UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA DE TELEINFORMÁTICA

CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

MARCEL ROCHA FONTELES VIEIRA

UMA ARQUITETURA SERVERLESS PARA APLICAÇÃO DE APRENDIZADO FEDERADO

FORTALEZA

2023

MARCEL ROCHA FONTELES VIEIRA

UMA ARQUITETURA SERVERLESS PARA APLICAÇÃO DE APRENDIZADO FEDERADO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Orientadora: Prof. Dra. Atslands Rego da Rocha

FORTALEZA

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

V716a Vieira, Marcel Rocha Fonteles.
Uma arquitetura serverless para aplicação de aprendizado federado / Marcel Rocha Fonteles Vieira. – 2023.
72 f. : il.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Tecnologia, Curso de Engenharia de Computação, Fortaleza, 2023.
Orientação: Prof. Dr. Atslands Rego da Rocha.

1. Aprendizado de máquina. 2. Aprendizado federado. 3. computação sem servidor. I. Título.

CDD 621.39

MARCEL ROCHA FONTELES VIEIRA

UMA ARQUITETURA SERVERLESS PARA APLICAÇÃO DE APRENDIZADO FEDERADO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Aprovada em:

BANCA EXAMINADORA

Prof. Dra. Atslands Rego da Rocha (Orientadora)
Universidade Federal do Ceará (UFC)

Prof. Dr. Flávio Rubens de Carvalho Sousa
Universidade Federal do Ceará (UFC)

Prof. Dra. Carina Teixeira de Oliveira
Instituto Federal de Educação, Ciência e Tecnologia do Ceará (IFCE)

À minha família pelo apoio incondicional ao longo da minha jornada. Em particular, quero expressar minha gratidão à minha mãe, Silvana. Agradeço do fundo do coração por estarem ao meu lado durante todos os desafios que enfrentei.

AGRADECIMENTOS

À minha mãe, Silvana, e ao meu padrasto, Ricardo, por seu apoio inabalável, incentivo e crença em minhas habilidades. Sua constante segurança e crença em meu potencial têm sido uma tremenda fonte de motivação em momentos desafiadores.

Além disso, gostaria de agradecer a toda a minha família por seu amor, apoio e compreensão. Sua crença em mim e seu incentivo durante todo o processo de pesquisa foram inestimáveis. Em especial, gostaria de agradecer à minha irmã, Larissa, e ao seu marido, Islan, pelo apoio inabalável.

À minha companheira Amanda, por todo amor e paciência, sempre esteve presente me incentivando e apoiando.

À Prof. Dra. Atslands Rocha por sua orientação e mentoria inestimáveis ao longo deste trabalho de pesquisa. Sua experiência e compromisso foram fundamentais para moldar minha compreensão do assunto.

À banca examinadora deste trabalho, pelo tempo dedicado a ler esta monografia.

RESUMO

Atualmente, as empresas salvam os dados de seus clientes para analisá-los posteriormente visando benefícios à empresa, seja melhorando o serviço prestado ou entendendo o comportamento do cliente para obter, por exemplo, um número maior de vendas. Essa análise de dados geralmente é realizada por meio de alguma técnica de aprendizado de máquina. Essas técnicas geralmente precisam de uma grande quantidade de dados para funcionar em um nível satisfatório e isso pode ser um empecilho para algumas empresas. Portanto, para aproveitar essas técnicas, é necessário considerar a colaboração com empresas concorrentes do mesmo setor e compartilhar dados entre elas. No entanto, essa abordagem pode não ser uma estratégia ideal, pois resultaria no compartilhamento das informações dos clientes com concorrentes diretos. Com isso, o aprendizado federado pode ser utilizado nesses casos para a construção de um modelo preditivo. O aprendizado federado permite o treinamento colaborativo realizado em dispositivos diversos sem compartilhar seus dados com um servidor central. Esta abordagem reduz a dependência dos recursos da nuvem, uma vez que transfere a responsabilidade para o usuário, e melhora o nível de privacidade dos dados em comparação com o método tradicional de aprendizado de máquina centralizado. Nesse caso, os modelos são treinados nos dispositivos dos usuários e, posteriormente, enviados para um servidor centralizado para realizar a agregação dos modelos por meio de um algoritmo específico. Dessa forma, não existe o envio dos dados dos usuários. Outro benefício do uso do aprendizado federado é que as empresas que têm uma grande quantidade de dados podem se unir para gerar um modelo ainda mais geral do comportamento de seus clientes. Nos últimos anos, notou-se o surgimento de uma nova arquitetura chamada de computação sem servidor. Nesse novo paradigma, os desenvolvedores podem focar mais na funcionalidade e menos na configuração do servidor e isso torna mais fácil e rápida a implementação ou remoção de funcionalidades. Além disso, problemas de escalabilidade são resolvidos pelo provedor de nuvem de forma automática e rápida. Este trabalho propõe uma arquitetura que utiliza computação sem servidor e técnicas de aprendizado federado para a construção de uma aplicação que seja capaz de treinar modelos e predizer novos valores com um alto grau de acerto. Foram realizados experimentos da arquitetura proposta neste trabalhos em ambiente de nuvem e de borda. Os resultados mostram que é possível utilizar computação sem servidor e aprendizado federado e ainda assim obter resultados de predição satisfatórios nos dois ambientes.

Palavras-chave: Aprendizado de máquina. Aprendizado federado. Computação sem servidor

ABSTRACT

Nowadays, companies save their customer data and analyze them a posterior to benefit the company, to improve the service provided or understand the customer's behavior to obtain, for example, a higher number of sales. This data analysis is usually performed using some machine learning technique. These techniques usually need a large amount of data to work satisfactorily, which can be a hindrance for some companies. Therefore, to take advantage of these techniques, it is necessary to consider collaborating with competing companies in the same industry and sharing data between them. However, there may be better strategies than this approach as it would result in sharing customer information with direct competitors. That said, federated learning can be used to build a predictive model in such cases. Federated learning enables collaborative training across devices without sharing customer data with a central server. This approach reduces the dependency on cloud resources since it transfers the responsibility to the user and improves data privacy compared to the traditional and centralized machine learning method. In this case, the models are trained on the users' devices and then sent to a centralized server to perform model aggregation using a specific algorithm. This way, there is no sending of data from the users. Another benefit of using federated learning is that companies with a large amount of data can join together to generate an even more general model of their customers' behavior. In recent years we have seen the emergence of a new architecture called serverless computing. In this new paradigm, developers can focus more on functionality and less on server configuration, making implementing or removing functionality easier and faster. Also, scalability problems are solved by the cloud provider automatically and quickly. This work proposes an architecture that uses serverless computing and federated learning techniques to build an application that is able to train models and predict new values with a high degree of accuracy. We performed experiments to evaluate our architecture in the cloud and edge computing environments. Our results show that it is possible to use serverless computing and federated learning and still obtain satisfactory prediction results in both environments.

Keywords: Machine learning. Federated Learning. Serverless computing

LISTA DE FIGURAS

Figura 1 – Modelo não linear de um neurônio 24
Figura 2 – Funções de ativação 25
Figura 3 – Diferentes arquiteturas de redes neurais 26
Figura 4 – CNN em processamento de imagens para reconhecimento de caracteres escritos à mão 28
Figura 5 – Algoritmo de Aprendizado Federado 30
Figura 6 – Categorias de aprendizado federado 32
Figura 7 – Ataque de envenenamento de modelos 33
Figura 8 – Arquitetura sem servidor 36
Figura 9 – Comparativo em *Virtual Machine* (VM) e Contêineres 39
Figura 10 – Computação na borda e na nuvem 40
Figura 11 – Módulos da Arquitetura de Aprendizado Federado com Funções como Serviço (AAFFS) 45
Figura 12 – Módulos orquestrador utilizando *Amazon Web Services* (AWS) 47
Figura 13 – Painel de controle do Grafana 50
Figura 14 – Módulos orquestrador utilizando *Kubernetes* e *Apache OpenWhisk* 50
Figura 15 – Módulos orquestrador utilizando *Raspberry Pi* e *Apache OpenWhisk* 51
Figura 16 – Módulos Cliente 52
Figura 17 – Exemplo de arquivo de configuração *Serverless* 53
Figura 18 – Módulo gerenciador utilizando AWS *Lambda* 54
Figura 19 – Módulo gerenciador utilizando *Kubernetes* e *Apache OpenWhisk* 54
Figura 20 – Módulo gerenciador utilizando *Raspberry Pi* e *Apache OpenWhisk* 55
Figura 21 – Arquitetura usando AWS 56
Figura 22 – Arquitetura usando *Kubernetes* e *Apache OpenWhisk* 56
Figura 23 – Arquitetura usando *Raspberry Pi* e *Apache OpenWhisk* 56
Figura 24 – Identificação de clientes 57
Figura 25 – Persistência dos modelos locais atualizados 58
Figura 26 – Perda por época por fração de clientes para o *MNIST* 61
Figura 27 – Acurácia por época por fração de clientes para o *MNIST* 61
Figura 28 – Acurácia por época por fração de clientes para o *MNIST* 62
Figura 29 – Acurácia do aprendizado federado e tradicional para o *MNIST* 62

Figura 30 – Perda por época por fração de clientes para o *CIFAR10* 63
Figura 31 – Acurácia por época por fração de clientes para o *CIFAR10* 63
Figura 32 – Acurácia por época por fração de clientes para o *CIFAR10* 64
Figura 33 – Latência para o *warm start* 66
Figura 34 – Latência para o *cold start* 66
Figura 35 – Registros do *OpenWhisk* em um cenário de *cold start* 67
Figura 36 – Duração da execução das funções de orquestração 68
Figura 37 – Execução concorrente para diferentes frações de clientes 68

LISTA DE TABELAS

Tabela 1 – Comparativo dos trabalhos relacionados 21
Tabela 2 – Linguagens suportadas e suas respectivas versões 36
Tabela 3 – Componentes do módulo orquestrador utilizado nos testes 55

LISTA DE ABREVIATURAS E SIGLAS

AAFFS	Arquitetura de Aprendizado Federado com Funções como Serviço
API	*Application Programming Interface*
AWS	*Amazon Web Services*
BaaS	*Backend as a Service*
CLI	*Command Line Interface*
CNN	*Convolutional Neural Network*
ECR	*Elastic Container Registry*
FaaS	*Function as a Service*
FedAvg	*Federated Averaging*
FL	*Federated Learning*
GCP	*Google Cloud Platform*
GDPR	*General Data Protection Regulation*
HTTP	*Hypertext Transfer Protocol*
IaaS	*Infrastructure as a Service*
IDE	*Integrated Development Environment*
IoT	*Internet of Things*
LGPD	Lei Geral de Proteção de Dados Pessoais
MPC	*Secure Multi-party computation*
NLLLoss	*Negative Log Likelihood Loss*
RAM	*Random Access Memory*
VM	*Virtual Machine*

SUMÁRIO

1 INTRODUÇÃO 14
1.1 Motivação 14
1.2 Objetivos 16
1.2.1 Objetivo geral 16
1.2.2 Objetivo específicos 17
1.3 Contribuições 17
1.4 Procedimentos metodológicos 17
1.5 Estrutura do documento 18
2 TRABALHOS RELACIONADOS 19
2.1 (JIANG *et al.*, 2021) Towards Demystifying Serverless Machine Learning Training 19
2.2 (GRAFBERGER *et al.*, 2021) Fedless: Secure and scalable federated learning using serverless computing 19
2.3 (WANG *et al.*, 2019) Distributed Machine Learning with a Serverless Architecture 20
3 FUNDAMENTAÇÃO TEÓRICA 22
3.1 Aprendizado de máquina 22
3.1.1 Redes neurais 23
3.1.2 Redes neurais convolucionais 27
3.2 Aprendizado federado 29
3.2.1 Categorias de aprendizado federado 31
3.2.2 Problemas do aprendizado federado 32
3.2.3 Técnicas de proteção aos ataques 34
3.3 Computação sem servidor 35
3.3.1 Contêineres 38
3.4 Computação na borda da rede 40
4 PROPOSTA 42
4.1 Funcionalidades 42
4.2 Arquitetura 43
5 MATERIAIS E MÉTODOS 46

5.1 **Módulo Orquestrador** 46
5.1.1 ***AWS Lambda*** 46
5.1.2 ***Kubernetes e Apache OpenWhisk*** 47
5.1.3 ***Raspberry Pi e Apache OpenWhisk*** 50
5.2 **Módulo Cliente** 51
5.3 **Módulo Gerenciador** 52
5.3.1 ***AWS Lambda*** 52
5.3.2 ***Kubernetes e Apache OpenWhisk*** 53
5.3.3 ***Raspberry Pi e Apache OpenWhisk*** 54
5.4 **Implementação completa** 55
5.4.1 ***Funcionamento geral*** 55
6 **AVALIAÇÃO** 60
6.1 **Métricas do aprendizado de máquina** 60
6.1.1 ***Conjunto de dados: MNIST*** 60
6.1.2 ***Conjunto de dados: CIFAR10*** 62
6.2 **Latência** 64
6.2.1 ***Warm start*** 65
6.2.2 ***Cold start*** 65
6.3 **Uso das funções como serviço** 67
7 **CONSIDERAÇÕES FINAIS** 69
7.1 **Limitações da pesquisa** 69
7.2 **Trabalhos futuros** 70
REFERÊNCIAS 71

1 INTRODUÇÃO

Atualmente, bilhões de pessoas acessam a internet diariamente e no ano de 2022 foi registrado mais de 7 bilhões de *smartphones* com acesso a internet no mundo (ERICSSON, 2023). Uma prática comum entre as empresas de tecnologia é o armazenamento de dados dos usuários e usá-los posteriormente para diversas finalidades, dentre elas campanhas publicitárias, melhoria do serviço prestado por essas empresas ou alguma outra funcionalidade que poderá aumentar o faturamento da empresa. A análise desses dados é realizada, normalmente, utilizando técnicas de aprendizado de máquina e o resultado é a criação de um modelo que irá predizer, sobre novos dados, algum comportamento de seus usuários.

Na forma tradicional de aprendizado de máquina, todos os dados são enviados para um servidor central, onde são armazenados, pré-processados e posteriormente utilizados para a construção de um modelo preditivo. Com isso, é necessário que esse servidor central tenha uma quantidade significativa de recursos de armazenamento e processamento para que seja capaz de realizar os algoritmos de aprendizado de máquina (RAMOS *et al.*, 2021). No entanto, a aplicação de técnicas de aprendizado colaborativo pode auxiliar na distribuição do processamento, permitindo o uso de máquinas com menos recursos.

Em muitos modelos de aprendizado de máquina, é necessária uma grande quantidade de dados para obter resultados satisfatórios (JAMES *et al.*, 2013), e algumas empresas não têm esse grande volume de informações. Portanto, para se beneficiar dessas técnicas, elas precisariam se unir a empresas concorrentes no mesmo setor e compartilhar os dados entre si, e isso pode não ser uma boa estratégia, pois as empresas estariam compartilhando as informações de seus clientes com seus concorrentes e, portanto, seria interessante utilizar uma estratégia de processamento colaborativo que ajude a preservar a privacidade dos dados, com um nível satisfatório de segurança das informações compartilhadas.

1.1 Motivação

Para superar esse desafio, os pesquisadores propuseram várias abordagens de aprendizado de máquina distribuído, incluindo o *Federated Learning* (FL) que oferece uma solução promissora para os desafios associados ao aprendizado de máquina tradicional, que é centralizado, permitindo o treinamento de modelos de aprendizado de máquina em fontes de dados distribuídas e com um nível satisfatório de privacidade dos dados (YANG *et al.*, 2019).

O FL é uma abordagem nova de aprendizagem de máquina que permite que várias entidades ou dispositivos treinem um modelo de forma colaborativa sem compartilhar seus dados com um servidor central (YANG *et al.*, 2019). Essa abordagem ganhou muita atenção nos últimos anos devido ao seu potencial para lidar com as preocupações de privacidade e segurança associadas à aprendizagem de máquina centralizada e houve um progresso significativo com muitas aplicações bem-sucedidas em vários domínios, incluindo saúde, finanças e cidades inteligentes (MOTHUKURI *et al.*, 2021).

Outro cenário em que a aprendizagem federada pode ocorrer é quando várias empresas, que possuem uma grande quantidade de dados e, portanto, não se enquadram no problema descrito acima, decidem se unir para criar um modelo preditivo mais geral do que seria possível se cada uma o criasse sozinha. Um exemplo disso é a união da *OpenAI*[1] e da *Microsoft*[2] no ano de 2023 para o desenvolvimento de, entre outros projetos, modelos de aprendizado de máquina melhores (MICROSOFT CORPORATE BLOGS, 2023). Essas empresas podem utilizar essa nova abordagem sem que haja a necessidade do compartilhamento das informações de seus usuários.

Na forma tradicional de aprendizado de máquina, existe um servidor central com uma quantidade massiva de dados dos usuários e tais servidores se tornam alvos de ataques *hackers* que tem por objetivo obter esses dados e vendê-los de forma clandestina. Em casos que duas ou mais empresas decidem entrar em colaboração para a construção de um modelo preditivo faz-se necessário o compartilhamento das informações de seus clientes e isso está cada vez mais complicado, pois novas legislações estão sendo criadas para evitar o compartilhamento e o uso indevido de dados pessoais dos usuários. A União Européia aprovou a *General Data Protection Regulation* (GDPR) (MULHOLLAND, 2018), um conjunto de leis que regulamenta como as empresas devem guardar os dados dos usuários e o seu descumprimento pode acarretar em multas milionárias. No Brasil, temos uma situação similar, em 2018 foi aprovada Lei Geral de Proteção de Dados Pessoais (LGPD), que foi influenciada pela GDPR, e passou a valer a partir do ano de 2021 (MULHOLLAND, 2018).

Para suprir esses e outros problemas surgiu a técnica chamada de FL. Nesse novo paradigma, não existe mais o envio dos dados do usuário para um servidor e com isso a empresa não precisa se preocupar com a segurança e o controle de acesso para esses dados. Tal técnica envolve cada usuário ser responsável por realizar o treinamento, usando os seus dados, de um

[1] https://openai.com/
[2] https://www.microsoft.com/pt-br

modelo local a partir de um modelo global. Após finalizado o treinamento, o modelo é enviado para os servidores da empresa e um algoritmo de agregação é executado, gerando um novo modelo global (YANG *et al.*, 2019).

Tal paradigma pode ser aplicado tanto em ambientes de nuvem, quanto na borda da rede. A computação em nuvem é um modelo centralizado de fornecimento de recursos e serviços de computação pela Internet. Ela permite que os usuários acessem e utilizem uma infraestrutura centralizada, eliminando a necessidade de hardware local e possibilitando recursos de computação dimensionáveis e flexíveis.

A computação de borda refere-se ao conceito de execução de tarefas computacionais na borda da rede ou próximo a ela. Em vez de transferir todos os dados para um servidor central ou nuvem para processamento, a computação de borda envolve a distribuição da computação para dispositivos de borda da rede ou nós intermediários. Ao aproximar a computação da borda da rede, a computação geralmente reduz a latência e aprimora os recursos de processamento em tempo real. Ela permite a análise de dados e a tomada de decisões mais rápidas, o que é particularmente valioso em aplicativos sensíveis ao tempo, como *Internet of Things* (IoT), sistemas autônomos e análise na borda.

Contudo, ainda é preciso implementar tais funcionalidades de maneira ágil, para isso pode ser utilizado o padrão de arquitetura computação sem servidor, dessa forma a empresa não precisa se preocupar com alguns problemas, dentre eles, configuração e escalabilidade do servidor, tudo isso é desempenhado pelo provedor desse serviço (HASSAN *et al.*, 2021).

1.2 Objetivos

Diante do contexto apresentado anteriormente sobre os desafios de desenvolver um modelo de aprendizado de máquina, utilizando dados de várias fontes, determina-se os seguintes objetivos deste trabalho.

1.2.1 Objetivo geral

Propor uma arquitetura de computação sem servidor para execução de aprendizado federado com dados sensíveis, com a intenção de viabilizar o desenvolvimento de um modelo com o uso de dados de diversas fontes sem a necessidade de compartilhá-los.

1.2.2 Objetivo específicos

1. Implementar funções como serviço para orquestração do FL;
2. Implementar FL na borda da rede para processamento colaborativo dos dados;
3. Realizar treinamento de aprendizado de máquinas em dispositivos com pouco poder computacional;
4. Desenvolver uma aplicação para validar a viabilidade da arquitetura proposta em ambientes de nuvem e borda.

1.3 Contribuições

Durante a pesquisa, foram analisadas diversas ferramentas para execução de funções sem servidor, sejam elas de código aberto ou proprietárias. Com isso, elaborou-se uma arquitetura para a realização de FL, utilizando uma combinação de dispositivos de borda e recursos de computação sem servidor baseados em nuvem. Nessa abordagem, é obtido um nível satisfatório de privacidade e segurança dos dados, além de aproveitar os benefícios da computação sem servidor, como escalabilidade e redução da sobrecarga operacional. Por fim, para determinadas cargas de trabalho, é possível obter uma economia nos custos de operação ao utilizar uma arquitetura desse tipo *serverless*.

Na implantação da arquitetura foi utilizada uma ferramenta proprietária da AWS chamada de *Lambda*[3] e uma ferramenta de código aberto, chamada de *Apache OpenWhisk*[4]. Além dessas duas ferramentas, foi utilizado um banco de dados *MongoDB* para persistência das informações. Essa pesquisa também que é possível executar funções sem servidor em ambientes de computação em borda, utilizando um *Raspberry Pi*[5] e o *Apache OpenWhisk*.

1.4 Procedimentos metodológicos

Na primeira etapa deste trabalho, foi realizado um levantamento das ferramentas disponíveis para a implantação de funções *serverless* e em quais delas eram compatíveis com a linguagem de programação *Python*. Com isso, foram selecionadas duas plataformas e definiu-se que a arquitetura proposta deve ser compatível com as plataformas selecionadas, além disso, foi definido o banco de dados para persistir as informações de treinamento.

[3] https://aws.amazon.com/pt/lambda/
[4] https://openwhisk.apache.org/
[5] https://www.raspberrypi.org/

Na segunda etapa, foi implementado o código das duas entidades principais existentes, o cliente e a função de orquestração. O código desenvolvido é compatível com as duas plataformas selecionadas.

Na terceira etapa, foi configurado o *Apache OpenWhisk*, ferramenta de código aberto de computação sem servidor em uma máquina virtual e em um *Raspberry Pi*.

Na quarta etapa, foram realizados os testes e computados os resultados. Nessa etapa, primeiro foram obtidos os dados utilizando o aprendizado de máquina tradicional e posteriormente os testes na arquitetura proposta de FL foram executados e registrados. Nessa arquitetura, os testes foram executados utilizando as duas plataformas escolhidas, *Apache OpenWhisk* e AWS *Lambda*. Nessa etapa de testes, foram utilizadas funções sem servidor em três ambientes distintos:

1. AWS *Lambda*.
2. *Apache OpenWhisk* em uma máquina virtual com processador Intel[6].
3. *Apache OpenWhisk* em um *Raspberry Pi*.

Na quinta etapa, foram comparados os resultados obtidos nos testes realizados. A comparação foi feita entre os resultados obtidos no aprendizado de máquina tradicional e os dados obtidos com o FL executado na arquitetura proposta. Por fim, avaliamos a viabilidade da arquitetura proposta.

1.5 Estrutura do documento

Este trabalho está organizado da seguinte forma: no Capítulo 2, são apresentados os trabalhos relacionados, mostrando seus objetivos e comparando com a proposta apresentada neste trabalho. No Capítulo 3 é exposto os fundamentos teóricos que embasaram essa pesquisa. No Capítulo 4, é exposta a metodologia utilizada para o desenvolvimento deste trabalho. No Capítulo 5, são apresentados os materiais e métodos da pesquisa e no Capítulo 6, são apresentados os resultados de experimentos e comparamos com a forma tradicional de se treinar um modelo preditivo. No Capítulo 7, concluí-se este trabalho apresentando as considerações finais e elencando ideais de trabalhos relacionados.

[6] https://www.intel.com.br/

2 TRABALHOS RELACIONADOS

Diante do contexto apresentado, é possível identificar alguns trabalhos relacionados atuais nas bases de pesquisas acadêmicas. Durante as buscas, foi realizado um recorte temporal no período entre 2018 e 2023, encontrando-se algumas pesquisas que propõem arquiterturas de aprendizado de máquina utilizando uma arquitetura de computação sem servidor. Nesta seção, são discutidas algumas pesquisas relevantes relacionadas ao trabalho em questão.

2.1 (JIANG *et al.*, 2021) Towards Demystifying Serverless Machine Learning Training

Em JIANG *et al*. (2021), os autores analisam as compensações entre custo e desempenho entre *Function as a Service* (FaaS) e *Infrastructure as a Service* (IaaS) para aprendizado de máquina distribuído. A pesquisa propôs o *LambdaML*, ferramenta que oferece suporte a diferentes variantes do aprendizado de máquina distribuído, como treinamento baseado em FaaS, treinamento híbrido, usando FaaS e IaaS, treinamento síncrono e assíncrono.

A pesquisa analisa os resultados da arquitetura proposta e realiza o comparativo do aprendizado de máquina distribuído entre três formas: treinamento utilizando apenas FaaS, treinamento utilizando apenas IaaS e treinamento utilizando FaaS e IaaS.

Diferentemente de JIANG *et al*. (2021), o nosso trabalho utiliza o FL, que embora tenha algumas semelhanças com o aprendizado de máquina distribuído, eles possuem diferenças importantes. Primeiro, os dados no FL são privados e não podem ser acessados pela entidade central, além disso no aprendizado de máquina distribuído, os nós trabalhadores dificilmente ficam ociosos, enquanto no FL, geralmente temos alguns clientes ociosos, pois não participarão da rodada de treinamento. Por fim, as ferramentas utilizadas na pesquisa suportam apenas uma plataforma e utilizam apenas máquinas com processadores na arquitetura *x86*, que são processadores utilizados em *desktops* tradicionais, enquanto a nossa pesquisa pode ser utilizada em duas plataformas distintas e é compatível com máquinas utilizando processadores *x86* ou *ARM*, normalmente presente em dispositivos embarcados.

2.2 (GRAFBERGER *et al*., 2021) Fedless: Secure and scalable federated learning using serverless computing

Em GRAFBERGER *et al*. (2021), os autores realizaram um estudo usando uma arquitetura sem servidor para a execução de um FL. O estudo propõe uma arquitetura composta

de duas entidades principais. A primeira tem o papel de orquestrar o treinamento e a segunda é a detentora dos dados, que pode ser chamada de cliente, e realizará o treinamento local quando solicitado. A entidade que tem o papel de orquestrar o treinamento foi implementada no *Apache OpenWhisk* utilizando máquinas com processadores da família *x86* e os clientes podem ser implantados nos seguintes provedores: AWS *Lambda*, *Apache OpenWhisk*, *Google Cloud Functions* e *Azure Function*. Neste trabalho a arquitetura foi testada utilizando três conjuntos de dados distintos.

Diferentemente de GRAFBERGER *et al.* (2021), esse trabalho realiza a implementação da entidade de orquestração em duas plataformas, na AWS *Lambda* e no *Apache OpenWhisk*, dessa forma a pesquisa demonstra que é possível executar a arquitetura em ambientes distintos. Nessa última plataforma, nosso trabalho realizou os testes em máquinas com processadores da família *x86* e *ARM*. Outra diferença é que nossos clientes podem ser implementados em funções sem servidor ou em qualquer dispositivo compatível com distribuição Linux.

2.3 (WANG *et al.*, 2019) Distributed Machine Learning with a Serverless Architecture

Em WANG *et al.* (2019), os autores propuseram uma arquitetura onde os usuários podem aproveitar funções refinadas para treinar modelos de ML na nuvem, usando arquiteturas sem servidor e utilizando apenas a AWS *Lambda*, o que elimina a necessidade de provisionamento e gerenciamento complexos de clusters. Essa abordagem também permite o treinamento distribuído assíncrono e incorpora um algoritmo baseado em *Deep Reinforcement Learning* que ajusta continuamente o número de funções e sua memória para otimizar o desempenho e o custo.

Diferentemente de WANG *et al.* (2019), o nosso trabalho utiliza ferramentas que são compatíveis com a AWS *Lambda* e o *Apache OpenWhisk*, sejam eles utilizando um processador *x86* ou *ARM*. Além disso, nosso trabalha utiliza FL, onde os dados são privados e podem ser acessados apenas pelas entidades que os possuem, enquanto em (WANG *et al.*, 2019) utiliza o aprendizado de máquina distribuído que necessita do compartilhamento das informações para funcionar corretamente.

A Tabela 1 resume as lacunas que o presente trabalho pretendeu preencher comparados com os trabalhos relacionados mencionados. Nossa intenção, com essa tabela, é evidenciar categorias que observamos na construção da nossa pesquisa em comparação com os outros trabalhos.

Tabela 1 – Comparativo dos trabalhos relacionados

Autores	Aprendizado	FaaS	Processamento na borda
Jiang *et al* (2021)	Distribuído	AWS *Lambda*	Não
GRAFBERGER *et al.* (2021)	Federado	*Apache OpenWhisk*	Não
WANG *et al.* (2019)	Distribuído	AWS *Lambda*	Não
Arquitetura proposta	Federado	AWS *Lambda* e *Apache OpenWhisk*	Sim

Fonte: Elaborado pelo autor (2023)

3 FUNDAMENTAÇÃO TEÓRICA

3.1 Aprendizado de máquina

Aprendizado de máquina é uma das últimas tentativas numa longa fila de tentativa para diluir o conhecimento humano e sua racionalidade numa forma que seja possível construir algoritmos automatizados (DEISENROTH *et al.*, 2020). Assim, podemos dizer que o aprendizado de máquina objetiva o desenvolvimento de algoritmos que conseguem extrair, de forma automática, informações importantes de um conjunto de dados. Isso indica que essa área de estudo busca metodologias de propósito geral e que possam ser utilizadas em uma vasta quantidade de conjuntos de dados.

O aprendizado de máquina pode ser dividido em três conceitos centrais: dados, modelos e aprendizado (DEISENROTH *et al.*, 2020).

Dados são as entradas no qual o algoritmo de aprendizagem da máquina é treinado. É constituído por várias amostras, que incluem tanto as variáveis de entrada, como a variável alvo. A qualidade e quantidade das amostras afeta diretamente o desempenho do modelo de aprendizagem da máquina (DEISENROTH *et al.*, 2020). Já o modelo é uma representação matemática da relação entre as características e a variável alvo. Esta pode ser uma simples equação linear ou uma complexa rede neural e sua escolha é feita com base no tipo de problema e na natureza dos dados. Por fim, o aprendizado é o processo de encontrar os melhores parâmetros de modelo que minimizem o erro de previsão nos dados de formação. Isto é conseguido através de um processo de ajuste iterativo dos parâmetros do modelo com base no erro de previsão até ser atingido um nível satisfatório de acerto.

O processo de aprendizado pode ser supervisionado ou não supervisionado (JAMES *et al.*, 2013). No aprendizado supervisionado os dados de treino consistem nas variáveis de entrada e nas variáveis alvo. O objetivo é treinar um modelo que mapeia as entradas para as saídas e faz previsões precisas de dados novos e não vistos. Já no aprendizado não supervisionado, os dados de treino consistem em apenas nas variáveis de entrada. Nessa estratégia, o objetivo é inferir a estrutura dos dados com base nos padrões dos dados de entrada.

Sobre o conjunto de variáveis, elas podem ser qualitativas ou quantitativas. As variáveis qualitativas qualifica a amostra em uma determinada categoria, por exemplo o sexo de uma pessoa. Já as variáveis quantitativas têm valores numéricos e quantificam uma determinada característica, por exemplo a idade ou o salário de uma pessoa (JAMES *et al.*, 2013). Com base

na variável alvo, podemos dividir os problemas em duas categorias. Problemas de regressão normalmente tem como alvo uma variável quantitativa, já problemas de classificação objetiva prever o valor de uma variável qualitativa. Neste trabalho iremos desenvolver uma arquitetura que poderá ser aplicada para os dois tipos de problema.

3.1.1 Redes neurais

Atualmente temos uma grande variedade de modelos disponíveis para uso e cada um tem pontos positivos e pontos negativos. Essa grande variedade de modelos decorre do fato que cada modelo resolve um determinado problema ou é bom para conjuntos de dados com certas características. Apesar de poder ser utilizado qualquer tipo de modelo na arquitetura proposta, focamos os testes em modelos usando redes neurais, já que são os mais utilizados atualmente.

Uma rede neural é uma técnica de processamento paralelo massivo feito de pequenas unidades simples de processamento que é naturalmente propensa a salvar o conhecimento, tornando-o disponível para uso (HAYKIN, 2009). Tal conhecimento é obtido pela rede através do processo de treinamento e essa estratégia é inspirada na estrutura e função do cérebro humano. Consiste em múltiplos nós de processamento interligados, chamados neurônios artificiais, que estão organizados em camadas. Cada neurônio recebe entrada de múltiplos outros neurônios, processa a entrada usando uma função de ativação e emite um sinal para outros neurônios na camada seguinte. As interligações entre os neurônios são representadas por pesos, que são parâmetros que controlam a força das ligações.

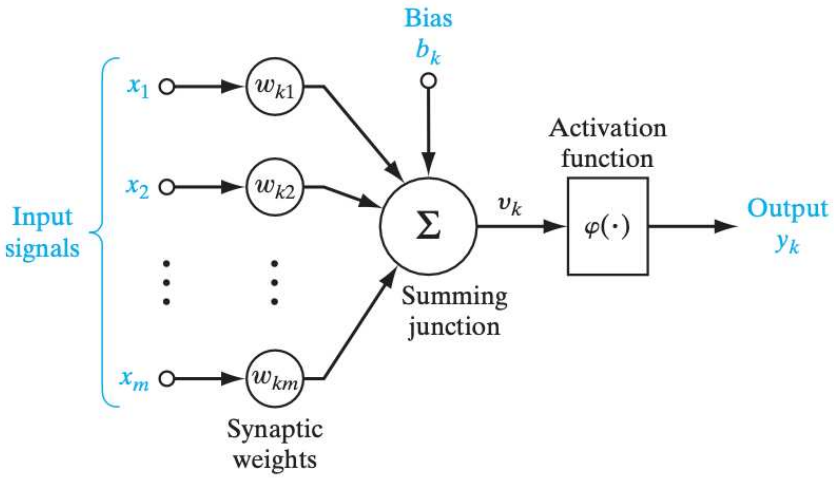
As redes neurais podem resolver problemas não lineares e bastante complexos (HAYKIN, 2009) que, aos olhos de um ser humano, aquele conjunto de dados não possui um padrão de comportamento. Além disso, tal estratégia é adaptativa, ou seja, tem a capacidade de se adaptar caso haja alguma mudança no comportamento geral dos dados.

Um neurônio é a unidade fundamental de processamento de informação de uma rede neural (HAYKIN, 2009). A Figura 1 apresenta um modelo de neurônio e que forma a base de complexos modelos de redes neurais. Analisando a figura, podemos perceber que essa unidade fundamental é composta por 3 estruturas básicas. A primeira é um conjunto de sinapses, cada um é relacionado a um peso específico, tal peso representa a força com o qual o sinal chegará na próxima parte do neurônio. O peso pode ser entendido como um multiplicador do sinal de entrada, onde altos valores significam uma forte influência na saída. A segunda estrutura é uma função de soma que agrupa todos os sinais de entrada daquela neurônio. Por fim, a função de

ativação é responsável por limitar a amplitude do sinal de saída. Normalmente, o sinal de saída é um intervalo fechado [0,1] ou [-1,1].

O modelo de neurônio inclui um viés, que nunca é nulo. É possível reduzir o viés de um modelo e isso é feito através do treinamento, mas não é possível eliminá-lo por completo.

Figura 1 – Modelo não linear de um neurônio



Fonte: Haykin (2009)

Em termos matemáticos, podemos descrever o neurônio apresentado na figura 1 com o par de equações (HAYKIN, 2009):

$$u_k = \Sigma_{j=1}^{m}(w_{kj}x_j) \tag{3.1}$$

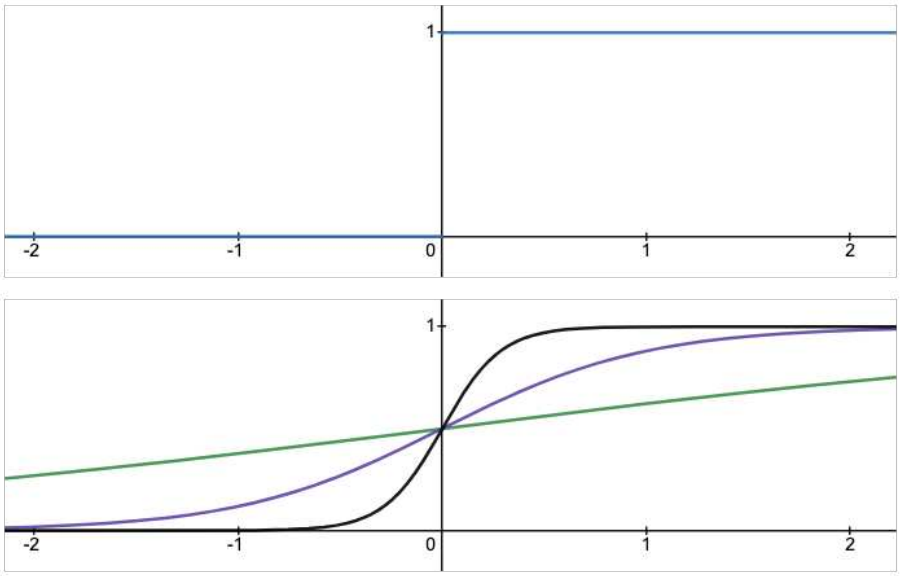
$$u_k = \varphi(y_k b_k). \tag{3.2}$$

As funções de ativação podem ser divididas em dois tipos básicos. O primeiro é chamado de função de limiar em que para valores até um limite essa função tem valor 0 e para valores maiores que o limite, tem valor 1. O segundo tipo é uma função sigmóide que tem um formato de "S"e é a função de ativação mais comumente utilizada nas redes neurais. Um exemplo desse tipo de função pode ser definida pela função logística (3.3), onde "a"é o parâmetro de inclinação da função.

$$\varphi(v) = \frac{1}{exp(-av)} \tag{3.3}$$

Na parte superior da Figura 2 vemos uma função de ativação do tipo limiar e na parte inferior vemos funções sigmoidais com diferentes inclinações. Conforme o valor do parâmetro de inclinação aumenta, mais a função sigmóide se aproxima da função de limiar.

Figura 2 – Funções de ativação



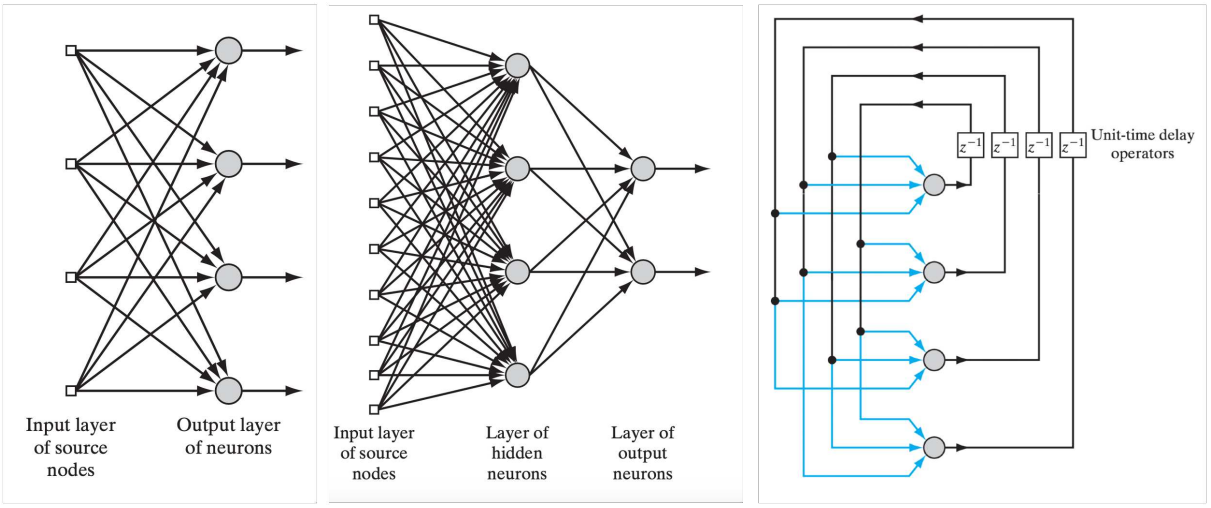
Fonte: Elaborado pelo autor (2023)

A partir de um neurônio podemos dividir a arquitetura das redes neurais em 3 classes principais(HAYKIN, 2009).

1. Redes neurais com apenas uma camada: nessa forma mais simples, temos a camada de entrada diretamente ligada aos neurônios da camada de saída. A camada de entrada não é contada pois nenhuma computação é executada nela, com isso temos apenas a camada de saída com algum tipo de processamento.
2. Redes neurais com múltiplas camadas: nessa arquitetura temos mais de uma camada com algum tipo de processamento e ela se diferencia da anterior pela presença de uma ou mais camadas ocultas entrada as camadas de entrada e saída, com isso, o sinal de entrada de uma camada é igual ao sinal de saída da camada anterior. Ao adicionar uma ou mais camadas ocultas em uma rede neural podemos obter melhores resultados a partir da mesma entrada.
3. Redes neurais recorrentes: nessa arquitetura é necessário ter pelo menos um ciclo de *feedback*, ou seja, podemos ter apenas uma camada de neurônios, onde em cada neurônio o sinal de entrada é o sinal de saída dessa mesma camada. Esse *feedback* pode voltar para o mesmo neurônio ou para neurônios distintos, quando o primeiro caso ocorre, temos o chamado *self-feedback*.

Na Figura 3 vemos os três tipos de arquiteturas, no lado esquerdo da figura temos representado as redes neurais com apenas uma camada. Já na parte central, temos múltiplas camadas e por fim, no lado direito, temos uma rede neural recorrente.

Figura 3 – Diferentes arquiteturas de redes neurais



Fonte: Haykin (2009)

As redes neurais podem ser utilizadas para solucionar problemas tanto de regressão quanto de classificação. Para problemas de regressão temos na camada de saída apenas um neurônio, já em problemas de classificação, a rede neural terá na camada de saída uma quatidade de neurnônios igual a quantidade de classes daquele problema (HASTIE *et al.*, 2001), ou seja, em um problema que desejamos classificar o sexo de um animal a camada de saída terá um total de dois neurônios.

Em redes neurais com apenas uma camada, o processo de treinamento é relativamente simples porque o erro pode ser computado como uma função direta dos pesos, o que permite um fácil cálculo de gradiente. No caso de redes com mais de uma camada, a perda é uma função de composição complicada dos pesos em camadas anteriores. O gradiente de uma função de composição é computado usando o algoritmo de retropropagação (AGGARWAL, 2018). O algoritmo de retropropagação utiliza a regra da cadeia do cálculo diferencial e é formado por duas fases *forward phase* e *backward phase* (AGGARWAL, 2018). Na primeira fase os sinais de entradas são passados para a rede neural que resulta numa cascata de cálculos em todas as camadas utilizando o conjunto atual de pesos. A saída prevista é comparada com o valor obtido na saída da rede neural. Na segunda fase, o objetivo é aprender o gradiente da função de perda com respeito aos diferentes pesos. Esses gradientes são usados para atualizar os pesos. Em

resumo, esse algoritmo funciona da seguinte forma:

1. Todos os pesos são inicializados com valores aleatórios.
2. São fornecidos dados de entrada à a rede neural e então é calculado o valor da função de erro. Esse cálculo é feito utilizando o valor esperado da saída. Isso só é possível porque o algoritmo de retropropagação é utilizado em aprendizado supervisionado.
3. É calculado o valor dos gradientes para cada peso da rede neural, isso é feito para tentar minimizar o valor da função de erro, já que sabemos que o vetor gradiente indica a direção de maior crescimento de uma função. Como queremos a diminuição da função de erro, basta seguir o sentido contrário do gradiente.
4. Após o cálculo do gradiente, os pesos são atualizados de modo iterativo. A cada iteração os gradientes são calculados novamente e esse processo é repetido até obtermos um erro menor do que um valor definido ou até atingir o número máximo de iterações. Após atingido essa condição de parada, podemos dizer que a rede neural está treinada.

3.1.2 Redes neurais convolucionais

Os neurônios de uma rede neural podem ser organizados de diversas formas e conforme essa organização é feita podemos definir redes neurais com propósitos diferentes. A *Convolutional Neural Network* (CNN) é um exemplo disso, ela é um caso especial de redes neurais com múltiplas camadas e são utilizadas para problemas que precisamos reconhecer padrões (HAYKIN, 2009).

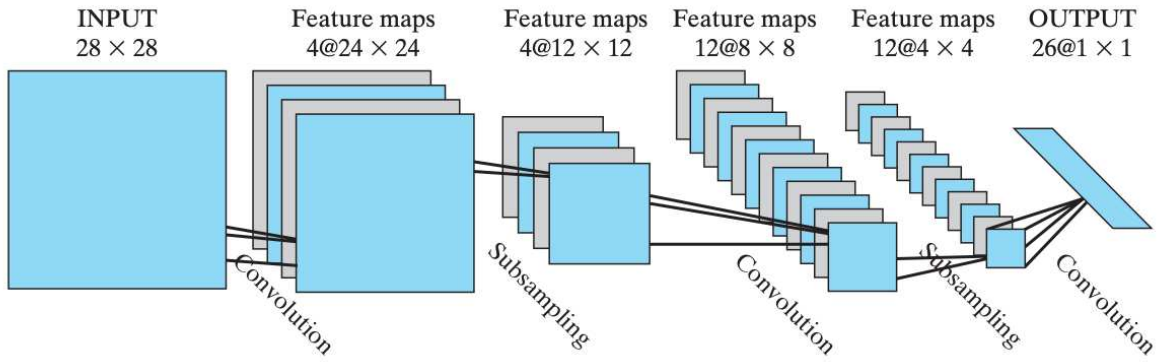
Uma rede convolucional é uma rede neural com múltiplas camadas projetada especificamente para identificar formas bidimensionais com forte dependência espacial em regiões locais, como por exemplo, imagens e vídeos (HAYKIN, 2009). Eles são inspirados pela estrutura e função do córtex visual, que é a parte do cérebro responsável pelo processamento da informação visual e historicamente é o tipo de rede neural mais bem sucessdida, ela pode ser usada até mesmo para processamento de texto. (AGGARWAL, 2018).

Tais redes realizam operações de convolução que são operações onde um filtro é usado para mapear as ativações de uma camada para outra. Tal operação utiliza um filtro tridimensional de pesos com a mesma profundidade da camada pela a qual ele está passando, mas com um tamanho espacial menor(AGGARWAL, 2018).

Uma CNN consiste em várias camadas, incluindo camadas convolutivas, camadas de pooling e camadas totalmente conectadas. Cada uma dessas camadas tem 3 dimensões que representa as dimensões espaciais e as caractéristicas(AGGARWAL, 2018), por exemplo, nos conjuntos de dados com imagens em preto e branco, a terceira camada tem valor 1. Na camada convolucional, a rede aplica um conjunto de filtros à imagem de entrada, que são usados para detectar diferentes características na imagem, tais como bordas, cantos e texturas. A camada de agrupamento é então utilizada para reduzir as dimensões espaciais dos mapas de características, mantendo ao mesmo tempo as informações importantes. Finalmente, a camada totalmente conectada é usada para fazer a previsão final com base nas características extraídas.

A principal vantagem de usar CNN é que elas podem aprender automaticamente a detectar as características mais relevantes nos dados de entrada e isto as torna bem adequados para tarefas complexas de reconhecimento de imagem, onde os métodos tradicionais de processamento de imagem podem falhar. O treinamento desse tipo de redes neurais é feito de forma supervisionada.

Figura 4 – CNN em processamento de imagens para reconhecimento de caracteres escritos à mão



Fonte: Haykin (2009)

A Figura 4 apresenta a arquitetura de uma CNN que propõe identificar caracteres escritos a mão e é composta por uma camada de entrada, quatro camadas ocultas e uma camada de saída. A camada de entrada é composta por 28x28 nós sensoriais que recebem imagens de diferentes caracteres que foram centralizados e são aproximadamente do mesmo tamanho. Em seguida, a primeira camada oculta executa uma operação de convolução, que consiste em 4 mapas de características cada um contendo 24x24 neurônios. Na segunda camada oculta é executada uma subamostragem e nessa camada temos os coeficientes e os viés para serem treinados e uma

função de ativação. Na terceira e quarta camada ocultas é realizada outra operação de convolução e subamostragem, respectivamente. Por fim, na camada de saída é realizada a última operação de convolução e temos 26 neurônios, cada um representando um caractere do alfabeto.

A arquitetura de CNN mostrado na Figura 4 é apenas umas das possibilidades para esse tipo de redes neurais. Diversas combinações das operações de convolução e subamostragem podem ser utilizadas e cada uma vai resolver problemas específicos. Com isso, esse tipo de rede neural é um exemplo que a escolha da arquitetura utilizada deve ser realizado levando em consideração o conjunto de dados que será utilizado (AGGARWAL, 2018).

3.2 Aprendizado federado

O aprendizado federado é um paradigma que faz com os donos dos dados treinem de forma colaborativa um modelo de aprendizado de máquina enquanto mantém seus dados privados (GRAFBERGER *et al.*, 2021). Na forma tradicional, todos os dados dos usuários são enviados para um servidor centralizado, onde são realizadas as operações de pré-processamento e treinamento dos modelos de predição. Após o treinamento, esse modelo é utilizado por todos os usuários. Esse uso pode ser feito de duas formas, na primeira, o modelo é enviado para o usuário e a predição é feita de forma local e utilizando os recursos computacionais do usuário. Uma segunda forma, é o dispositivo do usuário enviar uma requisição para o servidor central com as informações de entrada e receber como resposta a previsão.

Como na forma tradicional é necessário o envio de todos os dados do usuário para um servidor central, não há formas de garantir a privacidade do usuário e a empresa que guarda esses dados é responsável por eles, ou seja, qualquer tipo de vazamento ela será responsabilizada. Com isso um novo paradigma, chamado de aprendizado federado, foi proposto, tal paradigma consegue obter resultados similares de precisão com o do modelo centralizado, enquanto garante a privacidade do usuário (MCMAHAN *et al.*, 2016).

No aprendizado federado, os modelos descritos na seção anterior continuam funcionando da mesma forma, por exemplo, as CNN continuam realizando operações de convolução e de subamostragem, porém esse procedimento é realizado em vários dispositivos e os modelos gerados são enviados para um servidor central.

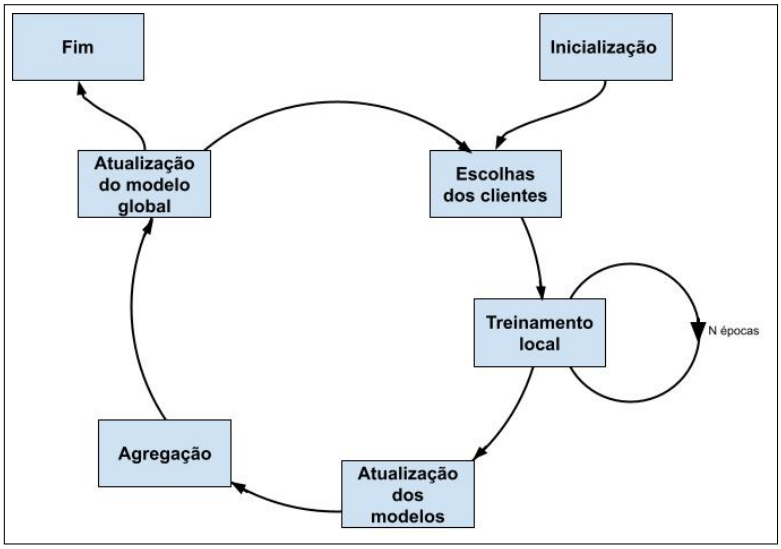
Em geral, um sistema de aprendizado federado possui duas entidades principais: os clientes e o servidor central (GRAFBERGER *et al.*, 2021). Não há restrição para o tipo de dispositivo dos clientes, dentro de um mesmo sistema podem existir smartphones, tablets,

computadores e outros dispositivos.

O algoritmo de aprendizado federado é iterativo, ou seja, é realizado por etapas que se repetem até se atingir o resultado desejado. A Figura 5 ilustra todas as etapas do algoritmo de FL e seu funcionamento completo ocorre da seguinte forma:

1. Inicialização: o servidor central inicia os pesos do modelo.
2. Escolha dos clientes: em cada iteração o servidor central escolhe uma quantidade definida de clientes que deverão realizar o treinamento e então envia os pesos do modelo para cada cliente. Essa escolha é feita de forma aleatória.
3. Treinamento local: Cada cliente treina o modelo com seus dados locais em uma quantidade fixa de épocas.
4. Atualização dos modelos: após o treinamento local, cada cliente envia seu modelo atualizado para o servidor central.
5. Agregação: o servidor central agrega todos os pesos de todos os modelos atualizados dos clientes participantes e calcula sua média.
6. Atualização do modelo global: o servidor central atualiza o modelo global usando a média dos pesos calculados na etapa anterior.
7. Por fim, são repetidas as etapas 2 a 6 até que se obtenha um nível de predição satisfatório.

Figura 5 – Algoritmo de Aprendizado Federado



Fonte: YANG et al. (2019)

Com isso, percebemos que normalmente o servidor não armazena nenhum dado e é responsável por organizar o treinamento, decidir quais clientes irão realizar o treinamento em

cada iteração e executar o algoritmo de agregação de modelos (GRAFBERGER *et al.*, 2021). Normalmente é implementada o *Federated Averaging* (FedAvg) (MCMAHAN *et al.*, 2016) como o algoritmo de agregação.

3.2.1 Categorias de aprendizado federado

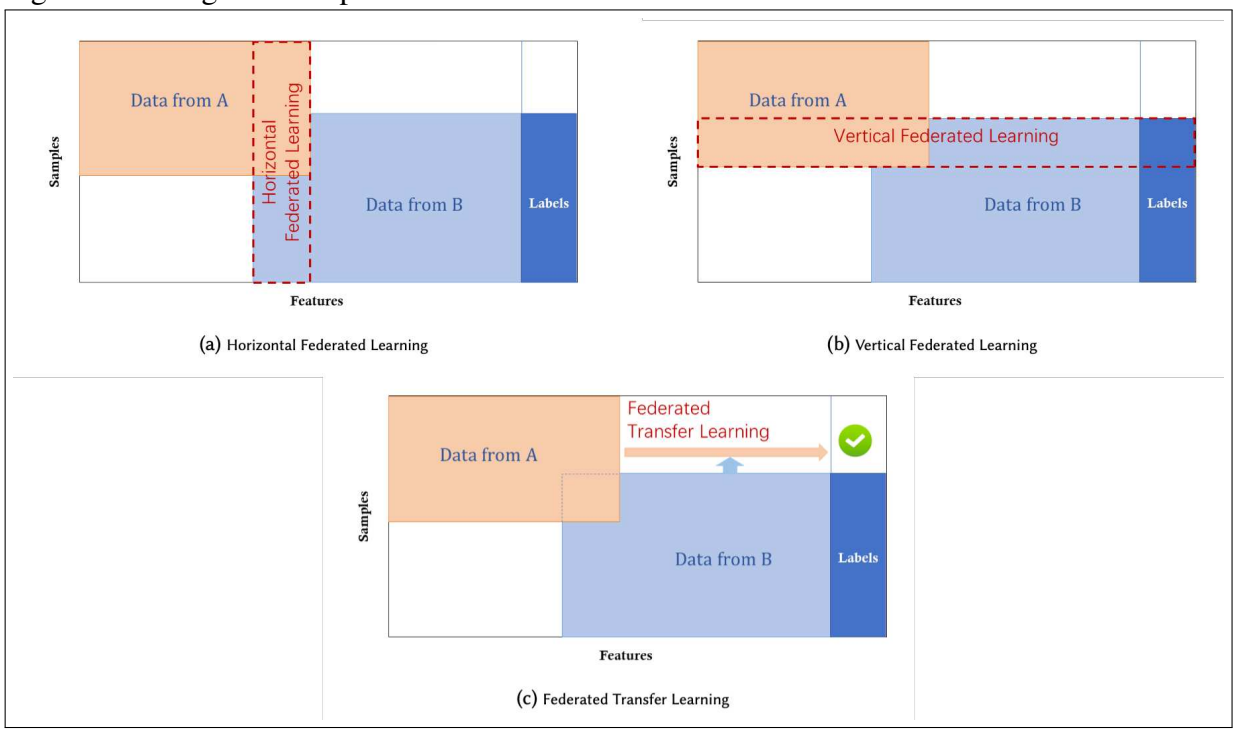
De acordo com a disposição do conjunto de dados, podemos classificar o aprendizado federado em três categorias. Quando temos cenários nos quais conjuntos de dados em locais diferentes compartilham características sobrepostas, mas diferente no espaço de amostra, chamamos de aprendizado federado horizontal ou aprendizado federado particionado por amostra (YANG *et al.*, 2019). Por exemplo, dois hospitais em regiões distintas tem um conjunto diferente de pacientes e com pouca ou nenhuma interseção, porém o negócio das duas empresas é bem similar e eles tendem a armazenar as mesmas informações dos pacientes.

Quando temos para uma mesma amostra de dados informações diferentes em locais diferentes, ou seja, dois conjuntos de dados que compartilham as mesmas amostras, mas com dados diferentes, chamamos esse cenário de aprendizado federado vertical (YANG *et al.*, 2019). Por exemplo, imagine que um usuário possua uma conta em um banco e uma outra conta em uma loja online. O banco possui as informações de saldo bancário, comportamento das despesas e classificação de crédito, já a loja guarda o histórico de compra e navegação online do usuário. Assim as duas empresas podem se unir para a construção de um modelo preditivo para dizer os produtos que esse usuário poderia vir a comprar. Assim o aprendizado federado vertical é um processo de agregar diferentes informações e realizar o treinamento do erro e dos gradientes de uma forma que garanta a privacidade ao usar dados de empresas distintas de forma colaborativa (YANG *et al.*, 2019).

Por fim temos o *Federated Transfer Learning* que é o cenário quando dois conjuntos de dados são completamente diferentes, tantos nas amostras quanto nas características (YANG *et al.*, 2019). Tal cenário foi concebido para generalizar o aprendizado federado e assim ter uma aplicação mais ampla para uma pequena interseção de usuários e características em comum. A Figura 6 ilustra as três categorias de aprendizado federado descritas acima.

É importante ressaltar, que apesar da garantia da privacidade, o aprendizado federado tem alguns tipos de adversários que são todos aqueles usuários ou entidades que por algum motivo teviram acesso à uma informação do sistema que não deveria ter (RAMOS *et al.*, 2021). Desse modo, podemos dividir os tipos de adversários em duas categorias: Os semi-honestos e os

Figura 6 – Categorias de aprendizado federado



Fonte: YANG et al. (2019)

maliciosos (YANG *et al.*, 2019).

3.2.2 Problemas do aprendizado federado

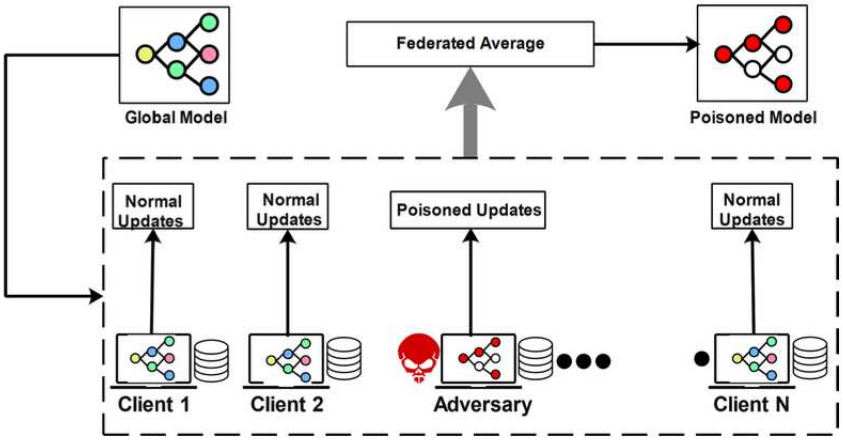
Os adversários maliciosos são aqueles que possuem intenção de sabotar o funcionamento do algoritmo, seja roubando informações privadas ou injetando erros nos modelos. Tais adversários podem burlar os protocolos estabelecidos e agir de modo inesperado para o sistema, já os adversários semi-honestos são aqueles que cumprem honestamente as regras estabelecidas pelo sistema/software, mas por curiosidade aprendem mais informações sensíveis do que deveriam (RAMOS *et al.*, 2021).

Uma forma de comprometer a precisão do modelo global é a presença de um usuário malicioso tentando executar um ataque originado de um cliente chamado de bizantino. Tal ataque acontece quando o usuário envia de maneira proposital falsos gradientes para o servidor central e ao ser realizado o algoritmo FedAvg, esses gradientes irão afetar a precisão do modelo (CHE *et al.*, 2022), ou seja, tem por objetivo convergir um modelo correto em um modelo incorreto e possui uma alta probabilidade de ocorrência. Esse tipo de ataque é chamado de ataque de envenenamento de modelos.

A Figura 7 ilustra esse tipo de ataque. Em um cenário real temos vários clientes

honestos enviando dados verdadeiros, mas podemos ter um usuário que envia dados falsos com o objetivo de inviabilizar o treinamento.

Figura 7 – Ataque de envenenamento de modelos



Fonte: QAMMAR et al, (2022)

Além desse tipo de ataque, sistemas de aprendizado federado podem sofrer outros tipos de ataques, alguns deles são:

- Ataques de inferência de associação: tipo de ataque que verifica se uma determinada amostra está presente no conjunto de dados utilizados no treinamento do modelo. Por exemplo, é possível um adversário inferir se a localização de um perfil foi usada no modelo de classificação de sexo no conjunto de dados de localização do FourSquare. (MELIS *et al.*, 2018). Nesse tipo de ataque, em cada iteração, o adversário recebe o modelo global, calcula a diferença dos gradientes e envia o modelo local atualizado para o servidor central. Depois, o adversário salva os parâmetros do modelo e calcula a diferença entre sucessivos modelos globais (QAMMAR *et al.*, 2022).
- Ataques de reconstrução: possuem como objetivo extrair informações dos dados brutos dos clientes no modelo federado ou das características que os representam. Como no modelo federado cada usuário treina localmente o modelo e apenas os gradientes são compartilhados, novos ataques vêm surgindo para inferir informações dos dados utilizados para realizar o treinamento local (RAMOS *et al.*, 2021).
- Ataques de inversão de modelo: neste tipo de ataque, um adversário tenta reconstruir os dados originais de treinamento, analisando os parâmetros do modelo. Isto pode ser feito enviando consultas ao modelo e observando a saída do modelo, e depois usando esta informação para reconstruir os dados de treinamento que foram usados para treinar o

modelo.

- Ataque de reidentificação: tem como objetivo desanonimizar informações de identificação pessoal nos dados de usuários utilizados no treinamento do modelo. Mesmo que as features sensíveis sejam removidas anteriormente, adversários se utilizam de conhecimentos prévios ou cruzamento de diferentes bases de dados para obter tais informações (RAMOS *et al.*, 2021).

3.2.3 Técnicas de proteção aos ataques

Apesar de o aprendizado federado ser algo recente, já vemos na literatura algumas técnicas para evitar alguns tipos de ataques. Cada técnica busca solucionar alguns ataques e não consegue proteger todos os tipos de ataque, assim, em um cenário ideal, teríamos uso de mais de uma técnica para garantir um nível satisfatório de segurança.

A privacidade diferencial é uma técnica que pode ajudar a proteger modelos de aprendizagem federados de ataques de inferência e de reidentificação. Nesta técnica, um ruído é adicionado às atualizações enviadas pelo cliente, com o objetivo de evitar que o atacante inferir informações sensíveis sobre os dados locais. Uma vez que o ruído é conhecido pelo sistema, é possível levá-lo em conta no modelo e assim garantir que a inferência continue aproximadamente válida (RAMOS *et al.*, 2021).

Uma segunda técnica para proteção é a *Secure Multi-party computation* (MPC) uma técnica criptográfica que permite às múltiplas partes calcular em conjunto uma função sobre suas entradas privadas sem revelar suas entradas umas às outras. No entanto, um problema dessa técnica é o fato de exigir muito tempo de treinamento, o que pode acabar resultando em perda de dados em um contexto federado (MOTHUKURI *et al.*, 2021).

No contexto da aprendizagem federada, o MPC pode ser usado para permitir que os clientes participantes colaborem de forma segura no treinamento de um modelo de aprendizagem de máquinas sem compartilhar seus dados brutos entre si.

Para proteger um sistema de aprendizado federado pode-se utilizar também a criptografia homomórfica que uma técnica criptográfica que permite que a computação seja realizado em dados criptografados sem a necessidade de descriptografá-los primeiro, dessa forma, essa técnica pode ser usada para permitir que os clientes participantes codifiquem seus dados locais e os envie a um servidor central para processamento, sem revelar seus dados ao servidor central ou a outros clientes participantes.

3.3 Computação sem servidor

A computação sem servidor é um modelo de computação em nuvem no qual o provedor da nuvem gerencia a infra-estrutura e os recursos necessários para executar e escalar aplicações. Isto significa que os desenvolvedores não precisam se preocupar com provisionamento, gerenciamento ou escalonamento de servidores ou infra-estrutura (HASSAN *et al.*, 2021).

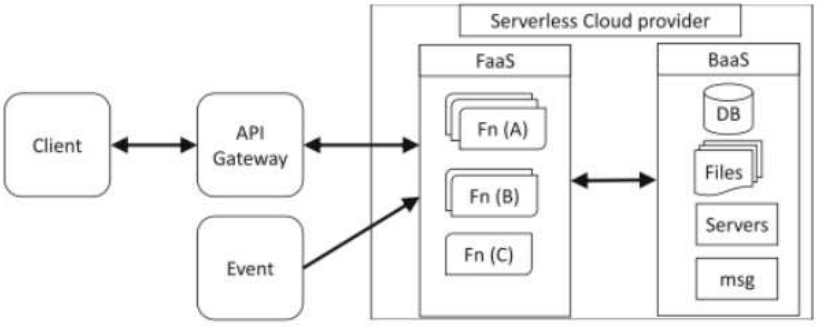
A computação em nuvem sem servidor oferece *Backend as a Service* (BaaS) e FaaS, como ilustra a Figura 8. O BaaS inclui serviços como armazenamento, envio de mensagens, gerenciamento de usuários, etc. Enquanto isso, o FaaS permite aos desenvolvedores implantar e executar seu código em plataformas de computação. O FaaS depende dos serviços fornecidos pelo BaaS, tais como banco de dados, mensagens, autenticações de usuários, etc. O FaaS é considerado como o modelo mais dominante de computação sem servidor, e é também conhecido como funções orientadas a eventos (HASSAN *et al.*, 2021).

Segundo a AWS, um dos maiores fornecedores de serviços de computação sem servidor, a computação sem servidor permite que você construa e execute aplicações e serviços sem pensar em servidores. Ela elimina tarefas de gerenciamento de infra-estrutura como provisionamento de servidor ou *cluster*, *patching*, manutenção do sistema operacional e provisionamento de capacidade.

Em uma arquitetura sem servidor do tipo FaaS, as aplicações são divididas em pequenas funções independentes que são acionadas por eventos específicos, tais como um usuário clicando em um botão, um arquivo de dados sendo carregado ou a realização de uma requisição Web, como pode ser visto na Figura 8. Estas funções são escritas utilizando uma determinada linguagem de programação, carregadas no provedor da nuvem e depois executadas em resposta a eventos, com o provedor da nuvem automaticamente escalando recursos para cima ou para baixo conforme necessário. É importante ressaltar que tais funções não guardam estado, ou seja, ao final de cada execução todas as informações são perdidas (HASSAN *et al.*, 2021), com isso é sempre necessário salvar o que foi processado em algum serviço de armazenamento persistente.

A linguagem de programação utilizada para escrever essas pequenas funções depende do conhecimento do programador e do provedor da nuvem, já que algumas algumas linguagens são permitidas serem utilizadas. Caso, seja necessário executar uma função em uma linguagem não suportada, é possível utilizar contêineres para solucionar esse problema.

Figura 8 – Arquitetura sem servidor



Fonte: HASSAN et al, 2021

Os quatro principais provedores que oferecem computação sem servidores são: AWS, Microsoft Azure, *Google Cloud Platform* (GCP) e IBM Cloud (MAISSEN *et al.*, 2020).

Tabela 2 – Linguagens suportadas e suas respectivas versões

	AWS	Azure (Linux)	Azure (Windows)	GCP	IBM
Node.js	10.x e 12.x	8.x, 10.x e 12.x	8.x, 10.x e 12.x	6.x, 8.x e 10.x	8.x e 10.x
Python	2.7, 3.6, 3.7 e 3.8	3.6, 3.7 e 3.8	-	3.7	2.7, 3.6 e 3.7
Go	1.11 e 1.13	-	-	1.11 e 1.13	1.11
.NET Core	2.1	2.2 e 3.1	2.2 e 3.1	-	2.2
Java	8 e 11	8	8	-	8
Ruby	2.5 e 2.7	-	-	-	2.5
Swift	-	-	-	-	4.2
PHP	-	-	-	-	7.3

Fonte: Adaptado de MAISSEN et al., (2020).

Os provedores, apesar de fornecerem serviços considerados com FaaS, possuem diferenças entre si. A mais importante são as linguagens suportadas (Tabela 2). Eles podem se diferenciar, também, em características como, por exemplo, número máximo de funções, número máximo de execuções simultâneas, tempo máximo de uma execução, modelo de precificação e eventos de disparo de execução (MARTINS *et al.*, 2020).

Além de disponível para uso nos grandes provedores de nuvem é possível utilizar uma solução de código aberto para a computação sem servidor, tal solução é chamada de *Apache OpenWhisk*. A maior diferença ao utilizar essa solução de código aberto é que primeiro será preciso hospedar e configurar tal serviço. Após feito isso, a gestão dos recursos das funções utilizadas é realizada pelo *Apache OpenWhisk* e é possível utilizar várias linguagens de programação, como exemplo, Node.js, Python, Java, Swift e outras.

Atualmente vemos empresas utilizando arquitetura FaaS para solucionar diversos tipos de problemas, como, por exemplo, a criação de *chatbots*, processamento de arquivos,

sistemas de detecção de ameaças, computação de borda e outros(HASSAN *et al.*, 2021).

Uma das principais vantagens da computação sem servidor é sua escalabilidade. Como o provedor da nuvem é responsável pelo escalonamento de recursos para cima ou para baixo com base na demanda, as aplicações podem lidar automaticamente com picos de tráfego sem a necessidade de intervenção manual. Isto também a torna uma solução econômica, pois os usuários são cobrados apenas pelo uso real dos recursos, em vez de pagar por uma quantidade fixa de infra-estrutura.

Outra vantagem da computação sem servidor é sua facilidade de uso. Os desenvolvedores podem se concentrar em escrever código para suas funções específicas, em vez de se preocupar com o gerenciamento da infra-estrutura. Isto pode levar a ciclos de desenvolvimento mais rápidos e, consequentemente, a um menor tempo para pôr a aplicação disponível para o usuário final.

Entretanto, a computação sem servidor também tem algumas limitações. Por exemplo, como cada função é executada isoladamente, pode haver um aumento da latência quando as funções precisam se comunicar umas com as outras. Além disso, arquiteturas sem servidor podem não ser as mais adequadas para certos tipos de aplicações, tais como aquelas com processos de longa duração, pois cada função tem um tempo limitado e curto de execução e como elas não guardam estado, não é possível recuperar as informações de uma execução passada (HASSAN *et al.*, 2021)

Atualmente, não existem muitas ferramentas de depuração e monitoramento de funções sem servidor (HASSAN *et al.*, 2021). Isso é um importante problema, já que a depuração do código é um excelente mecanismo para entender como o código se comporta e assim se facilita encontrar e corrigir *bugs*. Com isso, são necessários *Integrated Development Environment* (IDE) mais avançados, para os desenvolvedores poderem refatorar as funções, tais como fusão ou divisão de funções, e reversão de funções para a versão anterior (HASSAN *et al.*, 2021).

Outro problema é o chamado *cold start* que acontece quando uma função leva mais tempo do que o normal para ser executada. Na maioria das vezes, isto ocorre quando a função é invocada logo após a conclusão de sua implantação, ou quando a função não é utilizada há algum tempo, por exemplo, após 10 minutos sem invocações (MAISSEN *et al.*, 2020). Com isso, o provedor de nuvem remove todas as instâncias daquela função e então quando for necessário a sua execução teremos um tempo maior de latência, pois o provedor necessitará criar novamente uma nova instância para aquela função.

Ao trabalhar com funções sem servidor, o gerenciamento de dependências, ou biblioteca, pode ser um desafio (PETROSYAN; ASTSATRYAN, 2022), uma vez que os métodos tradicionais de instalação de dependências em um servidor não são aplicáveis em uma arquitetura sem servidor. Uma forma de resolver esse problema é a utilização de contêineres. Deve-se prestar atenção, para casos de linguagens compiladas, como, por exemplo, *Go*, o ambiente em que o programa foi compilado, pois mesmo que seja uma arquitetura sem servidor, a função é executado em uma máquina com um processador e um sistema operacional específico, que normalmente são processadores da família x86 executando alguma distribuição do Linux.

3.3.1 Contêineres

Um contêiner é uma instância executável de uma imagem que encapsula um programa junto com suas bibliotecas, dados, arquivos de configuração, e outros aspectos, em um ambiente isolado, portanto pode garantir a compatibilidade da biblioteca e permite aos usuários mover e implantar programas facilmente entre os clusters (ZHOU *et al.*, 2022), ou seja, oferecem uma maneira de isolar as aplicações e diminuem as preocupações com questões de compatibilidade.

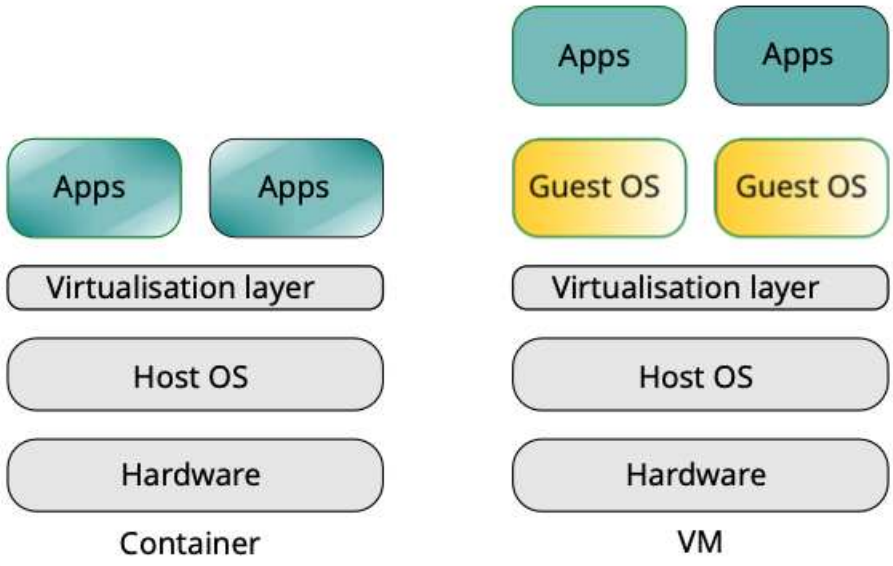
Os contêineres também são muito mais eficientes do que as VM tradicionais, pois podem compartilhar o mesmo kernel do sistema operacional e usar menos recursos, como é ilustrado na Figura 9. Ao contrário, uma VM tradicional carrega todo um kernel convidado na memória, que pode ocupar *gigabytes* de espaço de armazenamento e requer uma fração significativa dos recursos do sistema para funcionar (ZHOU *et al.*, 2022).

Uma das ferramentas disponíveis que entrega uma interface para o uso de contêineres é o *Docker*[1] está entre as ferramentas de contêinerização mais utilizadas atualmente. É possível utilizar o *Docker* em diversos sistemas operacionais e suas imagens de contêineres são composta de uma camada que são permitido operações de leitura e escrita acima de uma série de camadas somente de leitura.

Docker proporciona isolamento e comunicação de rede criando três tipos de redes: *host*, *bridge* e nenhuma. A rede do tipo *brigde* é a padrão. O motor *Docker* cria um *gateway* para a rede *bridge*. Este software permite que os contêineres se comuniquem dentro da mesma rede *bridge*; enquanto isso, isola os contêineres de uma rede *bridge* diferente (ZHOU *et al.*, 2022). Os contêineres na mesma máquina podem se comunicar através da rede padrão pelo endereço IP

[1] https://www.docker.com/

Figura 9 – Comparativo em VM e Contêineres



Fonte: ZHOU et al, 2020

da máquina.

Algumas pessoas podem pensar que a revolução da computação sem servidor é uma ameaça aos contêineres. Isto porque eles percebem as funções sem servidor como uma forma ainda mais eficiente de implementar o código de aplicação do que os contêineres *Docker*. Tais contêineres reduzem a carga de gerenciamento associada às máquinas virtuais e também proporcionam maior escalabilidade, mas não eliminam por completo. As funções sem servidor eliminam a necessidade de configurar e gerenciar qualquer tipo de infra-estrutura, além da tarefa básica de carregar o código para função sem servidor na plataforma *serverless* (JAMBUNATHAN; YOGANATHAN, 2018).

Porém funções sem servidor não significa que o *Docker* não é mais necessário (JAMBUNATHAN; YOGANATHAN, 2018) já que podemos implantar funções na forma de contêineres, isso significa transformar todo o código e as dependências necessárias para a execução da função em uma imagem de contêiner. Isso é extremamente vantajoso quando temos uma função complexa, que necessita de uma variedade de dependências ou quando desejamos utilizar uma linguagem não aceita pelo provedor de nuvem, é possível criar uma imagem para a função e realizar a implantação dessa função.

Por exemplo, o provedor de nuvem da *Amazon* possui um serviço chamado *Elastic Container Registry* (ECR) que permite armazenar, gerenciar e controlar as versões de imagens de contêineres *Docker* em um ambiente privado, seguro e altamente disponível. Isto permite que você implante facilmente suas aplicações de contêineres em ambientes de produção, utilizando

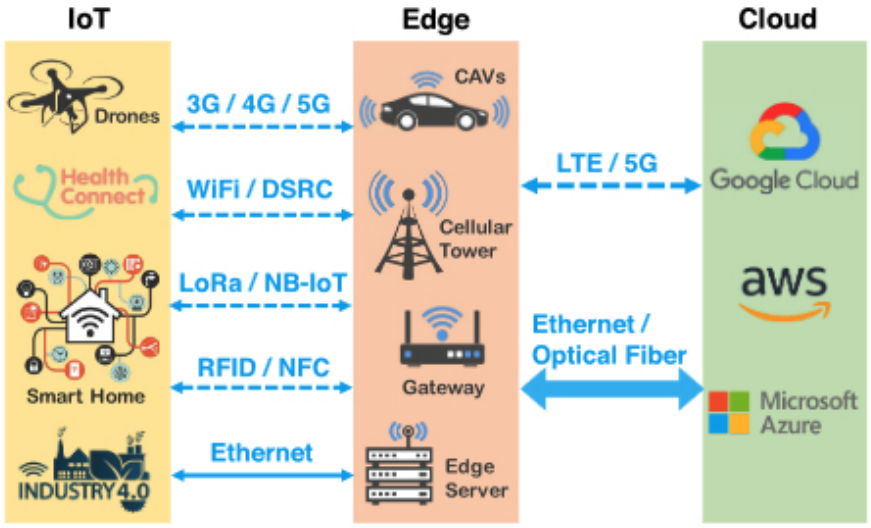
uma variedade de serviços AWS, tais como *Amazon Lambda*, um serviço para executar funções sem servidor.

Ao hospedar uma imagem de contêiner no ECR é possível implantar uma função fazendo uso desta imagem de forma muito simples, com apenas um clique é possível ou pode-se utilizar a *Command Line Interface* (CLI) para fazer isso de forma automatizada.

3.4 Computação na borda da rede

A computação de borda refere-se ao conceito de realizar tarefas computacionais na proximidade ou na borda da rede. Em vez de transferir todos os dados para um servidor central ou para a nuvem a fim de serem processados, a computação de borda envolve a distribuição do processamento para dispositivos de borda ou nós intermediários na rede. Ao aproximar a computação da borda da rede, é possível reduzir a latência e aprimorar a capacidade de processamento em tempo real. Isso possibilita a análise de dados e a tomada de decisões mais rápidas, o que é especialmente vantajoso em aplicativos sensíveis ao tempo, como IoT, sistemas autônomos e análise na borda (SHI *et al.*, 2019).

Figura 10 – Computação na borda e na nuvem



Fonte: SHI et al, 2019

A computação de borda e a computação em nuvem não são relações de substituição, mas sim complementares (SHI *et al.*, 2019), como mostra a Figura 10. Pode-se perceber, analisando a Figura 10, que a borda tem o papel de realizar o processamento dos dado mais próximo do usuário e, quando necessário, os dispositivos da borda da rede solicitam os recursos

da nuvem.

A onipresença de dispositivos inteligentes e o rápido desenvolvimento de tecnologias modernas de virtualização e nuvem trouxeram a computação de borda para o primeiro plano, definindo uma uma nova era na computação em nuvem. Os nodos de borda são geralmente mais restritos em recursos computacionais (CPU, memória, dentre outros) quando comparados aos nodos de nuvem. Portanto, a computação de borda precisa de grande poder computacional e suporte de armazenamento massivo de um centro de computação em nuvem, e a computação em nuvem também precisa do modelo de computação de borda para processar dados em massa e dados privados (SHI *et al.*, 2019).

4 PROPOSTA

Nesse trabalho é apresentada uma arquitetura, chamada de AAFFS, para a execução do aprendizado de máquina federado usando funções como serviço. Posteriormente, desenvolveu-se uma aplicação para validar a viabilidade da AAFFS. A arquitetura é compatível com plataformas distintas, com dispositivos heterogêneos e é capaz de executar modelos redes neurais arbitrárias.

4.1 Funcionalidades

Durante a pesquisa, procurou-se um conjunto de funcionalidades que deveriam ser requisitos na AAFFS para que ela possa ser aplicada em uma variedade de ambientes distintos. Primeiro, foi requisitado que a arquitetura fosse compatível com mais de um provedor de função como serviço, pois no mercado tem-se quatro provedores proprietário e alguns softwares de código aberto. Além disso, deveria ser capaz de funcionar com dispositivos distintos, pois uma característica comum do Aprendizado Federado é a participação de dispositivos diferentes.

É interessante, também, que a AAFFS possa funcionar com uma variedade de modelos de aprendizado de máquina, pois cada modelo objetiva resolver um problema específico, então para que a arquitetura seja genérica e possa funcionar com diversos tipos de problemas, ela deve ser capaz de funcionar com uma variedade de modelos. Por fim, para casos que seja necessário um nível de privacidade ainda maior ou em casos que o consumo de banda seja um característica impeditiva para o projeto, é interessante que a AAFFS funcione na borda da rede.

Com isso, as funcionalidades principais definidas para a AAFFS são:

1. **Compatibilidade com diferentes provedores de função como serviço:** a arquitetura proposta é compatível com uma variedade de provedores, sendo possível a utilização de ferramentas proprietárias ou de código aberto. Nos testes realizados, desenvolvemos uma aplicação compatível com dois provedores de função como serviço distintos.
2. **Compatibilidade com dispositivos clientes heterogêneos:** uma das características do aprendizado federado é a presença de dispositivos heterogêneos disponíveis para a realização do treinamento (RAMOS *et al.*, 2021), por exemplo, podem participar de um treinamento dispositivos como *desktop* tradicionais, *Raspberry Pi* ou outras funções como serviço. A arquitetura proposta pode ser

executada utilizando uma variedade de dispositivos utilizando processadores da família *x86* ou *ARM*.

3. **Compatibilidade com diferentes modelos de aprendizado de máquina:** nossa proposta não limita qual modelo de aprendizado de máquina pode ser utilizado, ou seja, ela é genérica e pode ser aplicada numa grande variedade de modelos. O limite será definido pelos recursos dos dispositivos utilizados no treinamento. Tais informações são: conjunto de dados a ser treinado, total de clientes, fração de clientes que deverão realizar o treinamento por rodada.
4. **Compatibilidade de orquestrar o treinamento na borda:** com o objetivo de colocar o processamento perto da entidade que o solicita para reduzir a latência, a arquitetura proposta é capaz de realizar o treinamento na borda da rede.

Além das funcionalidades principais, ferramentas de observabilidade é uma funcionalidade opcional e podem ser utilizadas na AAFFS para o monitoramento das entidades e dos recursos utilizados pelos dispositivos. Por exemplo, é possível visualizar métricas de uso da função de agregação ou, no caso do uso de uma ferramenta que seja hospedado pelo usuário, é possível visualizar métricas de uso dos recursos da máquina. Tais métricas podem ser visualizadas através do uso de banco de dados de séries temporais e de ferramentas para criação de painéis de controle.

A compatibilidade de orquestrar o treinamento na borda foi um requisito escolhido para obter um nível ainda maior de privacidade. Quando a computação é inserida na borda, os dados não necessitam trafegar até um servidor de computação em nuvem centralizado, eles trafegam apenas até a entidade de processamento que está próximo do usuário. Em casos específicos, quando o processamento é inserido na mesma rede do usuário, o dado não trafega para fora da rede, aumentando a privacidade e diminuindo a largura de banda consumida.

Com isso, neste trabalho, para exemplificar e testar o uso da AAFFS na borda da rede, foram realizados testes utilizando o *Raspberry Pi* como entidade responsável por realizar o processamento dos dados e tal dispositivo foi inserido na mesma rede do usuário.

4.2 Arquitetura

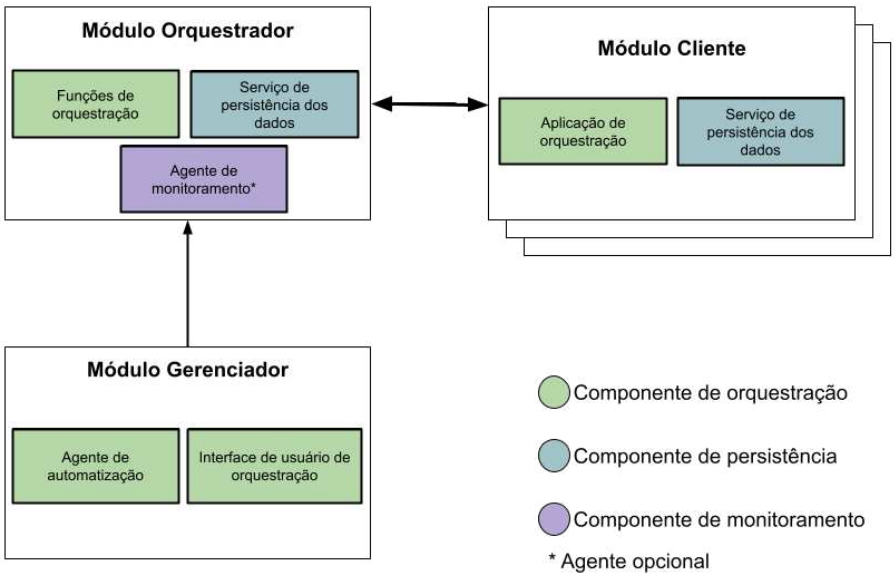
A Figura 11 apresenta a forma genérica da AAFFS proposta por esse trabalho. Tal arquitetura é composta por 3 módulos principais obrigatórios e cada um tem um papel específico e bem definido, sendo eles:

- **Módulo Orquestrador**: tem como função principal orquestrar o treinamento federado. Este módulo deverá ser implementado como uma função como serviço. Dentre as suas responsabilidades, ele deverá escolher o conjunto de clientes que realizará o treinamento em cada rodada, executar o algoritmo de agregação e salvar o modelo global atualizado. Esse módulo é composto também do serviço de persistência de dados e tem a possibilidade de apresentar um agente de monitoramento.
- **Módulo Gerenciador**: tem como função a implementação da função como serviço que orquestrará o treinamento federado. É responsável por iniciar o treinamento ao enviar um evento para o módulo orquestrador informando com as informações do treinamento.
- **Módulo Cliente**: é o detentor dos dados e tem como função principal realizar o treinamento local com seus dados e gerar um novo modelo atualizado localmente. Após realizado o treinamento, tem como responsabilidade o envio dos pesos do modelo atualizado. Possui uma entidade para salvar os dados que serão utilizados no treinamento.

Nesta arquitetura as iterações do FL acontecem no módulo Orquestrador e no módulo Cliente. Cada rodada do Aprendizado Federado é iniciado no módulo Orquestrador, onde os as instâncias do módulo Cliente são escolhidas. Após essa escolha, as iterações de aprendizado local são executadas em cada instância do módulo Cliente.

Os componentes de módulos distintos comunicam-se entre utilizando requisições *Hypertext Transfer Protocol* (HTTP). Os componentes distintos de um mesmo módulo utilizando requisições HTTP ou utilizando um protocolo específico, como o *MongoDB Wire Protocol* (MONGODB INC., 2023b).

Figura 11 – Módulos da AAFFS



Fonte: Elaborado pelo autor (2023)

5 MATERIAIS E MÉTODOS

Esta seção apresenta as ferramentas utilizadas para implementar a AAFFS. Ela está dividida de acordo com as três entidades principais apresentadas da arquitetura. Nesta seção, também é apresentada a implementação de uma aplicação utilizando a AAFFS.

5.1 Módulo Orquestrador

A AAFFS tem o objetivo de ser genérica, com isso, para provar que a proposta atinge tal objetivo, foi implementado o Módulo Orquestrador em mais de uma forma e apresenta-se nessa seção as ferramentas utilizadas em cada cenário e em qual componente no módulo ela pertence.

5.1.1 AWS Lambda

No primeiro cenário de testes, foram implementadas as funções de orquestração do módulo orquestrador utilizando um serviço da AWS chamado de *Lambda*. Tal ferramenta é um serviço de função como serviço e é um tipo de arquitetura orientada a eventos. Para a aplicação, o evento escolhido para se iniciar a execução da função de orquestração é uma requisição utilizando o protocolo HTTP e nesse cenário de teste foi implementado apenas uma função e associado várias rotas, cada responsável por uma funcionalidade da orquestração.

Como a arquitetura de funções como serviço não guardam estado, ou seja, cada execução é independente, é preciso a utilização de um serviço de armazenamento para persistência dos dados. Assim, em cada execução, a função como serviço conecta-se com o banco de dados para recuperar as informações necessárias para sua execução. A arquitetura proposta não especifica qual tecnologia de banco de dados utilizar, basta que seja possível realizar a comunicação entre a função como serviço e o banco de dados.

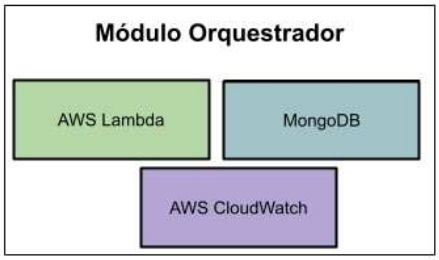
Para realizar a implantação de função de orquestração, foi utilizado o serviço ECR da AWS. Tal serviço é um repositório de imagens contêineres que pode ser utilizado em diversos outros serviços e um deles é poder realizar a implantação de funções como serviço. Para isso, foi criado uma imagem *Docker* da função de orquestração e salva no ECR.

Nos testes, foi utilizado o *MongoDB*, que é um sistema de banco de dados NoSQL popular, de código aberto e orientado a documentos. Ele foi projetado para oferecer alto desempenho, escalabilidade e flexibilidade para armazenar, consultar e manipular grandes

volumes de dados estruturados e semiestruturados (MONGODB INC., 2023a).

Finalmente, para a entidade de monitoramento, que é opcional, foi utilizado o *CloudWatch*[1] que é uma ferramenta da AWS que coleta e visualiza *logs*, que são os registros do sistema, métricas e dados de eventos em tempo real em painéis automatizados para otimizar sua infraestrutura e manutenção de aplicações (AMAZON WEB SERVICES INC, 2023). A Figura 12 apresenta os componentes do módulo orquestrador utilizados no testes quando utilizamos o *Lambda* como função de orquestração.

Figura 12 – Módulos orquestrador utilizando AWS



Fonte: Elaborado pelo autor (2023)

5.1.2 Kubernetes e Apache OpenWhisk

O segundo cenário de testes se configura com a utilização de uma máquina virtual e a utilização de uma ferramenta de código aberto para realizar o papel de funções de orquestração. Nesse teste foi utilizado uma máquina com processador *Intel Core i5* com 16 *gigabytes* de memória *Random Access Memory* (RAM).

Nesse cenário, foi utilizado o *Apache OpenWhisk*, que é uma ferramenta de código aberto para a implantação e execução de funções como serviço. Tal ferramenta pode ser instalada nativamente em qualquer computador com uma distribuição Linux instalada ou pode ser configurada através de contêineres. Nos testes optamos pela segunda opção, pois a gerência desses contêineres podem ser facilitadas com o uso de uma outra ferramenta de código aberto chamado de *Kubernetes*[2].

O *Kubernetes* é um sistema de código aberto desenvolvido para automatizar o provisionamento, escalabilidade e gerenciamento de serviços em contêineres. Originalmente projetado pela Google, essa ferramenta é atualmente mantida pela Cloud Native Computing Foundation (KUBERNETES AUTHORS, 2023c).

[1] https://aws.amazon.com/pt/cloudwatch/
[2] https://kubernetes.io/pt-br/

Segundo a documentação do *Kubernetes* (KUBERNETES AUTHORS, 2023b) a ferramenta é composta por diversos componentes que podem ser divididos em três categorias principais chamados de: *Control Plane Components*, *Node Components* e *Addons*.

O *Control Plane Components* é responsável pelas decisões globais sobre o *cluster*, bem como detectam e respondem a eventos. Tais componentes podem ser executados em qualquer máquina presente no *cluster*. É composto por:

- ***Application Programming Interface* (API) *Server*:** atua como o principal ponto de gerenciamento e plano de controle para o *cluster* do *Kubernetes*. Ele expõe a API do *Kubernetes*, que é usada por outros componentes e usuários para interagir com o *cluster*.
- ***Scheduler*:** responsável pela atribuição de contêineres a nós com base na disponibilidade de recursos, nos requisitos do aplicativo e em várias políticas.
- ***Etcd*:** é um armazenamento distribuído de valores-chave que armazena os dados de configuração do *cluster*, fornecendo um armazenamento de dados confiável para o estado do *cluster*.
- ***Controller Manager*:** responsável por gerenciar vários controladores que lidam com diferentes aspectos do *cluster*, como o ciclo de vida do nó e do *pod*, a replicação e os *endpoints*.

Um *pod* é a menor e mais básica unidade de implantação. Ele representa uma única instância de um processo em execução em um *cluster*. Um *pod* encapsula um ou mais contêineres fortemente acoplados, recursos de armazenamento e componentes de rede que são programados juntos em um único nó de trabalho.

O *Node Components* são executados em cada nó, mantendo *pods* em execução e fornecendo o ambiente de tempo de execução do *Kubernetes*. Os principais componentes dele são:

- ***Kubelet*:** é executado em cada nó e se comunica com os componentes principais. Ele gerencia os contêineres e seu ciclo de vida, garantindo que os contêineres especificados estejam em execução e saudáveis no nó.
- ***Container Runtime*:** responsável pela execução de contêineres, como *Docker*, *containerd* ou *CRI-O*. Ele interage com o *kubelet* para iniciar, parar e monitorar contêineres.
- ***Kube Proxy*:** lida com a comunicação de rede para serviços e realiza o balancea-

mento de carga entre os contêineres de um serviço.

Por fim, os *Addons* são componentes ou serviços adicionais que podem ser implantados em um *cluster* do *Kubernetes* para aprimorar sua funcionalidade e fornecer recursos adicionais. Esses complementos geralmente são opcionais e podem ser instalados com base nos requisitos específicos de seus aplicativos. Aqui estão alguns *addons* do *Kubernetes* comumente usados:

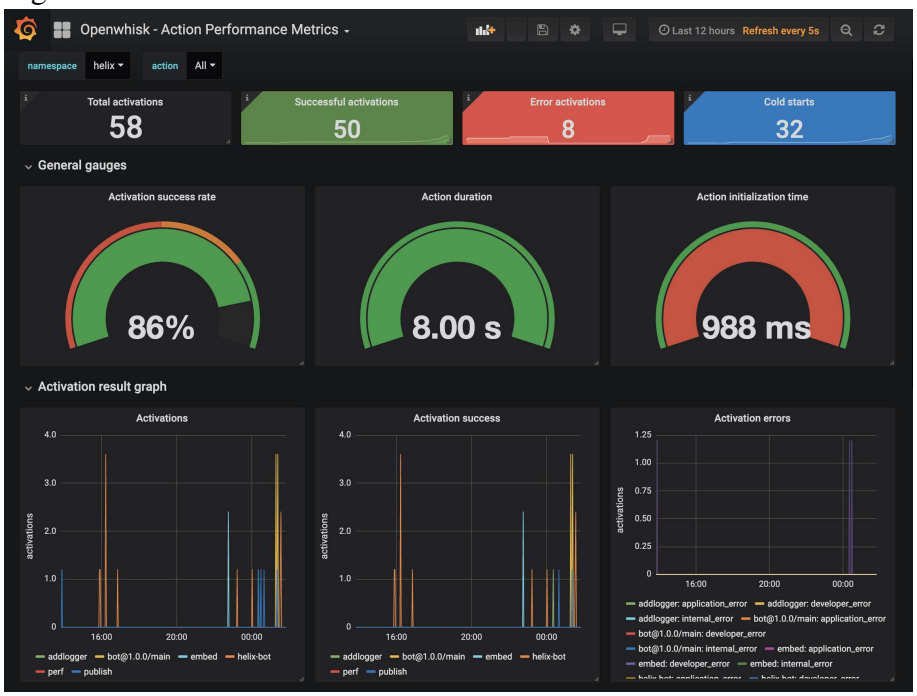
- ***CoreDNS*:** fornece descoberta de serviços dentro do *cluster* mapeando os nomes dos serviços para seus endereços IP correspondentes. Ele permite que contêineres e serviços se comuniquem entre si usando nomes fáceis de lembrar, em vez de lidar com endereços IP de baixo nível.
- ***Dashboard***: interface de usuário baseada na Web que fornece uma representação gráfica do *cluster*. Ele permite que os administradores gerenciem e monitorem aplicativos, visualizem a utilização de recursos e realizem operações básicas, como criar e dimensionar implantações, gerenciar serviços e inspecionar *logs*.
- ***Ingress Controller***: API que gerencia o acesso externo aos serviços em um *cluster* do *Kubernetes*. Tal controlador é responsável pela implementação das regras do ingresso e pelo roteamento de solicitações externas para os serviços apropriados.
- ***Metrics Server***: é responsável pela coleta dados de utilização de recursos de nós e *pods* no *cluster*. Ele fornece métricas como uso de CPU e memória, permitindo que você monitore o uso de recursos e ative recursos como escalonamento automático de *pod.*

Para interface de monitoramento, que também é opcional nesse cenário, foi utilizada uma ferramenta de código aberto chamado *Grafana*. Essa é uma ferramenta de análise e visualização que é comumente usada para fins de monitoramento e observabilidade. Ele permite que você consulte, visualize e compreenda várias métricas e dados de diferentes fontes de forma unificada e interativa. O Grafana oferece um rico conjunto de recursos que o tornam uma escolha popular para a criação de painéis personalizáveis e sistemas de monitoramento.

O *Apache OpenWhisk* é compatível com o Grafana e pode ser integrado facilmente para analisar os recursos utilizados pelas funções como serviço. A Figura 13 ilustra um exemplo de painel de controle do Grafana consumidos os dados enviados do *Apache OpenWhisk*.

A Figura 14 apresenta os componentes do módulo orquestrador utilizados no testes

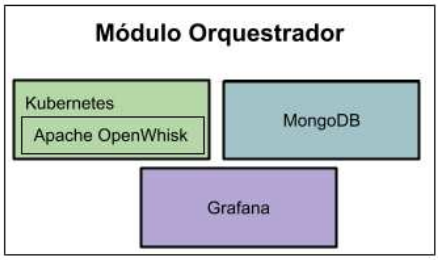
Figura 13 – Painel de controle do Grafana



Fonte: (APACHE OPENWHISK AUTHORS, 2023)

quando a combinação de *Kubernetes* com o *Apache OpenWhisk* como função de orquestração foi utilizada. Nos testes foi usado a ferramenta de observabilidade Grafana para analisar o uso dos recursos de cada função.

Figura 14 – Módulos orquestrador utilizando *Kubernetes* e *Apache OpenWhisk*



Fonte: Elaborado pelo autor (2023)

5.1.3 Raspberry Pi e Apache OpenWhisk

No terceiro e último cenário de testes para a arquitetura, o processamento foi transferido para a borda da rede, ou seja, a função como serviço foi implantada mais próximo do usuário. Isso foi feito para reduzir o tempo de latência, que precisa ser baixo em aplicações que necessitam de uma rápida resposta para tomada de decisão.

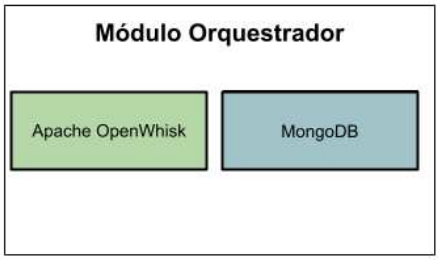
Nesse cenário continuou-se a utilizar o *Apache OpenWhisk*, mas sem o uso do *Kubernetes*. Foi decidido não utilizar uma ferramenta de orquestração de contêineres por falta de

recursos disponíveis no dispositivo que irá executar a função do orquestrador. Foi utilizado um *Raspberry Pi 4* com 4 *gigabytes* de memória RAM e 32 *gigabytes* de armazenamento, utilizando o sistema operacional *Raspberry Pi OS*. Esse dispositivo foi inserido na mesma rede que as entidades possuidoras dos dados e o treinamento foi realizado para a obtenção de métricas de latência.

Diferentemente dos outros cenários, onde apenas uma função como serviço foi implementada, neste último cenário foi necessário a separação das funcionalidades e assim, foram implantadas seis funções como serviço, cada uma associada a um *endpoint* específico e cumprindo um papel definido.

Na pesquisa, não foram realizados testes com ferramentas de observabilidade nesse cenário. Essa decisão foi tomada, pois a placa *Raspberry Pi* tem recursos limitados e ficaria sobrecarregada com a instalação do Grafana. Porém, é possível adicionar uma ferramenta de observabilidade para esse cenário, mas para isso a ferramenta deveria ser instalada e gerenciada em outro dispositivo. A Figura 15 apresenta os componentes presentes no módulo orquestrador utilizados nesses testes.

Figura 15 – Módulos orquestrador utilizando *Raspberry Pi* e *Apache OpenWhisk*



Fonte: Elaborado pelo autor (2023)

5.2 Módulo Cliente

O módulo cliente é composto por um conjunto diversos de aplicações e, consequentemente, de dispositivos. São as entidades deste módulo que possuem os dados utilizados no treinamento federado e a cada rodada de treinamento um subconjunto de clientes são escolhidos para treinar o modelo.

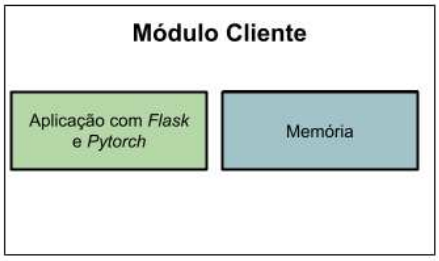
As entidades deste módulo podem ser de vários tipos, como por exemplo: uma aplicação *Python* sendo executada em um *desktop* tradicional, uma aplicação *Python* embarcado em um *Raspberry Pi* ou uma função como serviço desenvolvida em *Python*. Em geral, o cliente

pode ser uma aplicação com conexão a internet, desenvolvida em *Python* e capaz de gerenciar modelos de aprendizado de máquina *Pytorch*.

Os requisitos de como são armazenados os dados nos clientes também são bastante genéricos, podendo ser utilizados uma variedade de bancos de dados. Com exceção de clientes implementados em funções como serviço, os dados podem, também, ser armazenados em memória.

Para o cenário de testes, foram utilizado 50 clientes desenvolvidos em *Python* e utilizando o pacote *Flask* para criar alguns *endpoints* em que a função de orquestração usa para a coordenação do treinamento. Além disso, foi utilizado o pacote *Pytorch* para realizar o aprendizado de máquina em cada cliente e os dados foram guardados em memória. A Figura 16 apresenta os componentes específicos do módulo cliente utilizado em todos os testes.

Figura 16 – Módulos Cliente



Fonte: Elaborado pelo autor (2023)

5.3 Módulo Gerenciador

Por fim, este último módulo tem a responsabilidade gerenciar a implantação das funções de orquestração e coordenar o início do treinamento federado. Para os diferentes cenários, ferramentas específicas se fizeram necessárias para a implantação das funções como serviço.

5.3.1 AWS Lambda

No cenário de testes utilizando as funções como serviço da AWS *Lambda* utilizamos o *framework Serverless*[3]. Essa ferramenta auxilia na implantação de funções como serviço na AWS. Todas as configurações necessárias para implantar uma função são definidas em um arquivo de texto e com apenas um comando é possível implantá-la. Por exemplo, no cenário dessa pesquisa, a função implantada utiliza uma imagem Docker salva no repositório ECR e

[3] https://www.serverless.com/

associada a essa função, são adicionadas seis *endpoints* distintos de gatilhos, cada um fazendo com que a função desempenhe um papel específico. Com o uso desse framework, não foi necessário a configuração manual de todos os serviços listados acima, outro benefício é que se torna mais fácil de replicar o cenário testado.

A Figura 17 ilustra um exemplo de um arquivo de configuração do *framework Serverless*, ao analisá-la, é possível ver que começa definindo um nome para o serviço, depois disso, são definidos as configurações gerais da função como serviço, como por exemplo, memória utilizada e região de implantação. Posteriormente são configurados características específicas da função, nessa figura é demonstrado que será utilizado uma imagem do ECR para implantar a função. A Figura 17 ainda ilustra que dois *endpoints* foram associados à função especificada.

Figura 17 – Exemplo de arquivo de configuração *Serverless*

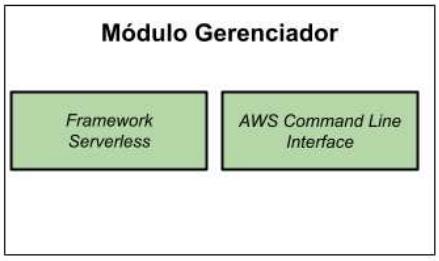
Fonte: Elaborado pelo autor (2023)

Outra ferramenta utilizada foi o CLI da AWS, que precisa ser instalado e configurado no módulo gerenciador para que o *framework Serverless* funcione corretamente. A Figura 18 apresenta os componentes utilizados para esse cenário.

5.3.2 Kubernetes e Apache OpenWhisk

No cenário de testes que foram utilizado o *Kubernetes* e o *Apache OpenWhisk* hospedado em uma máquina virtual, o módulo gerenciador precisou de duas ferramentas para fazer o gerenciamento das duas ferramentas. Para o orquestrador de contêineres, foi preciso

Figura 18 – Módulo gerenciador utilizando AWS *Lambda*



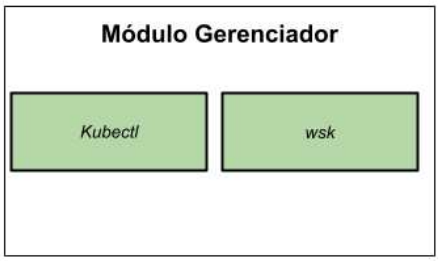
Fonte: Elaborado pelo autor (2023)

instalar e configurar o *kubectl* que é uma ferramenta de CLI que serve como o principal meio de interação com os *clusters* do *Kubernetes*. Ela permite que administradores, desenvolvedores e operadores gerenciem e controlem os recursos do *Kubernetes*, executem várias operações e obtenham informações sobre o estado do cluster (KUBERNETES AUTHORS, 2023a).

Além da ferramenta para o gerenciamento do *cluster* do *Kubernetes*, foi preciso instalar e configurar outra CLI, chamada de *wsk*. Ela é usada para interagir e gerenciar a plataforma sem servidor do *Apache OpenWhisk* e fornece uma maneira conveniente para desenvolvedores e administradores implantarem funções, gerenciarem ações e trabalharem com outros recursos em uma implantação do *Apache OpenWhisk*.

A Figura 19 apresenta os componentes utilizados no módulo gerenciador quando testamos a arquitetura em uma máquina virtual utilizando o *Kubernetes* e o *Apache OpenWhisk*.

Figura 19 – Módulo gerenciador utilizando *Kubernetes* e *Apache OpenWhisk*

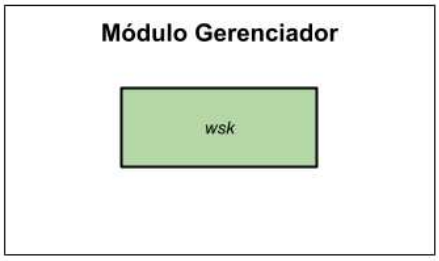


Fonte: Elaborado pelo autor (2023)

5.3.3 Raspberry Pi e Apache OpenWhisk

No terceiro e último cenário, o módulo gerenciador precisou apenas da ferramenta *wsk*, a mesma mencionada no cenário anterior. Neste último cenário, ela tem a mesma responsabilidade que é interagir e gerenciar a plataforma sem servidor do *Apache OpenWhisk*. A Figura 20 ilustra os componentes utilizados nesse cenário.

Figura 20 – Módulo gerenciador utilizando *Raspberry Pi* e *Apache OpenWhisk*



Fonte: Elaborado pelo autor (2023)

5.4 Implementação completa

Após descrito todos os componentes dos três cenários de teste, neste tópico são apresentados, para cada cenário, como os componentes interagem entre si para atingir o objetivo de realizar um treinamento federado. Foi utilizada a AAFFS com os componentes genéricos para apresentar a aplicação, pois em todos os cenários a comunicação entre os componentes de cada módulo são similares. Em todos cenários testados a comunicação é feita utilizando requisições HTTP. O código que simula uma aplicação que utiliza a AAFFS está disponível no repositório: https://github.com/marcelfonteles/federated-learning-serverless.

A Tabela 3 resume todo os cenários de teste e apresenta todos os componentes utilizados nos três cenários de teste do módulo orquestrador.

Tabela 3 – Componentes do módulo orquestrador utilizado nos testes

	Funções de orquestração	Persistência dos dados	Monitoramento
Cenário 1	AWS *Lambda*	*MongoDB*	AWS *CloudWatch*
Cenário 2	*Kubernetes* e *Apache OpenWhisk*	*MongoDB*	*Grafana*
Cenário 3	*Apache OpenWhisk*	*MongoDB*	-

Fonte: Elaborado pelo autor (2023)

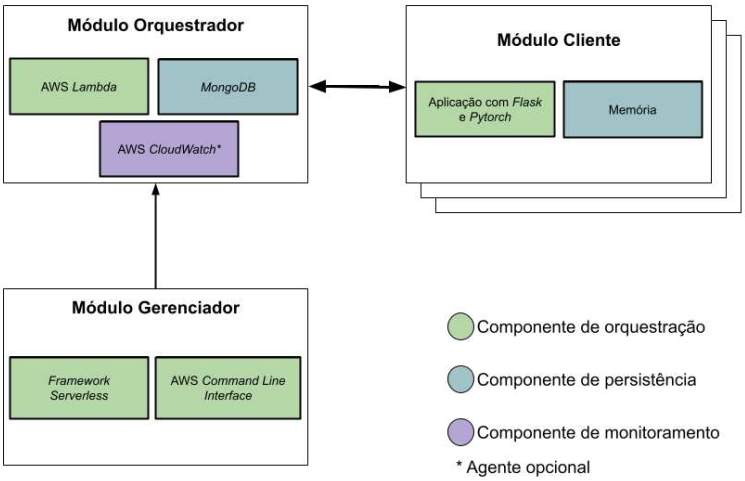
A Figura 21 apresenta todos os componentes específicos utilizados em cada módulo quando o cenário de teste utilizou o AWS *Lambda*.

As Figuras 22 e 23 ilustram, respectivamente, todos os componentes específicos das arquiteturas utilizando uma máquina virtual e utilizando uma *Raspberry Pi*.

5.4.1 Funcionamento geral

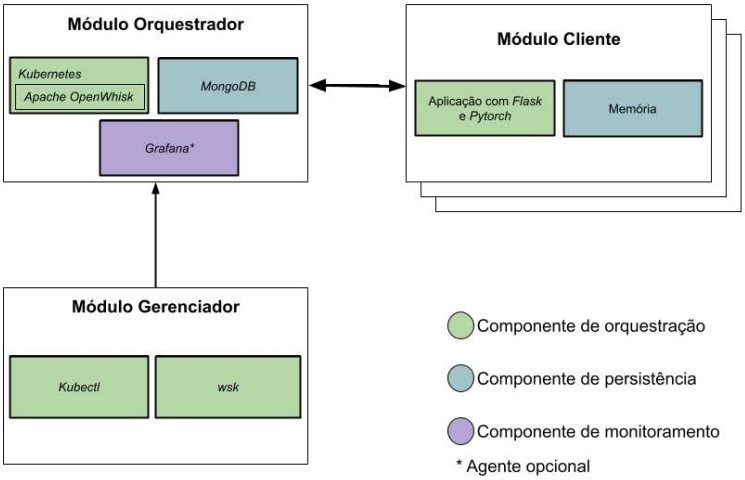
Para os três cenários, os teste são iniciados quando uma requisição é feita a partir do módulo gerenciador para uma função como serviço presente no módulo orquestrador. Essa requisição informa os parâmetros do treinamento, como o *dataset* a ser treinado, a quantidade de

Figura 21 – Arquitetura usando AWS



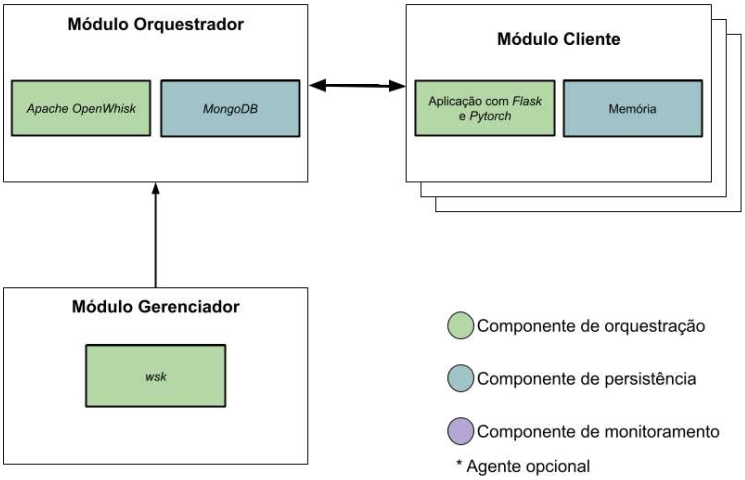
Fonte: Elaborado pelo autor (2023)

Figura 22 – Arquitetura usando *Kubernetes* e *Apache OpenWhisk*



Fonte: Elaborado pelo autor (2023)

Figura 23 – Arquitetura usando *Raspberry Pi* e *Apache OpenWhisk*

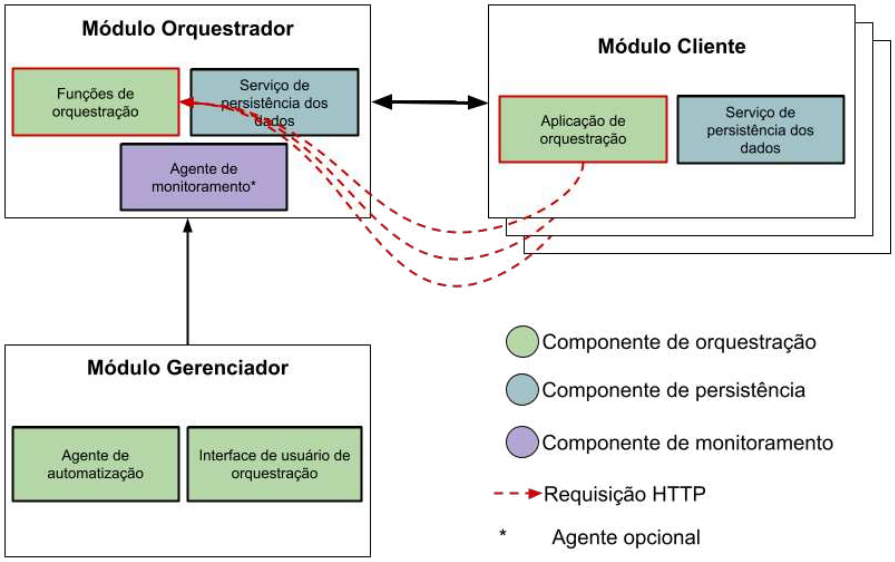


Fonte: Elaborado pelo autor (2023)

clientes disponível para treinamento e a fração de clientes que devem realizar o treinamento a cada rodada.

Os clientes, ao serem inicializados, se identificam para a função de orquestração, assim é possível saber todos os clientes disponíveis para treino. Essa identificação ocorre por meio de uma requisição que é feita pelo cliente para a o módulo orquestrador, como é ilustrado na Figura 24. Após receber as informações dos clientes, a função de orquestração as salva no *MongoDB*.

Figura 24 – Identificação de clientes



Fonte: Elaborado pelo autor (2023)

Quando a quantidade de clientes inscritos se iguala ao total definido pelo módulo gerenciador na inicialização, a primeira rodada do treinamento federado é iniciada. Com isso, o módulo orquestrador escolhe os clientes que deverão realizar o treinamento local na primeira rodada, essa escolha é feita de forma aleatória para simular um cenário real, em que nem sempre todos os clientes estão disponíveis para treinamento naquele determinado momento.

Na AAFFS, a forma com a qual os clientes tomam conhecimento que deverão treinar em uma determinada rodada pode ser implementada de diversas formas, por exemplo, pode-se implementar um sistema de mensageria, onde o cliente se inscreve para receber informações do orquestrador. Outra forma disso ocorrer é a função como serviço enviar uma requisição para um *endpoint* específico, informando que aquele cliente deve realizar o treinamento naquela rodada.

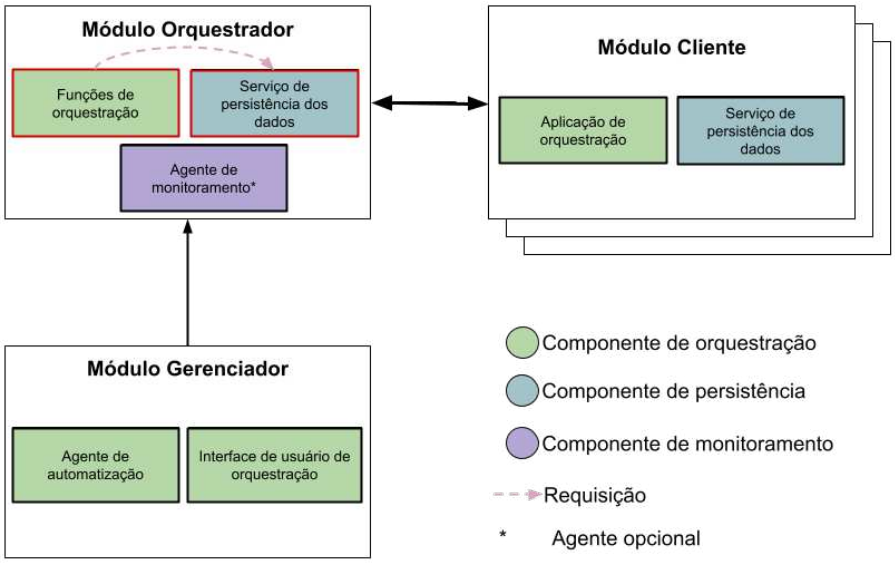
Na aplicação criada para testar a arquitetura, foi utilizada uma terceira alternativa na qual os clientes enviam uma requisição para o módulo orquestrador e recebem como resposta se

o treinamento local deve ser realizado. Essa requisição é feita periodicamente e foi desenvolvida utilizando *cron jobs*, que é um recurso de agendamento de tarefas baseado em tempo encontrado em sistemas operacionais do tipo *Unix*. Eles permitem que os usuários agendem tarefas ou comandos recorrentes para serem executados automaticamente em intervalos específicos, como diariamente, semanalmente ou mensalmente.

Após a seleção, os clientes enviam uma outra requisição HTTP para a função de orquestração para obter o modelo global atualizado e então realizam o treinamento local, atualizando o modelo recebido. Ao final do treinamento, o modelo local atualizado é enviado para o módulo orquestrador.

A cada recebimento de modelo local atualizado, a função orquestradora salvas os pesos dos modelos no banco de dados, como ilustrado na Figura 25, pois ao final de cada execução os dados salvos em memória na arquitetura FaaS são perdidos. A informação é salva no banco de dados usando o protocolo adequado ao sistema de armazenamento utilizado, no caso do *MongoDB*, o protocolo usado é o *MongoDB Wire Protocol* (MONGODB INC., 2023b).

Figura 25 – Persistência dos modelos locais atualizados



Fonte: Elaborado pelo autor (2023)

Quando o último cliente envia para a função como serviço o modelo local atualizado, o módulo orquestrador inicia a agregação de modelos. Primeiro, é recuperado do banco de dados todos os pesos dos modelos locais daquela rodada, em seguida o algoritmo *FedAvg* (MCMAHAN *et al.*, 2016) é executado, resultando em um modelo global atualizado.

Após finalizado o *FedAvg*, o modelo global é salvo banco de dados. uma nova rodada

é iniciada e todo processo descrito acima é repetido até que se atinja os objetivos previamente definidos, que podem ser: um número fixo de rodadas ou uma determinada taxa de acurácia do modelo.

Para os três cenários de testes, o procedimento relatado acima foi seguido, a comunicação entre os componentes foi realizada utilizando requisições HTTP. A principal diferença entre os cenários testados é a ausência de uma ferramenta de observabilidade quando testado a arquitetura utilizando uma *Raspberry Pi*.

6 AVALIAÇÃO

Nesta seção, são apresentados e discutidos os resultados obtidos do uso da AAFFS nos três cenários de testes apresentados. São comparados os dados dos três cenários entre si e com as informações presentes na literatura. São analisado, também, os resultados quando aplica-se o treinamento federado na borda da rede.

Para cada cenário de teste, foram seguidas as boas práticas descritas nas referências, e cada experimento foi repetido cinco vezes (PAPADOPOULOS *et al.*, 2021).

6.1 Métricas do aprendizado de máquina

Para medir a performance do aprendizado de máquina da arquitetura, foram utilizados dois conjuntos de dados distintos e duas métricas aplicadas a cada um dos conjuntos. Para todos os cenários de testes descritos anteriormente, foram obtidas métricas similares, o que indica que não há diferença entre utilizar diferentes componentes na arquitetura.

As métricas utilizadas foram a acurácia que refere-se à capacidade de um modelo prever corretamente o resultado ou o rótulo de um determinado conjunto de dados. É uma medida do desempenho do modelo e geralmente é expressa como uma porcentagem. A segunda métrica foi a função de perda *Negative Log Likelihood Loss* (NLLLoss).

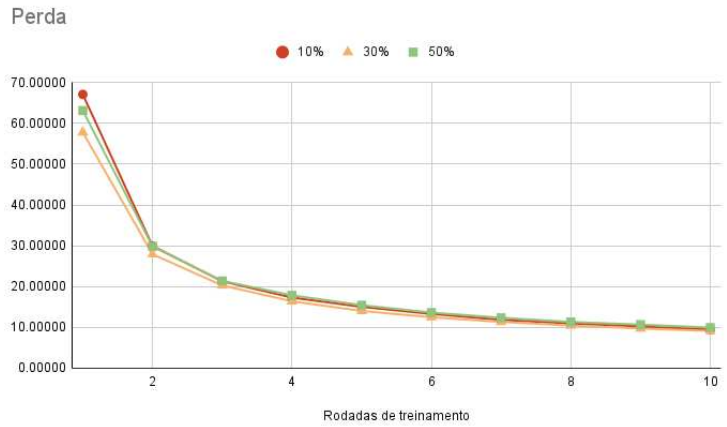
O primeiro conjunto de dados utilizado foi o *MNIST*, que consiste grande conjunto de imagens de dígitos manuscritos, cada uma das quais é uma imagem em escala de cinza. O outro conjunto foi o *CIFAR10*, que também são imagens em dez categorias distintas, porém, nesse último conjunto de dados as imagens são coloridas.

Foi realizado uma comparação entre o aprendizado tradicional e o aprendizado federado com três frações de clientes distintas, ou seja, três quantidades de clientes distintas que deverão realizar o treinamento a cada rodada.

6.1.1 Conjunto de dados: MNIST

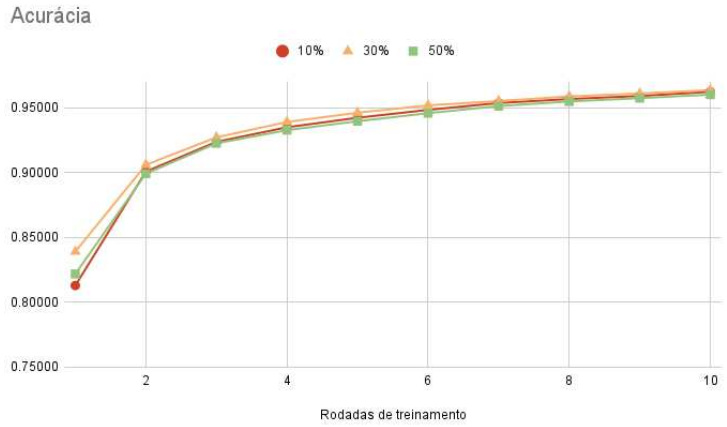
Primeiro, como é observado nas Figuras 26 e 27, são apresentados os comparativos da perda e da acurácia, respectivamente, por época utilizando diferentes frações clientes para o conjunto de dados *MNIST*. É percebido que nas primeiras épocas temos uma diferença considerável entre as frações de clientes distintas, porém conforme mais épocas são utilizadas, os valores de perda e acurácia se tornam cada vez mais similares.

Figura 26 – Perda por época por fração de clientes para o *MNIST*



Fonte: Elaborado pelo autor (2023)

Figura 27 – Acurácia por época por fração de clientes para o *MNIST*

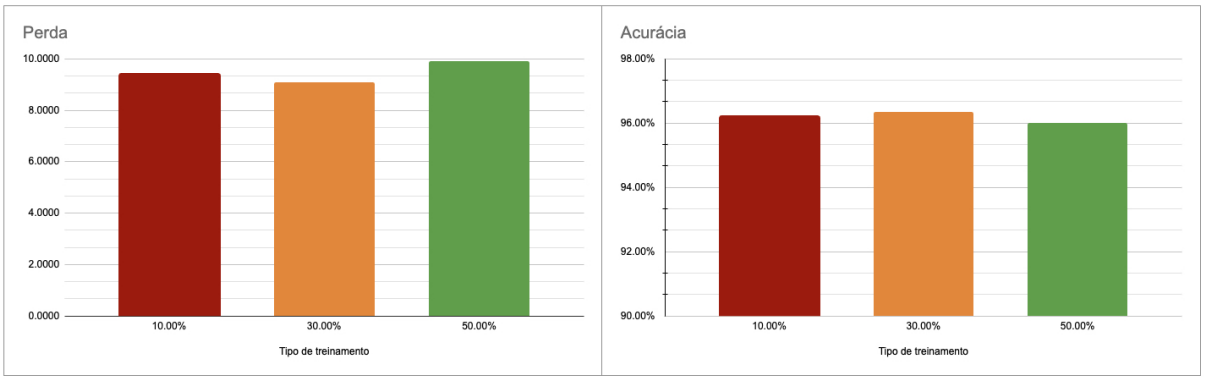


Fonte: Elaborado pelo autor (2023)

Apesar de similares, eles possuem uma diferença considerável, como pode ser visto na Figura 28. Para dez rodadas de treinamento, pode ser percebido que quando escolhido trinta por cento dos clientes, obtém-se o melhor resultado. Esse resultado não pode ser generalizado para todos os cenários, ele se refere ao modelo utilizado e ao conjunto de dados, mas podemos perceber que nem sempre aumentar o número de clientes irá resultar em um modelo melhor.

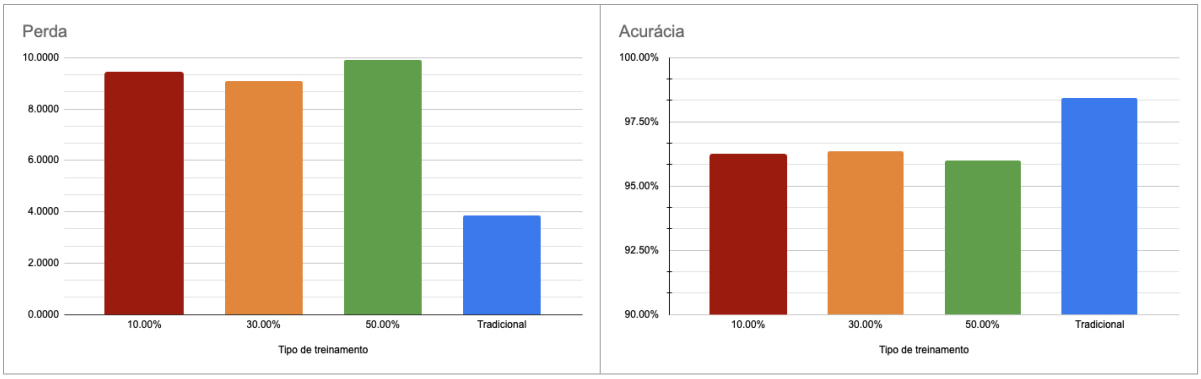
Quando foram comparado os resultados do aprendizado federado com o aprendizado tradicional, foi percebido um nível de performance menor no FL, porém, essa redução não inviabiliza o uso dessa nova abordagem, pois a diferença é pequena. O aprendizado federado apresenta uma acurácia de 96,35% enquanto na forma tradicional a acurácia foi de 98,43%. A Figura 29 apresenta a perda e a acurácia para o aprendizado federado e o tradicional, utilizando o conjunto de dados *MNIST*.

Figura 28 – Acurácia por época por fração de clientes para o *MNIST*



Fonte: Elaborado pelo autor (2023)

Figura 29 – Acurácia do aprendizado federado e tradicional para o *MNIST*



Fonte: Elaborado pelo autor (2023)

6.1.2 Conjunto de dados: CIFAR10

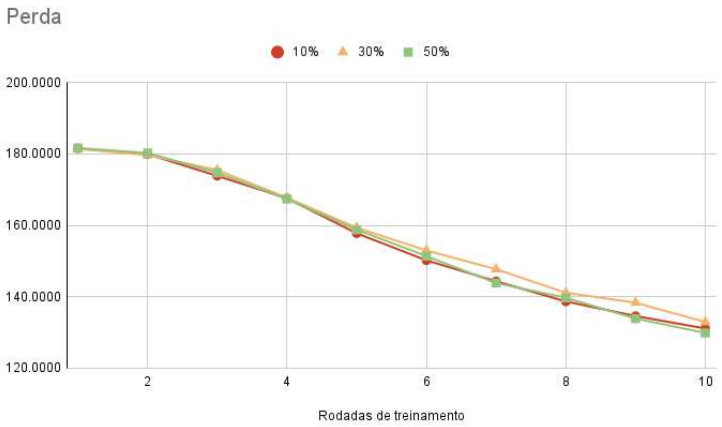
Para o *CIFAR10*, a perda e a acurácia para diferentes frações de clientes apresenta um comportamento diferente do apresentado pelo *MNIST*. Como pode ser visto nas Figuras 30 e 31, há diferenças no comportamento do aprendizado conforme é alterado a fração de clientes utilizados para o treinamento.

O comportamento da perda e da acurácia por fração de clientes são similares na primeira rodada e se diferenciam conforme as rodadas aumentam. Quando a fração de clientes realizado o treinamento a cada rodada é aumentada de dez por cento para trinta por cento, é observado uma diminuição da acurácia final do modelo, porém quando é utilizada uma fração cinquenta por cento do clientes, é obtida a melhor acurácia final dos três cenários testados, como pode ser visto na Figura 32.

Com isso, é percebido que a alteração no número de clientes realizando o treinamento por rodada impactam o resultado final do aprendizado e o tamanho do impacto vai depender de

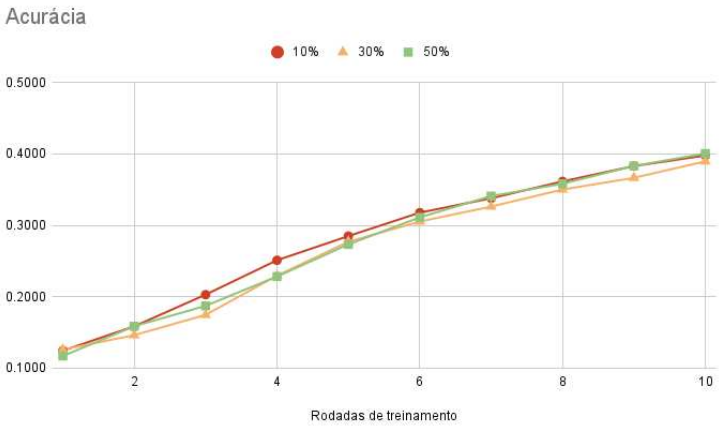
um conjunto de fatores, como conjunto de dados e modelo utilizado.

Figura 30 – Perda por época por fração de clientes para o *CIFAR10*



Fonte: Elaborado pelo autor (2023)

Figura 31 – Acurácia por época por fração de clientes para o *CIFAR10*



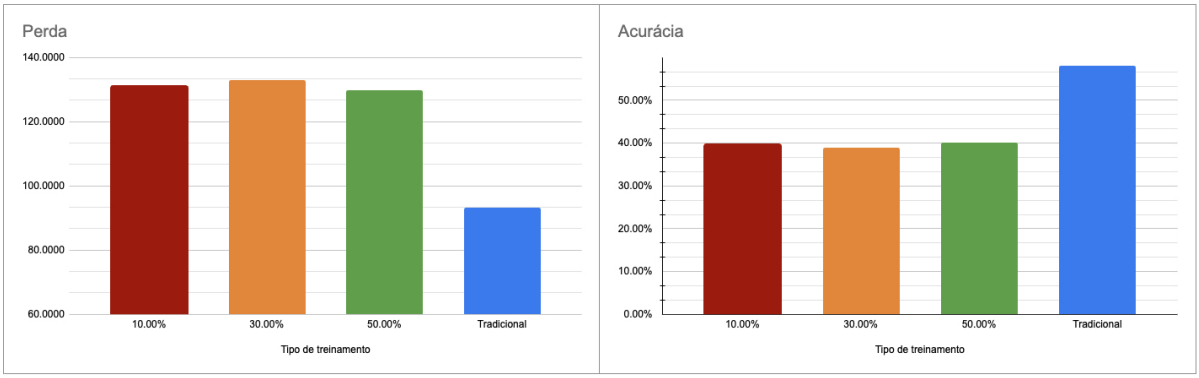
Fonte: Elaborado pelo autor (2023)

Quando são comparados os resultados obtidos com o aprendizado tradicional, é percebido que para o *CIFAR10* a diferença foi significativa. No melhor cenário do FL, foi obtido uma acurácia de 40,07%, enquanto no aprendizado tradicional foi apresentado uma acurácia de 58,16%. Isso não inviabiliza o aprendizado federado, isso é um indicativo que o modelo utilizado para os testes não é adequado para o conjunto de dados utilizado, pois a literatura apresenta registros de aprendizado federado que obtém uma acurácia de 90,08% (SHAMSIAN *et al.*, 2021).

Apesar de uma diferença significativa, foi demonstrado que é possível executar o aprendizado federado utilizando FaaS e a diferença de acurácia entre os paradigmas utilizados é devido ao modelo escolhido para a realização dos testes. Para cenários como esses é necessário

fazer uma análise exploratória detalhada para entender o comportamento dos dados e assim poder decidir qual modelo se encaixa melhor para o conjunto em análise.

Figura 32 – Acurácia por época por fração de clientes para o *CIFAR10*



Fonte: Elaborado pelo autor (2023)

6.2 Latência

Para comparar os resultados obtidos quando são utilizados recursos da nuvem e da computação na borda da rede, foi utilizada a latência para entender o comportamento da AAFFS nesses dois cenários. A latência refere-se ao atraso ou à quantidade de tempo que os dados levam para viajar de sua origem até o destino em um computador ou sistema de comunicação. É essencialmente o intervalo de tempo entre o início de uma ação e a resposta ou o resultado dessa ação. Essa métrica foi obtida ao realizar uma média de todas as requisições executadas durante o período de testes.

O objetivo de colocar o processamento na borda da rede é obter tempos menores de latência quando comparados com os serviços utilizados na nuvem. Com isso, aplicações que necessitam de uma resposta rápida para o funcionamento correto se beneficiam quando temos o processamento na borda da rede.

A AAFFS é capaz de operar tanto na nuvem quanto na borda e para o paradigma FaaS, as métricas de latências foram divididas em duas categorias. Essa divisão ocorre, pois os tempos de latência têm comportamentos diferentes de acordo com o estado em que a função se encontra no momento em que o evento de gatilho ocorre. Uma função pode está no estado *cold* ou *warm*, o primeiro, refere-se ao processo de inicialização de um serviço ou função quando ele é chamado pela primeira vez ou após um período de inatividade. No AWS *Lambda* não é possível controlar a duração desse período sem atividade antes que suas instâncias sejam removidas,

enquanto no *Apache OpenWhisk* é possível fazer isso através da mudança de alguns parâmetros nos arquivos de configuração.

O estado *warm*, ou *warm start*, refere-se ao cenário em que um serviço ou função é chamado e a infraestrutura já tem os recursos necessários alocados e prontos para a execução. Como nesse estado, existe pelo menos uma instância da função em execução, ao receber um evento, o processamento começa imediatamente e por isso, percebe-se uma latência menor quando comparadas com o estado *cold*.

Os tempos de latência entre os cenários executando a função de orquestração utilizando o AWS *Lambda* e o *Apache OpenWhisk* são similares e por isso esses dois cenários foram tratados como execução na nuvem e foram comparados com a execução utilizando o *Raspberry Pi*.

6.2.1 Warm start

A Figura 33 apresenta os resultados obtidos durante os testes. A nuvem demonstrou apresentar resultados consideravelmente superiores quando comparados com o *Apache OpenWhisk* executado na borda da rede. Para a nuvem foi obtida uma média de 0,821 segundos por requisição, enquanto na borda apresentou uma média de 3,981 segundos, cerca de 3,7 vezes mais demorado.

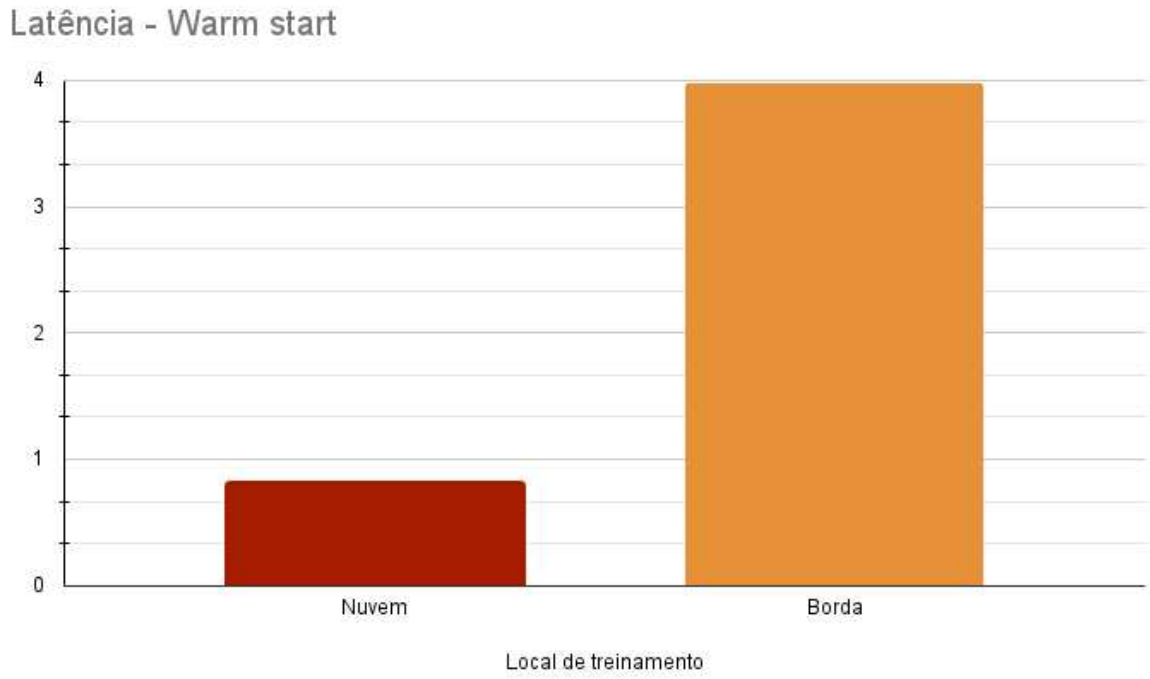
Esse comportamento se deve, principalmente, pelo fato de o *Apache OpenWhisk* ser uma ferramenta que consome bastante recursos e isso sobrecarregou a *Raspberry Pi*. A utilização de um software que consuma menos recursos para o uso na borda da rede irá resultar em um tempo menor de latência.

6.2.2 Cold start

No cenário de *cold start*, que foi ilustrado na Figura 34, a diferença foi ainda maior entre os dois cenários. A nuvem apresentou uma latência média de 6,52 segundos, enquanto a borda da rede apresentou uma média de 23,924 segundos para a mesma métrica.

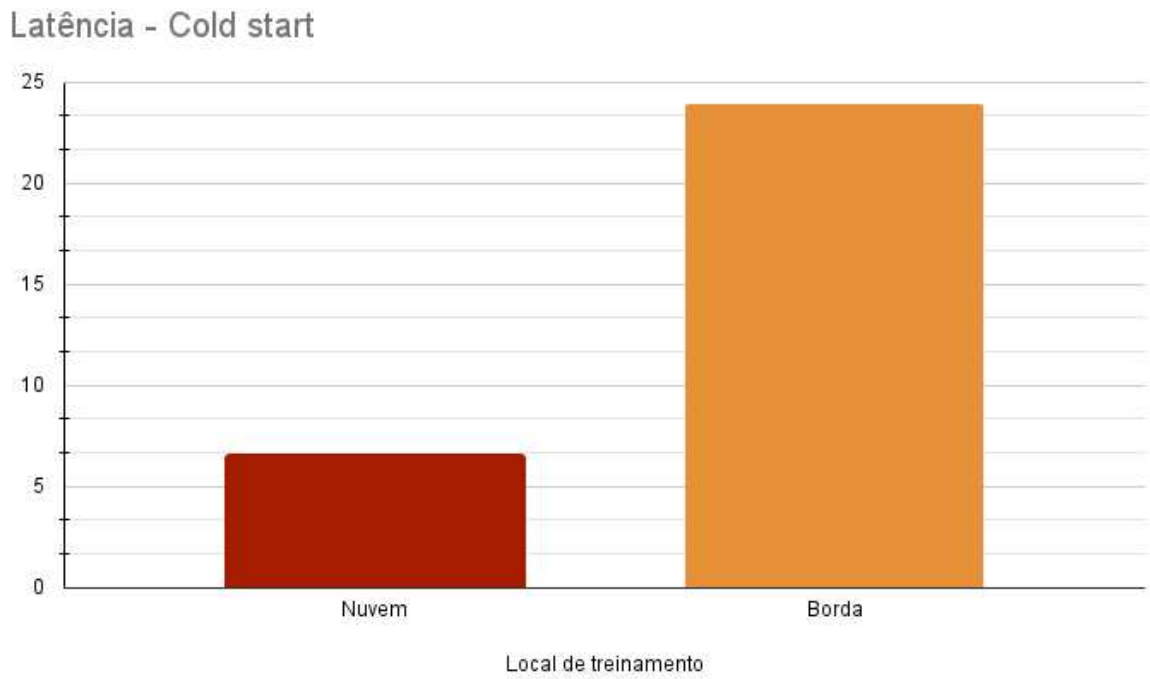
Essa diferença de aproximadamente 4,85 vezes mais demorado na borda decorre do fato da ausência de recursos na *Raspberry Pi*. O processo de inicialização de uma função começa com a verificação de que não existem contêineres em execução para aquela função específica e então é iniciado a criação de um conteiner *Docker*. Essa etapa de criação do contêiner tomou um tempo considerável durante o período de testes.

Figura 33 – Latência para o *warm start*



Fonte: Elaborado pelo autor (2023)

Figura 34 – Latência para o *cold start*



Fonte: Elaborado pelo autor (2023)

A Figura 35 apresenta os registros do *Apache OpenWhisk* sendo executado em uma *Raspberry Pi*. Analisando esses registros é possível visualizar como a aplicação está se comportando e identificar possíveis erros. Na figura, a primeira linha do registro é o recebimento da requisição. Após isso, como a função está no estado *cold start*, a aplicação executa tarefas necessárias para instanciar a função e isso pode ser visualizado olhando os registros na Figura 35. Após instanciado, os parâmetros são enviados para a função, isso é mostrado na última linha da figura, e sua execução é iniciada.

É possível perceber que do momento em que a requisição HTTP é recebida até o momento em que o processamento pela função inicia de fato, temos uma duração de 19 segundos. Isso é um indicativo que a utilização de um hardware mais potente irá resultar numa redução significante da latência na borda da rede.

Figura 35 – Registros do *OpenWhisk* em um cenário de *cold start*

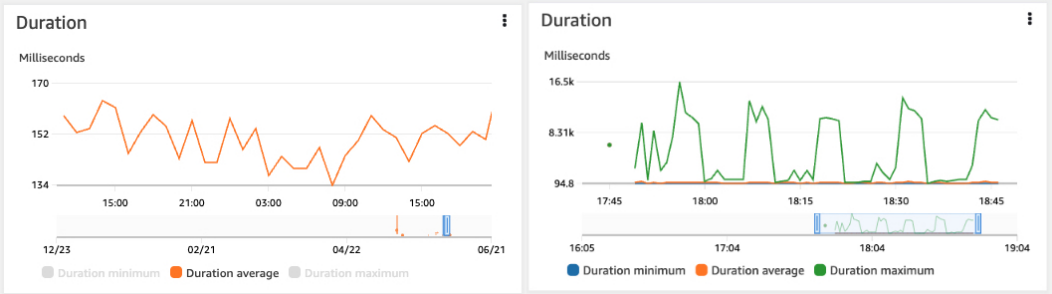
Fonte: Elaborado pelo autor (2023)

6.3 Uso das funções como serviço

Nos três cenários de teste o uso dos recursos no módulo de orquestração foram similares, ocorrendo uma variação apenas em decorrência da variação da fração de clientes realizando o treinamento em cada rodada. Segundo os dados obtidos pelo serviço *CloudWatch* da AWS, o tempo médio de execução de uma função foi de 150 milissegundos. A Figura 36 ilustra a duração máxima, mínima e média das funções de orquestração. Os picos apresentados na figura é devido ao *cold start*.

É importante ressaltar que o tempo médio de execução é baixo e com isso temos um

Figura 36 – Duração da execução das funções de orquestração



Fonte: Elaborado pelo autor (2023)

custo menor quando comparado com a arquitetura tradicional, pois a cobrança é feita com base no tempo de execução da função nos provedores desse serviço.

Com relação a escalabilidade, percebe-se uma diferença entre a fração de clientes utilizada. Na arquitetura tradicional, conforme essa fração aumenta, seria necessário uma máquina com maiores recursos para suprir a necessidade de mais clientes enviando requisições para o servidor. No caso do paradgima FaaS, a escalabilidade é realizada de com a execução de uma nova instancia da função e isso é feito de forma automática. A Figura 37 apresenta o número de execução concorrentes das funções para cada fração de clientes utilizadas nos testes e é possível perceber que quanto maior essa fração, maior o número de execuções concorrentes.

Figura 37 – Execução concorrente para diferentes frações de clientes

Fonte: Elaborado pelo autor (2023)

7 CONSIDERAÇÕES FINAIS

Em conclusão, este trabalho explorou o conceito de aprendizagem federada no contexto da computação sem servidor, mais especificamente, da função como serviço, testando cenários utilizando processamento da nuvem ou da computação na borda da rede. Por meio desta pesquisa e experimentação, foi possível evidenciar o potencial dessa nova abordagem e suas implicações em três cenários testados.

Foi proposta uma arquitetura capaz de funcionar em diferentes ambientes, seja em nuvem ou em um dispositivo inserido na borda da rede. A AAFFS também é capaz de funcionar com dispositivos heterogêneos, característica comum no paradigma de aprendizado federado.

Ao aproveitar os recursos da computação sem servidor, foi demonstrado a viabilidade e os benefícios da distribuição de modelos de aprendizado de máquina entre a nuvem e os dispositivos de borda. Essa abordagem híbrida não apenas reduz a carga sobre a infraestrutura de nuvem centralizada, mas também aumenta a privacidade, reduz a latência e permite a inferência em tempo real na borda.

A integração da aprendizagem federada com a computação sem servidor apresenta oportunidades interessantes para uma ampla gama de aplicativos. Setores como saúde, finanças e IoT podem aproveitar essa abordagem para permitir tarefas de aprendizado de máquina com um nível satisfatório de privacidade e baixa latência na borda. Além disso, a escalabilidade e a flexibilidade das arquiteturas sem servidor as tornam ideais para lidar com cargas de trabalho variáveis e acomodar a disponibilidade dinâmica de dispositivos de borda.

No entanto, ainda há desafios em termos de garantir a segurança, gerenciar a heterogeneidade nos dispositivos de borda e otimizar a sobrecarga de comunicação. Pesquisas futuras devem se concentrar em enfrentar esses desafios e explorar ainda mais o potencial da aprendizagem federada em ambientes de computação sem servidor.

7.1 Limitações da pesquisa

Este trabalho apresentou algumas limitações. Uma delas é referente a implantação de *software* orquestradores de funções como serviço na borda da rede. Atualmente esses *softwares* são voltados, principalmente, para a execução na nuvem com recursos abundantes e normalmente utilizando máquinas com processadores da família *x86*.

Essa limitação gerou uma dificuldade para executar o *Apache OpenWhisk* em um

Raspberry Pi, assim, foi necessário compilar diversos componentes do software para processadores da família *x86*, o que tomou um tempo considerável.

7.2 Trabalhos futuros

Para trabalhos futuros, é interessante implementar a arquitetura na borda da rede utilizando um *software* mais leve e que consuma menos recursos e com isso, estudar se há uma redução significativa da latência e que possa justificar deixar o processamento mais próximo do usuário. Além disso, é interessante testar outras ferramentas na borda da rede. Outro trabalho futuro interessante é o estudo de formas de adicionar técnicas de segurança que possam mitigar ataques, por exemplo, criptografia homomórfica e privacidade diferencial.

É importante também analisar o comportamento dos custos na arquitetura sem servidor e compará-la com a arquitetura tradicional.

REFERÊNCIAS

AGGARWAL, C. C. **Neural Networks and Deep Learning**: A textbook. Cham: Springer, 2018. 497 p. ISBN 978-3-319-94462-3.

AMAZON WEB SERVICES INC. **Introduction to MongoDB**. 2023. Disponível em: <https://aws.amazon.com/pt/cloudwatch/>. Acesso em: 22 mai. 2023.

APACHE OPENWHISK AUTHORS. **ntroduction to Grafana**. 2023. Disponível em: <https://github.com/apache/openwhisk/tree/master/core/monitoring/user-events>. Acesso em: 23 mai. 2023.

CHE, C.; LI, X.; CHEN, C.; HE, X.; ZHENG, Z. A decentralized federated learning framework via committee mechanism with convergence guarantee. **IEEE Transactions on Parallel and Distributed Systems**, Institute of Electrical and Electronics Engineers (IEEE), v. 33, n. 12, p. 4783–4800, dec 2022.

DEISENROTH, M. P.; FAISAL, A. A.; ONG, C. S. **Mathematics for Machine Learning**. [S.l.]: Cambridge University Press, 2020.

ERICSSON. Ericsson mobility report - business review edition. 2023. Disponível em: <https://www.ericsson.com/4901bf/assets/local/reports-papers/mobility-report/documents/2023br/emr-monetization-report.pdf>.

GRAFBERGER, A.; CHADHA, M.; JINDAL, A.; GU, J.; GERNDT, M. Fedless: Secure and scalable federated learning using serverless computing. In: **2021 IEEE International Conference on Big Data (Big Data)**. Los Alamitos, CA, USA: IEEE Computer Society, 2021. p. 164–173. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/BigData52589.2021.9672067>.

HASSAN, H.; BARAKAT, S.; SARHAN, Q. Survey on serverless computing. **Journal of Cloud Computing**, v. 10, 07 2021.

HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. **The Elements of Statistical Learning**. New York, NY, USA: Springer New York Inc., 2001. (Springer Series in Statistics).

HAYKIN, S. S. **Neural networks and learning machines**. Third. Upper Saddle River, NJ: Pearson Education, 2009.

JAMBUNATHAN, B.; YOGANATHAN, K. Architecture decision on using microservices or serverless functions with containers. In: **2018 International Conference on Current Trends towards Converging Technologies (ICCTCT)**. [S.l.: s.n.], 2018. p. 1–7.

JAMES, G.; WITTEN, D.; HASTIE, T.; TIBSHIRANI, R. **An Introduction to Statistical Learning: with Applications in R**. [S.l.]: Springer, 2013.

JIANG, J.; GAN, S.; LIU, Y.; WANG, F.; ALONSO, G.; KLIMOVIC, A.; SINGLA, A.; WU, W.; ZHANG, C. Towards demystifying serverless machine learning training. In: **Proceedings of the 2021 International Conference on Management of Data**. ACM, 2021. Disponível em: <https://doi.org/10.1145/3448016.3459240>.

KUBERNETES AUTHORS. **Command line tool (kubectl)**. 2023. Disponível em: <https://kubernetes.io/docs/reference/kubectl/>. Acesso em: 25 mai. 2023.

KUBERNETES AUTHORS. **Kubernetes Components**. 2023. Disponível em: <https://kubernetes.io/docs/concepts/overview/components/>. Acesso em: 23 mai. 2023.

KUBERNETES AUTHORS. **O que é Kubernetes?** 2023. Disponível em: <https://kubernetes.io/pt-br/docs/concepts/overview/what-is-kubernetes/>. Acesso em: 22 mai. 2023.

MAISSEN, P.; FELBER, P.; KROPF, P.; SCHIAVONI, V. FaaSdom. In: **Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems**. ACM, 2020. Disponível em: <https://doi.org/10.1145%2F3401025.3401738>.

MARTINS, H.; ARAUJO, F.; CUNHA, P. R. Benchmarking serverless computing platforms. **Journal of Grid Computing**, v. 18, 12 2020.

MCMAHAN, H. B.; MOORE, E.; RAMAGE, D.; HAMPSON, S.; ARCAS, B. A. y. Communication-efficient learning of deep networks from decentralized data. arXiv, 2016. Disponível em: <https://arxiv.org/abs/1602.05629>.

MELIS, L.; SONG, C.; CRISTOFARO, E. D.; SHMATIKOV, V. **Exploiting Unintended Feature Leakage in Collaborative Learning**. arXiv, 2018. Disponível em: <https://arxiv.org/abs/1805.04049>.

MICROSOFT CORPORATE BLOGS. **Microsoft and OpenAI extend partnership**. 2023. Disponível em: <https://blogs.microsoft.com/blog/2023/01/23/microsoftandopenaiextendpartnership/>. Acesso em: 13 jun. 2023.

MONGODB INC. **Introduction to MongoDB**. 2023. Disponível em: <https://www.mongodb.com/docs/manual/introduction/>. Acesso em: 22 mai. 2023.

MONGODB INC. **MongoDB Wire Protocol**. 2023. Disponível em: <https://www.mongodb.com/docs/manual/reference/mongodb-wire-protocol/>. Acesso em: 25 mai. 2023.

MOTHUKURI, V.; PARIZI, R. M.; POURIYEH, S.; HUANG, Y.; DEHGHANTANHA, A.; SRIVASTAVA, G. A survey on security and privacy of federated learning. **Future Generation Computer Systems**, v. 115, p. 619–640, 2021. ISSN 0167-739X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167739X20329848>.

MULHOLLAND, C. S. Dados pessoais sensíveis e a tutela de direitos fundamentais: uma análise à luz da lei geral de proteção de dados (lei 13.709/18). **Revista de Direitos e Garantias Fundamentais**, v. 19, n. 3, p. 159–180, dez. 2018. Disponível em: <https://sisbib.emnuvens.com.br/direitosegarantias/article/view/1603>.

PAPADOPOULOS, A. V.; VERSLUIS, L.; BAUER, A.; HERBST, N.; KISTOWSKI, J. v.; ALI-ELDIN, A.; ABAD, C. L.; AMARAL, J. N.; TůMA, P.; IOSUP, A. Methodological principles for reproducible performance evaluation in cloud computing. **IEEE Transactions on Software Engineering**, v. 47, n. 8, p. 1528–1543, 2021.

PETROSYAN, D.; ASTSATRYAN, H. Serverless high-performance computing over cloud. **Cybernetics and Information Technologies**, v. 22, n. 3, p. 82–92, 2022. Disponível em: <https://doi.org/10.2478/cait-2022-0029>.

QAMMAR, A.; DING, J.; NING, H. Federated learning attack surface: Taxonomy, cyber defences, challenges, and future directions. **Artif. Intell. Rev.**, Kluwer Academic Publishers, USA, v. 55, n. 5, p. 3569–3606, jun 2022. ISSN 0269-2821. Disponível em: <https://doi.org/10.1007/s10462-021-10098-w>.

RAMOS, H.; MAIA, G.; PAPA, G.; ALVIM, M.; LOUREIRO, A.; CARDOSO-PEREIRA, I.; CAMPOS, D.; FILIPAKIS, G.; RIQUETTI, G.; CHAGAS, E.; BARROS, P.; GOMES, G.; ALLENDE-CID, H. Aprendizado Federado aplicado à Internet das Coisas. In: REZENDE, J.; CARDOSO, K.; ROSA, P.; SILVA, F. (Ed.). **Minicursos do XXXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. 1. ed. SBC, 2021. p. 196–248. ISBN 9786587003849. Disponível em: <https://sol.sbc.org.br/livros/index.php/sbc/catalog/view/81/355/611-1>.

SHAMSIAN, A.; NAVON, A.; FETAYA, E.; CHECHIK, G. **Personalized Federated Learning using Hypernetworks**. 2021.

SHI, W.; PALLIS, G.; XU, Z. Edge computing [scanning the issue]. **Proceedings of the IEEE**, v. 107, n. 8, p. 1474–1481, 2019.

WANG, H.; NIU, D.; LI, B. Distributed machine learning with a serverless architecture. In: **IEEE INFOCOM 2019 - IEEE Conference on Computer Communications**. [S.l.: s.n.], 2019. p. 1288–1296.

YANG, Q.; LIU, Y.; CHEN, T.; TONG, Y. Federated machine learning: Concept and applications. arXiv, 2019. Disponível em: <https://arxiv.org/abs/1902.04885>.

ZHOU, N.; ZHOU, H.; HOPPE, D. Containerisation for high performance computing systems: Survey and prospects. **IEEE Transactions on Software Engineering**, p. 1–20, 2022.