso.")
67      else:
68          print(f"Falha na conexao. Codigo de retorno: {reason_code
                }")
69
70  def connect_with_retries(client, host, port, max_retries=5, delay
        =10):
71      for attempt in range(1, max_retries + 1):
72          try:
73              client.connect(host, port)
74              print("Connected to MQTT broker.")
75              return True
76          except Exception as e:
77              print(f"Tentativa numero {attempt} de se conectar
                    falhada.
78              Erro: {e}")
79              if attempt < max_retries:
80                  time.sleep(delay)
81              else:
82                  print("Maximo de tentativas de conexao atingido."
                        )
83                  return False
84
85  ecg_data = []
86  batch_size = 187
87
88  loaded_model = load_keras_model('ecg_classifier.h5')
89
90  client = mqtt.Client(callback_api_version=
91  mqtt.CallbackAPIVersion.VERSION2)
92  client.on_message = on_message
93  client.on_connect = on_connect
```

```
94
95 client.reconnect_delay_set(min_delay=1, max_delay=60)
96
97 mqtt_host = "mosquitto.default.svc.cluster.local"
98 mqtt_port = 1883
99
100 if connect_with_retries(client, mqtt_host, mqtt_port):
101     topic = "dados/#"
102     client.subscribe(topic)
103     client.loop_forever()
```

Após isso, foi gerado uma imagem docker com esse código. Com isso, foi possível enviar a imagem da aplicação para o DockerHub. Que posteriormente seria consumida pela aplicação dentro do cluster kubernetes.

Com a aplicação em uma imagem docker no repositório público do DockerHub tudo que restou foi criar uma objeto kubernetes do tipo deployment dentro do cluster para consumir essa imagem. Assim, o seguinte arquivo YAML foi gerado:

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: mqtt-predictor
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: mqtt-predictor
10   template:
11     metadata:
12       labels:
13         app: mqtt-predictor
14     spec:
15       containers:
16       - name: mqtt-predictor
17         image: vitortrindader/mqtt-processor:latest
```

6 Prova de Conceito e Resultados

Este capítulo descreve os resultados obtidos com a implementação da arquitetura de computação em borda para a classificação de sinais de ECG em tempo real, utilizando um cluster Kubernetes, broker MQTT, e um modelo de rede neural convolucional treinado com dados de ECG. Também serão discutidos os desafios enfrentados e as soluções implementadas para garantir o funcionamento correto da aplicação.

6.1 Criação do Cluster Kubernetes

Para configuração do cluster Kubernetes foram instalados as ferramentas Kubelet, kubeadm e kubctl em cinco máquinas virtuais, sendo quatro delas *worker nodes* e uma delas o *master node*. A configuração de um cluster Kubernetes minimamente funcional termina quando os *worker nodes* se conectam ao *master node* através de um comando de *join* gerado no *master node*. Uma forma simples de verificar se tudo ocorreu com sucesso é executar o comando *"kubectl get nodes"* dentro do *control plane*, é esperado que todos os nós estejam com status *ready*. A Figura 13 ilustra esse momento de validação de *cluster*.

Figura 13 – Validação do Funcionamento Inicial do Cluster Kubernetes

```
vitor@master-node:~$ kubectl get nodes
NAME            STATUS   ROLES           AGE   VERSION
node            Ready    control-plane   46d   v1.28.13
worker-node-1   Ready    <none>          46d   v1.28.13
worker-node-2   Ready    <none>          46d   v1.28.13
worker-node-3   Ready    <none>          44d   v1.28.13
worker-node-4   Ready    <none>          44d   v1.28.13
```

Fonte: Produzida pelo autor

6.2 Implementação do *NGINX Ingress Controller*

O *NGINX Ingress Controller* tem o objetivo de gerenciar as entradas e saídas de tráfego no cluster. Para instalação do *Ingress* foi seguido a documentação oficial do *NGINX*. Assim, como forma de verificar a instalação era esperado que tivesse sido criado dois principais objetos: o *NGINX Ingress Controller* e o LoadBalancer do *NGINX*. Para isso foram executados os seguintes comandos:

```
1 kubectl get pods -n nginx-ingress
2 kubectl get services -n nginx-ingress
```

A Figura 14 mostra o resultado obtido após a execução dos comandos acima, indicando que o *NGINX Ingress Controller* foi bem instalado.

Figura 14 – Verificação da Instalação *NGINX*

```
vitor@master-node:~$ kubectl get pods -n nginx-ingress
NAME                             READY   STATUS    RESTARTS      AGE
nginx-ingress-794bb556dd-x9jst   1/1     Running   8 (23m ago)   22d
vitor@master-node:~$ kubectl get services -n nginx-ingress
NAME            TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)                                   AGE
nginx-ingress   LoadBalancer   10.106.153.80   192.168.15.140   80:31728/TCP,443:31229/TCP,1883:30852/TCP   28d
```

Fonte: Produzida pelo autor

6.2.1 Adaptação do *NGINX* para Protocolo TCP

Uma das principais dificuldades enfrentadas durante o projeto foi a adaptação do *NGINX* para o protocolo TCP, uma vez que, tradicionalmente, o *NGINX* é utilizado para gerenciar tráfego HTTP. A quantidade limitada de documentação disponível para configurar o *NGINX* como *ingress* para outros protocolos de rede tornou esse processo ainda mais desafiador.

Diversos métodos foram testados para adaptar o *NGINX* ao protocolo TCP, sendo um dos principais a criação de um *ConfigMap* específico para a nova porta de comunicação TCP dentro do *namespace* do *NGINX* e a alteração dos argumentos do *deployment* do *ingress*. No entanto, essa abordagem não se mostrou eficaz.

A solução que obteve sucesso foi a criação de um recurso *TransportServer*, seguido de uma configuração global no *NGINX*, especificando o uso do protocolo TCP na porta 1883 — utilizada comumente por *brokers* MQTT. Para validar a configuração, a Figura 14 exibe a saída do comando *kubectl get services -n nginx-ingress*, onde é possível observar a porta 1883 listada como TCP na coluna *PORT(S)*.

6.3 Criação do Broker MQTT

O *broker* MQTT foi configurado como um serviço dentro do cluster. Para isso, um *service*, um *ConfigMap* e um *deployment* foram criados com esse fim. Dentro do kubernetes uma forma simples de validar a criação desses objetos foi rodar os seguintes comandos:

```
1 kubectl get services
2 kubectl get deployments
```

A Figura 15 expõe o resultado obtido após a execução dos comandos acima, indicando que o *Broker MQTT* estava presente no *cluster* e poderia ser acessado externamente.

Figura 15 – Criação do *Broker MQTT* no *cluster*

```
vitor@master-node:~$ kubectl get services
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP    46d
mosquitto    ClusterIP   10.96.13.167   <none>        1883/TCP   28d
vitor@master-node:~$ kubectl get deployments
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
mosquitto       1/1     1            1           28d
```

Fonte: Produzida pelo autor

Para validar o funcionamento do *broker* MQTT, foi criado um script em Python que envia dados aleatórios para o *broker*. O código utiliza a biblioteca `paho.mqtt.client`, conforme mostrado abaixo:

```python
import paho.mqtt.client as mqtt
import time
import random

client = mqtt.Client()
client.connect("192.168.15.140", 1883, 60)

try:
    while True:
        signal = random.randint(0, 100)
        client.publish("meu/topico", signal)
        print(f"Enviado: {signal}")
        time.sleep(1)
except KeyboardInterrupt:
    print("Interrompido pelo user")

client.disconnect()
```

Este script foi utilizado para testar a comunicação com o *broker* MQTT exposto pelo *NGINX Ingress*, validando o sucesso da configuração do cluster. A resposta vista da execução do código acima pode ser vista na Figura 16:

Figura 16 – Validação do *Broker MQTT* no *cluster*

Fonte: Produzida pelo autor

6.4 Desenvolvimento do Classificador de ECG

A aplicação utilizada no *cluster* foi uma rede neural para classificação de sinais ECG em cinco categorias: *Normal*, *Supraventricular*, *Ventricular*, *Fusion* e *Unknown*. Após o treinamento do modelo e a criação do código responsável por receber os sinais ECG e aplicar a rede neural sobre eles, o código foi transformado em uma imagem *Docker*.

Inicialmente, para validação do funcionamento, foi desenvolvido um código simples que enviava sinais ECG para um *broker MQTT* local, sem utilizar o *cluster Kubernetes*. A imagem *Docker* também foi executada localmente, fornecendo a classificação dos sinais em um tópico específico desse *broker*. Com a validação concluída, o próximo passo foi publicar a imagem no *DockerHub* para possibilitar sua utilização dentro do *cluster*.

6.5 Teste da Aplicação Fim a Fim

Para testar o funcionamento da aplicação fim a fim, foi desenvolvido um código que simulava o envio de sinais de ECG para o *broker* MQTT configurado no cluster Kubernetes. O cenário simula uma ambulância enviando dados de monitoramento de um paciente para um centro de diagnóstico remoto.

O código utilizado para esse teste foi implementado em Python, utilizando a biblioteca `paho.mqtt.client` para se comunicar com o *broker* MQTT e a biblioteca `neurokit2` para simular dados de ECG. Abaixo está o trecho de código utilizado para o envio dos dados:

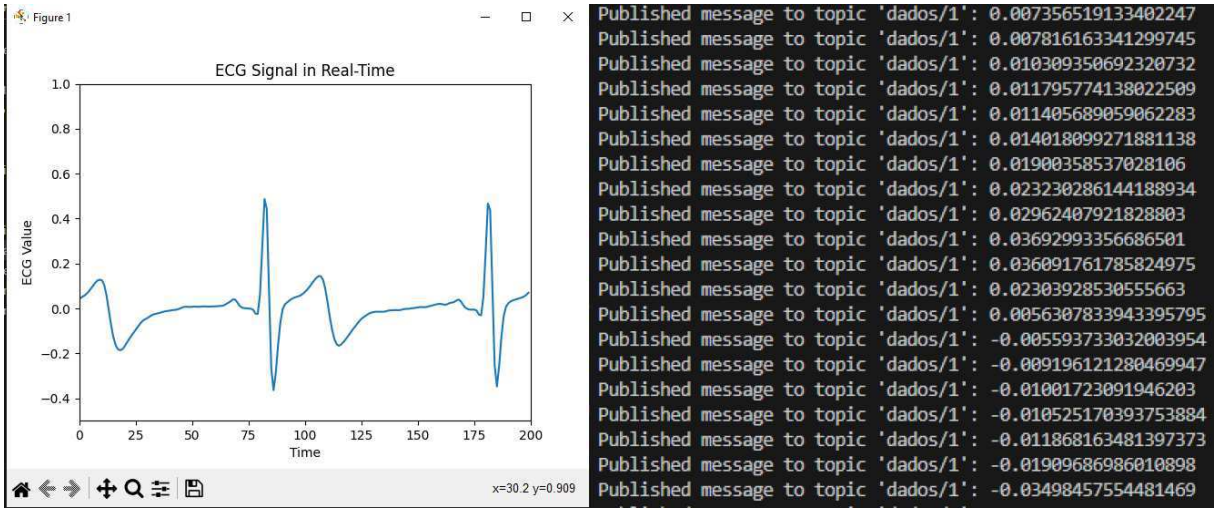
```
1 import paho.mqtt.client as mqtt
2 import time
3 import neurokit2 as nk
4 import matplotlib.pyplot as plt
5 from collections import deque
```

```
6
7  # Configurcoes para visualizacao ao vivo
8  plt.ion()
9  fig, ax = plt.subplots()
10 x_data = deque(maxlen=200)  # Para armazenar apenas os ultimos
       200 pontos
11 y_data = deque(maxlen=200)
12 line, = ax.plot(x_data, y_data)
13
14 # ECG ------------------------
15 data = nk.data("bio_eventrelated_100hz")
16 processed_data, info = nk.bio_process(ecg=data["ECG"],
17                                        rsp=data["RSP"], eda=data["
                                              EDA"], sampling_rate=100)
18
19 # MQTT ------------------------
20 def on_publish(client, userdata, mid):
21     return
22
23 def on_message(client, userdata, message):
24     return
25
26 client = mqtt.Client()
27 client.on_publish = on_publish
28 client.on_message = on_message
29 client.connect("192.168.15.140", 1883)
30 client.loop_start()
31
32 topic = "dados/1"
33 client.subscribe(topic)
34
35 cont = 0
36
37 # Configuracao do grafico
38 ax.set_xlim(0, 200)  # Ajustar o eixo X para mostrar os ultimos
       200 pontos
39 ax.set_ylim(-0.5, 1)  # Ajustar os limites do eixo Y para os
       valores do ECG
40 plt.xlabel('Time')
41 plt.ylabel('ECG Value')
42 plt.title('ECG Signal in Real-Time')
43
```

```
try:
    while True:
        # Enviar mensagem ao broker MQTT
        message = str(float(processed_data['ECG_Clean'][cont]))
        client.publish(topic, message)
        print(f"Published message to topic '{topic}': {message}")

        # Adicionar novos pontos ao grafico
        x_data.append(cont)
        y_data.append(float(processed_data['ECG_Clean'][cont]))
        line.set_xdata(range(len(x_data)))
        line.set_ydata(y_data)

        # Atualizar visualizacao do grafico
        ax.relim()  # Recalibrar limites dos dados
        ax.autoscale_view()  # Ajustar a visualizacao do grafico
        plt.draw()
        plt.pause(0.01)  # Pequena pausa para atualizacao do
           grafico

        # Aguardar proximo ciclo
        time.sleep(0.0167)  # Aproximadamente 60Hz
        cont += 1

        if cont >= 15000:
            cont = 0  # Reiniciar contagem ao alcancar 15000
               amostras
except KeyboardInterrupt:
    print("Exiting...")
    client.disconnect()
    client.loop_stop()
    plt.ioff()  # Desligar o modo interativo
    plt.show()
```

Neste cenário, os dados de ECG foram enviados continuamente para o tópico MQTT, com uma taxa de amostragem adequada para simular o envio em situações realistas. O processamento dos dados foi realizado pelo *broker* MQTT e distribuído aos consumidores conectados ao mesmo. A Figura 17 ilustra a saída da execução do código de envio de dados para o broker:

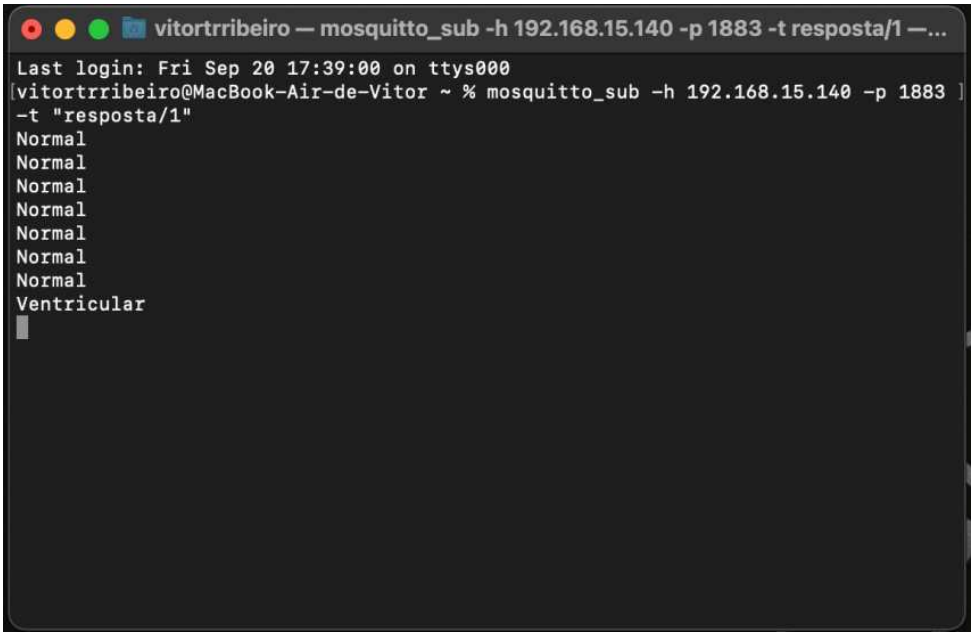
Figura 17 – Envio de Dados Para o *Broker MQTT* no *cluster*



Fonte: Produzida pelo autor

Para verificar se o fluxo de dados estava funcionando conforme o esperado, foi configurada uma segunda máquina conectada à mesma rede, a qual ficou responsável por se inscrever no tópico MQTT correto e ler os dados processados. Como esperado, os dados chegaram corretamente no tópico de destino, confirmando que a aplicação estava funcionando conforme o planejado, desde o envio dos sinais até a classificação realizada pelo modelo de rede neural. A Figura 18 mostra a recepção da classificação do sinal.

Figura 18 – Recepção da Classificação dos Dados



Fonte: Produzida pelo autor

6.6 Desafios e Soluções Implementadas

Durante o desenvolvimento e teste da solução, diversos desafios foram encontrados, principalmente relacionados ao ambiente virtualizado e à configuração do cluster Kubernetes. Um dos problemas mais críticos ocorreu quando a aplicação entrou em estado de *CrashLoopBackOff* após a tentativa de executar o modelo treinado no cluster. A causa foi identificada como a falta de suporte ao conjunto de instruções AVX (Advanced Vector Extensions) nas máquinas virtuais criadas no *VirtualBox*.

AVX é um conjunto de instruções desenvolvido pela Intel para acelerar operações matemáticas complexas e processamento vetorial, especialmente útil para tarefas intensivas em dados, como aprendizado de máquina e processamento de sinais. O TensorFlow, a biblioteca utilizada para a classificação dos sinais de ECG, aproveita o suporte ao AVX para otimizar o processamento de grandes volumes de dados e melhorar o desempenho.

Como o ambiente virtualizado não suportava AVX, foi necessário utilizar uma versão não oficial do TensorFlow, adaptada para ambientes sem esse conjunto de instruções. Após essa modificação, a aplicação foi reempacotada em um novo contêiner *Docker*, e o problema foi solucionado, permitindo a execução do modelo sem falhas.

Outro desafio foi adaptar o *NGINX Ingress Controller* para lidar com o protocolo TCP, necessário para a comunicação via MQTT. Inicialmente, o *NGINX Ingress* estava configurado para gerenciar apenas tráfego HTTP, o que gerou incompatibilidades com o *broker* MQTT. A solução encontrada foi configurar um *TransportServer*, recurso do *NGINX* para gerenciar protocolos não HTTP, o que resolveu a questão.

Um problema ocorreu durante a criação e validação do *broker MQTT*. Inicialmente, a aplicação fim a fim não funcionava corretamente: os dados eram enviados, mas a recepção da classificação não acontecia. Isso ocorreu porque o *deploy* do *broker* estava configurado com duas réplicas, o que fazia com que o usuário não lesse, necessariamente, a réplica correta. Para solucionar esse problema, foi necessário apenas ajustar a quantidade de réplicas para uma.

7 Conclusão

Neste trabalho, o objetivo de implementar e validar uma arquitetura distribuída para uma solução de telemedicina foi alcançado. Para isso, foi desenvolvida uma aplicação capaz de realizar o processamento de sinais biomédicos em tempo real em um ambiente distribuído, utilizando tecnologias de computação em borda e orquestração de contêineres. O sistema desenvolvido, baseado em um cluster *Kubernetes*, permitiu a classificação de sinais de ECG utilizando redes neurais convolucionais, demonstrando a viabilidade dessa abordagem no contexto de telemedicina.

A escolha do *Kubernetes* se mostrou acertada, proporcionando escalabilidade e flexibilidade na distribuição das cargas de trabalho entre os nós do cluster. Com o suporte do *MetalLB* para balanceamento de carga e do *NGINX Ingress Controller* para gerenciar o tráfego, foi possível criar uma infraestrutura capaz de atender às necessidades de comunicação entre os dispositivos e os serviços do cluster.

A utilização de um *broker* MQTT foi essencial para o fluxo contínuo de dados entre os dispositivos e a aplicação, especialmente em um cenário de monitoramento de saúde em tempo real. O sistema foi capaz de processar e classificar sinais de ECG à medida que eram recebidos, publicando os resultados em tópicos específicos, o que validou o funcionamento de ponta a ponta da solução.

Entretanto, foram identificadas algumas limitações durante o desenvolvimento. Uma delas foi a falta de suporte às instruções AVX nas máquinas virtuais usadas para testes, o que impactou o desempenho da aplicação que utilizava o *TensorFlow*. Para contornar essa limitação, foi necessária a adaptação do modelo para rodar em ambientes sem AVX, o que trouxe à tona a importância de selecionar adequadamente o hardware para produção.

De modo geral, a solução apresentou resultados promissores, com a arquitetura demonstrando capacidade de realizar diagnósticos em tempo real e processar dados de forma eficiente em um ambiente distribuído. O trabalho destacou também a importância de configurações flexíveis e otimizadas para garantir a interoperabilidade e o desempenho dos serviços.

7.1 Trabalhos Futuros

Com base nos resultados alcançados, diversas oportunidades para melhorar e expandir o sistema foram identificadas:

- Migrar para uma infraestrutura física com suporte completo a instruções AVX e outras otimizações de hardware, permitindo maior desempenho para modelos de aprendizado de máquina.
- Ampliar a aplicação para incluir o processamento de outros sinais biomédicos, como pressão arterial e saturação de oxigênio e até mesmo processamento de vídeo, aumentando o escopo da solução para abranger um sistema completo de monitoramento de saúde.
- Desenvolver uma interface gráfica para permitir que profissionais de saúde visualizem os dados em tempo real, tornando o sistema mais acessível e amigável para os usuários finais.
- Realizar testes em ambientes de produção com dispositivos médicos reais, a fim de validar o desempenho do sistema em condições de uso real e identificar possíveis áreas de otimização.

Essas ações futuras podem ampliar o impacto e a aplicabilidade da solução em cenários práticos de telemedicina, trazendo avanços tanto em termos de tecnologia quanto de usabilidade para profissionais de saúde e pacientes.

Referências

ABADI, M. et al. Tensorflow: A system for large-scale machine learning. *OSDI 16*, v. 16, n. 1, p. 265–283, 2016. Citado na página 44.

ACHARYA, U. R.; AL. et. Heart rate variability: a review. *Medical and Biological Engineering and Computing*, v. 44, n. 12, p. 1031–1051, 2007. Acesso em: 5 out. 2024. Citado na página 38.

ACHARYA, U. R.; AL. et. Automated diagnosis of coronary artery disease using different durations of ecg segments with cnn. *Knowledge-Based Systems*, v. 132, p. 62–71, 2017. Acesso em: 5 out. 2024. Citado na página 40.

ALICHERRY, M.; LAKSHMAN, T. Network aware resource allocation in distributed clouds. In: *Proceedings of the 28th IEEE Conference on Computer Communications*. [S.l.]: IEEE, 2009. p. 792–800. Acesso em: 5 out. 2024. Citado na página 26.

BARHAM, P.; AL. et. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, v. 37, n. 5, p. 164–177, 2003. Acesso em: 5 out. 2024. Citado na página 27.

BERNSTEIN, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, v. 1, n. 3, p. 81–84, 2014. Acesso em: 5 out. 2024. Citado na página 29.

BISHOP, C. M. *Pattern Recognition and Machine Learning*. 1. ed. [S.l.]: Springer, 2006. Acesso em: 5 out. 2024. Citado na página 36.

BOKOLO, A. J. Exploring the adoption of telemedicine and virtual software for care of outpatients during and after covid-19 pandemic. *Irish Journal of Medical Science*, v. 190, n. 1, p. 1–10, 2021. Citado na página 17.

BURNS, B.; BEDA, J.; HIGHTOWER, K. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. 1st. ed. [S.l.]: O'Reilly Media, 2016. Citado 3 vezes nas páginas 13, 31 e 43.

BURNS, B.; OPPENHEIMER, D.; BREWER, E. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*. 1st. ed. [S.l.]: O'Reilly Media, 2019. Citado na página 32.

CHADHA, R.; AL. et. Resource scaling in the cloud: Vms vs. containers. In: *Proceedings of the 2015 IEEE International Conference on Cloud Engineering*. [S.l.]: IEEE, 2015. p. 262–269. Acesso em: 5 out. 2024. Citado na página 27.

CONSORTIUM, E. A. Scaling ai at the edge: Opportunities and challenges. *Edge AI Journal*, v. 3, n. 2, p. 45–52, 2021. Citado na página 35.

DASTJERDI, A. V.; BUYYA, R. Fog computing: Helping the internet of things realize its potential. *Computer*, v. 49, n. 8, p. 112–116, 2016. Citado 2 vezes nas páginas 22 e 24.

DEAN, J.; GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, v. 51, n. 1, p. 107–113, 2008. Citado na página 25.

DINESEN, B. et al. Personalized telehealth in the future: A global research agenda. *Journal of Medical Internet Research*, v. 18, n. 3, p. e53, 2016. Citado na página 16.

DRAGONI, N. et al. Microservices: Migration of a mission critical system. *Software - Practice and Experience*, v. 49, n. 2, p. 255–280, 2017. Citado 2 vezes nas páginas 29 e 30.

EHEALTH, W. G. O. for. *Telemedicine: opportunities and developments in Member States: report on the second global survey on eHealth*. [S.l.]: World Health Organization, 2010. 93 p. p. (Global Observatory for eHealth Series, 2). Citado 2 vezes nas páginas 13 e 16.

ESTEVA, A.; AL. et. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, v. 542, p. 115–118, 2017. Acesso em: 5 out. 2024. Citado na página 37.

ESTEVA, A. et al. A guide to deep learning in healthcare. *Nature Medicine*, v. 25, p. 24–29, 2019. Citado na página 35.

FAWAZ, H. I.; AL. et. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, v. 33, p. 917–963, 2019. Acesso em: 5 out. 2024. Citado na página 36.

FOWLER, M.; LEWIS, J. *Microservices: A Definition of this New Architectural Term*. 2014. Acesso em: 28 ago. 2024. Citado na página 30.

FRAMPTON, R. *NGINX Cookbook: Over 70 recipes for real-world configuration, performance tuning, and log analysis with NGINX*. [S.l.]: Packt Publishing, 2018. Citado na página 44.

GAJARAWALA, S. N.; PELKOWSKI, J. N. Telehealth benefits and barriers. *The Journal for Nurse Practitioners*, v. 17, n. 2, p. 218–221, 2021. Citado na página 18.

GOLDBERGER, A. L.; AL. et. A unified approach to the prognosis of atrial fibrillation: machine learning and integrated physiology. *Journal of Clinical Investigation*, v. 118, p. 12–15, 2008. Acesso em: 5 out. 2024. Citado na página 39.

GOLDENBERG, I.; AL. et. Long qt syndrome. *Journal of the American College of Cardiology*, v. 51, n. 24, p. 2291–2300, 2008. Acesso em: 5 out. 2024. Citado na página 40.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. 1. ed. [S.l.]: MIT Press, 2016. Acesso em: 5 out. 2024. Citado na página 36.

GOVIL, K.; AL. et. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, v. 18, n. 3, p. 229–262, 1999. Acesso em: 5 out. 2024. Citado na página 27.

GREENHALGH, T. et al. Video consultations for covid-19. *British Medical Journal*, v. 368, p. m998, 2020. Citado 2 vezes nas páginas 17 e 18.

GREFF, K.; AL. et. Lstm: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, v. 28, n. 10, p. 2222–2232, 2017. Acesso em: 5 out. 2024. Citado na página 36.

GUTH, J. et al. Comparison of iot platform architectures: A field study based on a reference architecture. *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, p. 1–8, 2016. Citado na página 44.

HANNUN, A.; AL. et. Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network. *Nature Medicine*, v. 25, p. 65–69, 2019. Acesso em: 5 out. 2024. Citado 3 vezes nas páginas 37, 39 e 40.

HE, H.; WILSON, D. Analysis of bare metal vs. virtualized infrastructure for high-performance computing. *International Journal of High Performance Computing Applications*, SAGE, v. 29, p. 157–170, 2015. Acesso em: 5 out. 2024. Citado na página 25.

HIGHTOWER, K.; BURNS, B.; BEDA, J. Kubernetes: Orchestrating the future of computing. *IEEE Cloud Computing*, v. 5, n. 2, p. 22–29, 2018. Citado na página 31.

HIGHTOWER, K.; BURNS, B.; BEDA, J. Kubernetes patterns: Reusable elements for designing cloud-native applications. O'Reilly Media, 2020. Citado na página 43.

HOSSEINI, M.; AL. et. Optimized deep learning for eeg big data and seizure prediction bci via internet of things. *IEEE Transactions on Big Data*, v. 3, n. 2, p. 180–191, 2016. Acesso em: 5 out. 2024. Citado na página 37.

Intechopen. *An Overview of Wireless Mesh Networks*. 2024. Acesso em: 05 out. 2024. Citado na página 23.

KEESARA, S.; JONAS, A.; SCHULMAN, K. Covid-19 and health care's digital revolution. *New England Journal of Medicine*, v. 382, n. 23, p. e82, 2020. Citado 3 vezes nas páginas 13, 16 e 17.

KLIGFIELD, P.; AL. et. Recommendations for the standardization and interpretation of the electrocardiogram. *Circulation*, v. 115, p. 1306–1324, 2007. Acesso em: 5 out. 2024. Citado na página 38.

KRUSE, C. S. et al. Evaluating barriers to adopting telemedicine worldwide: A systematic review. *Journal of Telemedicine and Telecare*, v. 24, n. 1, p. 4–12, 2017. Citado na página 17.

LAU, D. H.; AL. et. Atrial fibrillation and stroke prevention: New concepts and controversies. *Heart, Lung and Circulation*, v. 22, p. 147–153, 2013. Acesso em: 5 out. 2024. Citado na página 40.

LEWIS, J.; FOWLER, M. *Microservices and the World of APIs*. 2015. Acesso em: 28 ago. 2024. Citado na página 31.

LIGHT, R. Mosquitto: server and client implementation of the mqtt protocol. *The Journal of Open Source Software*, v. 2, n. 13, p. 265, 2017. Citado na página 44.

Losant. *Hierarchical Edge Computing - A Practical Edge Architecture for IIoT*. 2024. Acesso em: 05 out. 2024. Citado na página 22.

MARTIS, R.; AL. et. Application of higher order cumulant features for cardiac health diagnosis using ecg signals. *International Journal of Neural Systems*, v. 23, n. 4, p. 1350014, 2013. Acesso em: 5 out. 2024. Citado na página 40.

MEHROTRA, A. et al. Rapidly converting to "virtual practices": Outpatient care in the era of covid-19. *NEJM Catalyst Innovations in Care Delivery*, v. 1, n. 2, p. 10.1056/CAT.20.0091, 2020. Citado na página 18.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, v. 2014, n. 239, 2014. Citado 3 vezes nas páginas 27, 28 e 44.

METALLB: A load-balancer implementation for Kubernetes. 2021. Citado na página 44.

Microsoft. *Mount Sinai Health System improves patient outcomes, engages providers, and scales for 3TB of annual data growth using AI on Azure*. 2023. Acesso em: 28 ago. 2024. Citado na página 25.

MIOTTO, R.; AL. et. Deep patient: An unsupervised representation to predict the future of patients from the electronic health records. *Scientific Reports*, v. 6, p. 26094, 2016. Acesso em: 5 out. 2024. Citado na página 37.

MOODY, G. B.; MARK, R. G. The impact of the mit-bih arrhythmia database. *IEEE Engineering in Medicine and Biology Magazine*, IEEE, v. 20, n. 3, p. 45–50, May-June 2001. Citado na página 51.

NEWMAN, S. *Building Microservices: Designing Fine-Grained Systems*. 1st. ed. [S.l.]: O'Reilly Media, 2015. Citado 2 vezes nas páginas 29 e 30.

NOTTINGHAM. Cardiology teaching package. *Nottingham*, 2024. Acesso em: 6 out. 2024. Citado na página 38.

PAHL, C. et al. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, v. 7, n. 3, p. 677–692, 2019. Citado na página 43.

PATEL, A. Kubernetes — architecture overview. *Medium*, 2021. Acesso em: 6 out. 2024. Citado 2 vezes nas páginas 33 e 34.

RAHMANI, A. M. et al. Fog computing in healthcare—a review and discussion. *IEEE Access*, v. 5, p. 9206–9222, 2018. Citado na página 19.

RAJPURKAR, P.; AL. et. Cardiologist-level arrhythmia detection with convolutional neural networks. *Nature Biomedical Engineering*, v. 2, p. 1–8, 2017. Acesso em: 5 out. 2024. Citado 3 vezes nas páginas 37, 39 e 40.

ROMAN, R.; LOPEZ, J.; MAMBO, M. Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges. *Future Generation Computer Systems*, v. 78, p. 680–698, 2018. Citado 3 vezes nas páginas 20, 21 e 35.

ROSADO, D. G.; AL. et. Virtualization in cloud computing: An overview. *Journal of Computing and Security*, Elsevier, v. 3, p. 1–16, 2014. Acesso em: 5 out. 2024. Citado na página 25.

ROSENBLUM, M.; GARFINKEL, T. Virtual machine monitors: Current technology and future trends. In: *Proceedings of the 1995 USENIX Annual Technical Conference*. [S.l.]: USENIX Association, 1995. p. 1–10. Acesso em: 5 out. 2024. Citado na página 26.

SANTOS, T. et al. Artificial neural networks for the classification of human heartbeat signals in public ecg databases: A systematic review. *International Journal of Medical Informatics*, v. 131, p. 103984, 2019. Citado na página 44.

SATYANARAYANAN, M. The emergence of edge computing. *Computer*, v. 50, n. 1, p. 30–39, 2017. Citado 4 vezes nas páginas 13, 18, 19 e 24.

SHAH, J.; AL. et. Security issues in bare metal and virtualized cloud computing platforms. *Journal of Cloud Computing*, Springer, v. 9, n. 1, p. 1–12, 2020. Acesso em: 5 out. 2024. Citado na página 26.

SHI, W. et al. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, v. 3, n. 5, p. 637–646, 2016. Citado 5 vezes nas páginas 13, 18, 19, 20 e 23.

SHICKEL, B.; AL. et. Deep learning for health informatics. *IEEE Journal of Biomedical and Health Informatics*, v. 21, n. 1, p. 4–21, 2017. Acesso em: 5 out. 2024. Citado 2 vezes nas páginas 36 e 37.

SINGH, R.; AL. et. Deployment strategies of cloud infrastructure: Bare metal, virtual machine, or containers? In: *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing*. [S.l.]: IEEE, 2019. p. 55–62. Acesso em: 5 out. 2024. Citado 3 vezes nas páginas 26, 27 e 28.

SMITH, J. E.; NAIR, R. *The Architecture of Virtual Machines*. [S.l.]: Morgan Kaufmann, 2005. Acesso em: 5 out. 2024. Citado na página 26.

Spiceworks. *What is Edge Computing?* 2024. Acesso em: 28 ago. 2024. Citado na página 21.

THöNES, J. Microservices. *IEEE Software*, v. 32, n. 1, p. 116–116, 2015. Citado na página 30.

TURNBULL, J. *The Docker Book: Containerization is the New Virtualization*. 1. ed. [S.l.]: Turnbull Press, 2014. Acesso em: 5 out. 2024. Citado 2 vezes nas páginas 28 e 29.

VARGHESE, B.; BUYYA, R. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*, v. 79, p. 849–861, 2016. Citado 2 vezes nas páginas 19 e 20.

VASSOLER, R. *Kubernetes for Developers: Using Docker and Helm*. 1st. ed. New York, NY: Apress, 2020. Citado na página 31.

WANG, S. et al. Edge computing in cyber-physical systems: An architecture and applications. *IEEE Transactions on Industrial Informatics*, v. 15, n. 4, p. 2252–2262, 2019. Citado na página 19.

WEATHERSPOON, H.; KUBIATOWICZ, J. Erasure coding vs. replication: A quantitative comparison. *International Workshop on Peer-to-Peer Systems (IPTPS)*, p. 328–338, 2002. Citado na página 24.

WHITELAW, S. et al. Applications of digital technology in covid-19 pandemic planning and response. *The Lancet Digital Health*, v. 2, n. 8, p. e435–e440, 2020. Citado na página 17.

WIIG, S. et al. Resilient and responsive healthcare services and systems: challenges and opportunities in a changing world. *BMC Health Services Research*, BioMed Central, v. 20, n. 1, p. 330, 2020. Citado na página 17.

XU, M. et al. Edge computing-driven connected health applications: Challenges and opportunities. *IEEE Network*, v. 34, n. 3, p. 104–110, 2020. Citado na página 35.

YI, S.; LI, C.; LI, Q. A survey of fog computing: Concepts, applications and issues. *Proceedings of the 2015 Workshop on Mobile Big Data*, p. 37–42, 2015. Citado 2 vezes nas páginas 21 e 24.

ZHANG, Z.; AL. et. Application of deep learning to electrocardiogram analysis for intelligent diagnosis of cardiovascular diseases. *Nature Communications*, v. 10, p. 1–9, 2019. Acesso em: 5 out. 2024. Citado na página 40.

ZHAO, B.; AL. et. Convolutional neural networks for time series classification. *Journal of Systems Engineering and Electronics*, v. 28, n. 1, p. 162–169, 2017. Acesso em: 5 out. 2024. Citado 2 vezes nas páginas 37 e 40.