UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

MATHEUS WOEFFEL CAMARGO

Cloud Computing para deploy de modelos de Deep Learning para a detecção de Retinopatia Diabética

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. Philippe Olivier Alexandre Navaux

Porto Alegre
2023

Dedico esse trabalho ao meu tio
Newerton Camargo (in memoriam),
cujas risadas e conselhos carregarei para sempre.

AGRADECIMENTOS

Agradeço primeiramente ao Professor Philippe Navaux, pela oportunidade de me orientar e pelas incontáveis contribuições para a conclusão deste trabalho. Agradeço também ao Cristiano Kunas, que da mesma forma, teve importantes contribuições para a realização deste trabalho.

Aos membros da banca, Professor Luciano Gaspary e Professora Mariana Mendoza, pela disponibilidade de avaliar este trabalho e pelos ensinamentos que obtive nas respectivas disciplinas durante a graduação, que carrego com grande estima.

Aos meus pais, Nilton Camargo e Ingrid Woeffel, por todo o apoio não somente durante a graduação, mas em toda minha vida. Também à minha irmã, Camila Woeffel, pelas conversas e apoio durante toda a minha graduação.

À minha namorada e futura esposa, Clara Figueiredo, pelo exemplo de dedicação e todo o seu amor e apoio, que foram essenciais para estes e outros momentos fundamentais da minha graduação.

Ao meu amigo e professor, Diego Braga, pelos conselhos e a grande ajuda na revisão da monografia.

Aos amigos que fiz durante a graduação, Filipe Dias, Raphael Scherpinski, Gabriel Engel, Rodrigo Bastos e Alencar da Costa, por todos os momentos que partilhamos e todo o apoio de vocês. Em especial, ao meu grande amigo Alencar da Costa, por partilhar esse momento difícil comigo.

Finalmente, gostaria de agradecer a todos os professores e colaboradores da UFRGS, que direta ou indiretamente me ajudaram durante toda a minha jornada acadêmica.

RESUMO

A incidência de diabetes está aumentando a um ritmo alarmante em todo o mundo. Por consequência, crescem os casos de retinopatia diabética (RD), uma complicação da diabetes que em sua forma mais grave pode levar à cegueira. A falta de mão de obra especializada para diagnóstico, essencial para o tratamento bem-sucedido da doença, traz a necessidade de estudo de alternativas para o diagnóstico via meios computacionais. Pesquisas recentes sobre o uso de *Deep Learning* para a detecção de RD se mostram como uma importante alternativa para melhorar a utilização da mão de obra especializada a partir da priorização de casos mais graves. Partindo deste contexto, o objetivo do presente trabalho é avaliar o desempenho e custo financeiro de alternativas baseadas em computação em nuvem para o *deploy* de modelos de *Deep Learning* para classificação de RD. Utilizando como base os serviços *real-time inference* e *serverless inference*, ambos da plataforma Amazon Sagemaker, foram consideradas otimizações e diferentes alternativas de configuração, obtendo até $24,34\%$ de redução de custo financeiro e até $2,16\times$ de aumento de desempenho. Por fim, foram utilizados conceitos como conteinerização e infraestrutura como código durante a implementação da solução, para permitir a reprodução das alternativas de *deploy* e dos experimentos realizados de maneira facilitada.

Palavras-chave: Computação em nuvem. Infraestrutura como código. Retinopatia Diabética. *Deep Learning*. Deploy. Aprendizado de máquina.

Cloud Computing for deploying Deep Learning models for Diabetic Retinopathy detection

ABSTRACT

The incidence of diabetes is increasing at an alarming rate worldwide. As a consequence, cases of diabetic retinopathy (DR), a diabetes complication that in its most severe form can cause blindness, are growing. The lack of specialized workforce for diagnosis, essential for a successful treatment, brings the need to study alternatives for diagnosis via computational means. Recent research on the use of Deep Learning for DR detection are seen as an important alternative to improve the use of specialized workforce by prioritizing more severe disease stages. Based on this context, the main goal of this work is to evaluate the performance and cost of cloud computing-based alternatives for the deployment of Deep Learning models for DR detection. Using Amazon Sagemaker's real-time inference and serverless inference services, optimizations and different configuration alternatives were considered, achieving up to a 24.34% cost reduction and up to a $2.16\times$ performance increase. Finally, concepts such as containerization and infrastructure as code were used during the implementation of the solution to facilitate the reproduction of deployment alternatives and experiments performed.

Keywords: Cloud computing, Infrastructure as Code, Diabetic Retinopathy, Deep Learning, Deploy, Machine Learning.

LISTA DE ABREVIATURAS E SIGLAS

ARM	Advanced RISC Machines
API	Application Programming Interface
APTOS	Asia Pacific Tele-Ophthalmology Society
ASGI	Asynchronous Server Gateway Interface
AWS	Amazon Web Services
CDN	Content Delivery Network
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSV	Comma-Separated Values
ECR	Elastic Container Registry
GCP	Google Cloud Platform
GPU	Graphics Processing Unit
HTTP	Hypertext Transfer Protocol
HPC	High-Performance Computing
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
ISA	Instruction Set Architecture
OCI	Open Container Initiative
OS	Operational System
PaaS	Platform as a Service
RD	Retinopatia Diabética
RISC	Reduced Instruction Set Computer
SaaS	Software as a Service
URL	Uniform Resource Locator

VM Virtual Machine

WSGI Web Server Gateway Interface

LISTA DE FIGURAS

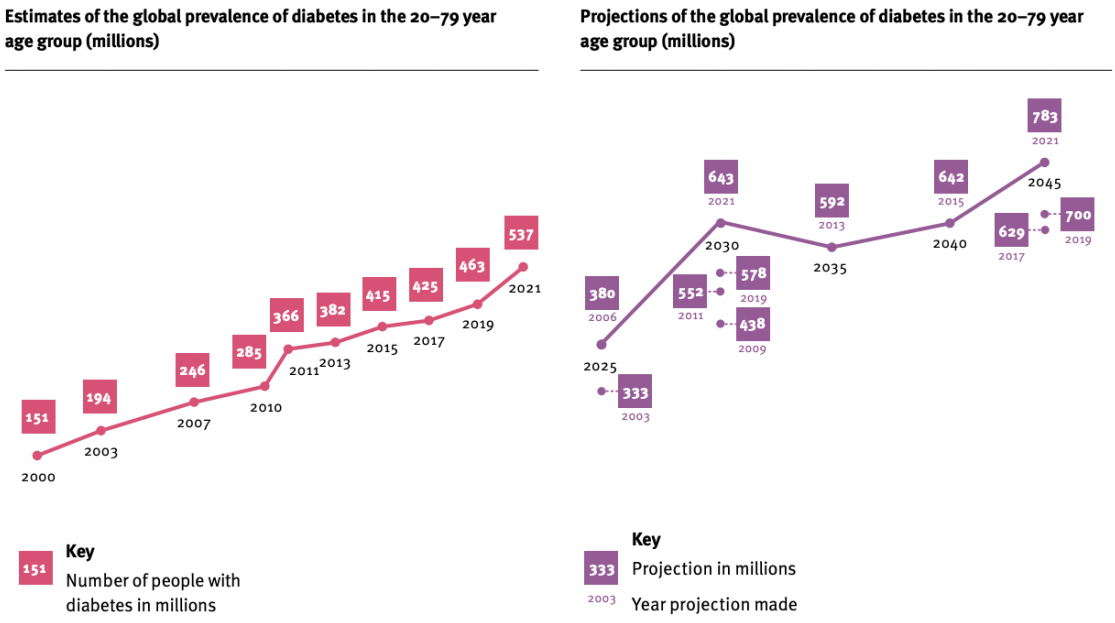
LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A diabetes vem crescendo no mundo em ritmos alarmantes. Em 2021, estima-se que cerca de 537 milhões de pessoas sofreram com essa doença mundialmente [figura 1.1]. Espera-se que 1/3 dessas pessoas venham a desenvolver retinopatia diabética (RD). A RD é uma complicação consequente da diabetes, em que o alto nível de açúcar no sangue causa danos à retina, prejudicando a visão e podendo em seu nível mais grave levar até mesmo à cegueira.

Figura 1.1: Estimativa e projeção da prevalência de diabetes no mundo. No gráfico da esquerda podemos notar a evolução da prevalência de diabetes, mais recentemente estimada em 537 milhões de pessoas mundialmente. Na direita, vemos um gráfico da projeção de diabetes, onde é projetado que cerca de 783 milhões de pessoas sofram com diabetes no ano de 2045.



Fonte: (FEDERATION, 2021)

Neste contexto, o diagnóstico da doença em seus estágios iniciais ainda é a melhor forma de prevenção, já que após o diagnóstico é possível evitar o agravamento do quadro a partir do controle dos níveis de açúcar do sangue e da pressão sanguínea. Para o diagnóstico, é importante a realização do exame de tomografia de coerência óptica (*OCT*), que obtém imagens da retina do paciente, posteriormente avaliadas por um oftalmologista capacitado.

Um exemplo da eficácia de políticas públicas para o diagnóstico da doença é a redução do número de casos de cegueira ocasionada pela RD no Reino Unido. Após a implantação de uma política de incentivo à realização de exames para diagnóstico de RD,

que previamente era a maior causa da cegueira em adultos no Reino Unido, a doença perdeu este posto, conforme mostrado em Liew, Michaelides and Bunce (2014).

Muito embora a realização de exames preventivos seja reconhecida mundialmente como a principal forma de combate à doença, essa prática ainda não é amplamente difundida, em consequência da escassez de profissionais especializados capazes de realizar a avaliação das imagens de fundo de olho. Na China, a razão entre pacientes com diabetes e oftalmologistas chega a 1:3000, conforme citado em Dai et al. (2021), tornando inviável a aplicação de políticas de realização de exames ao nível nacional.

Portanto, faz-se necessária a aplicação de métodos computacionais capazes de avaliar imagens de fundo de olho e detectar a presença de retinopatia diabética em seus diferentes graus de complexidade. Um método muito promissor é o uso de *Deep Learning*, na qual redes neurais são treinadas de maneira supervisionada para receber imagens de fundo de olho e realizar a classificação da imagem interpretando-a segundo diferentes estágios de retinopatia diabética. Tal tecnologia, se disponibilizada para profissionais da saúde, poderia ser utilizada para priorizar os casos mais graves, desta forma proporcionando uma melhor utilização da limitada mão de obra capaz de confirmar os diagnósticos.

Partindo desta conjuntura, o presente trabalho se propõe a avaliar diferentes alternativas para *deploy* de dois modelos de rede neural, treinados para a detecção de Retinopatia Diabética, em um ambiente de computação em nuvem. Serão avaliados tanto custo financeiro como desempenho de diferentes infraestruturas utilizadas para a fase de inferência dos modelos.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo apresentaremos a base teórica dos temas abordados no trabalho. Na seção 2.1 serão apresentados conceitos relativos a redes neurais e, em especial, à arquitetura CNN, utilizada pelos modelos avaliados neste trabalho. Na sequência, as seções 2.2 e 2.3 apresentam conceitos relativos à conteinerização e a ferramenta Docker, amplamente utilizada no trabalho e essencial para o entendimento de otimizações realizadas. A seguir, na seção 2.4 abordaremos conceitos fundamentais de computação em nuvem e, em especial, de *serverless computing*, um modelo de computação amplamente utilizado no trabalho. Finalmente, na seção 2.5 apresentaremos conceitos de infraestrutura como código e a relevância de seu uso para o presente trabalho.

2.1 Redes Neurais

As redes neurais são um subconjunto de algoritmos de *machine learning*, ou aprendizado de máquina. O aprendizado de máquina consiste no treinamento de sistemas para realização de uma tarefa específica sem a necessidade de serem explicitamente programados para tal. Dentro dessa classe de algoritmos, diversos tipos de aprendizado são possíveis:

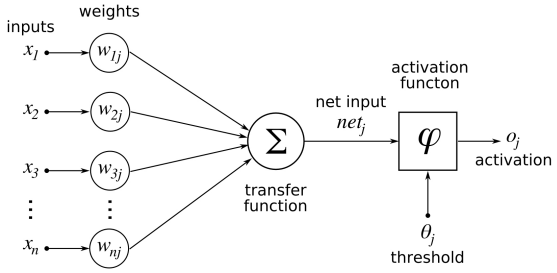
- Aprendizado supervisionado: No modelo de aprendizado supervisionado, são fornecidos dados rotulados, pares de entrada e saídas desejados, para que os algoritmos consigam buscar por uma função que aproxime a relação de entrada e saída contida nesse conjunto de dados fornecidos. Espera-se que essa busca encontre uma função capaz de classificar entradas corretamente, mesmo aquelas que não estejam no conjunto de dados original de treinamento.
- Aprendizado não supervisionado: No aprendizado supervisionado, o conjunto de dados fornecido ao algoritmo não possui nenhum tipo de rótulo ou de saída desejada, diferentemente do aprendizado supervisionado. Nesse tipo de aprendizado, os algoritmos visam encontrar padrões e agrupamentos nos dados, sem nenhum conhecimento prévio de quais são estes padrões e agrupamentos.
- Aprendizado por reforço: No aprendizado por reforço, um agente aprende a tomar decisões baseado em *feedbacks* parciais, que visam informar o quão boas foram as decisões tomadas pelo agente. Diferentemente do aprendizado supervisionado,

onde o *feedback* é fornecido de maneira direta e completa, no aprendizado por reforço o *feedback* é indireto e incompleto, apenas dando uma estimativa do quão boa foi a tomada de decisão do agente.

Dentre os paradigmas descritos anteriormente, as redes neurais se enquadram em algoritmos de aprendizado de máquina supervisionados. Esses modelos são inspirados na forma como redes neurais biológicas constituem os cérebros de animais. Sua unidade fundamental é o neurônio que, assim como as células neurais, recebem um conjunto de sinais de entrada e emitem um conjunto de sinais de saída.

Cada neurônio de uma rede neural recebe um conjunto de entradas, multiplica-as por um conjunto de pesos e em seguida somam-se esses valores, constituindo o que é chamado de *net input*. Posteriormente, soma-se esse resultado a um *bias* ou *threshold* que é um único valor escalar. Na sequência, aplica-se esta soma a uma função de ativação, que visa transformar a saída do neurônio em uma saída não linear. Por exemplo, um exemplo de função de ativação possível é a função sigmoíde, que visa transformar a saída do neurônio em um número entre 0 (neurônio não transmitindo impulso) e 1 (neurônio transmitindo o maior impulso possível). Tal processo está representado na figura 2.1.

Figura 2.1: Exemplo de funcionamento de um neurônio. Cada entrada é multiplicada por um peso específico. Na sequência, o somatório desses valores, o *net input*, é somado a um *bias* ou *threshold*, então sendo aplicado a uma função de ativação que transforma a saída do neurônio em um valor entre 0 e 1.



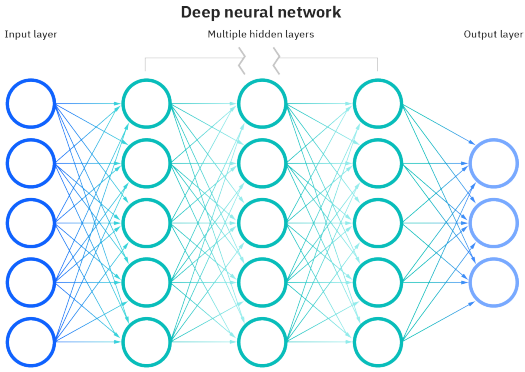
Fonte: (DICKSON, 2019)

Os neurônios são então organizados em camadas, de maneira sequencial. A camada 0 representa o input, que então é processado pela camada 1 e repassado para a próxima camada. As camadas de neurônios entre a entrada e saída são comumente chamadas de *hidden layers*. Essa arquitetura é melhor representada na figura 2.2.

Dentre os diversos tipos de redes neurais, uma das arquiteturas utilizadas pelo presente trabalho é a das redes convolucionais (CNN - *Convolutional Neural Networks*), conforme utilizado em Moreira et al. (2020) para a detecção de Retinopatia Diabética.

As CNNs são redes neurais que se inspiram no funcionamento da visão humana,

Figura 2.2: Arquitetura de uma rede neural. Podemos notar a relação sequencial entre as camadas, e a caracterização das camadas intermediárias entre a entrada e a saída como *hidden layers*.

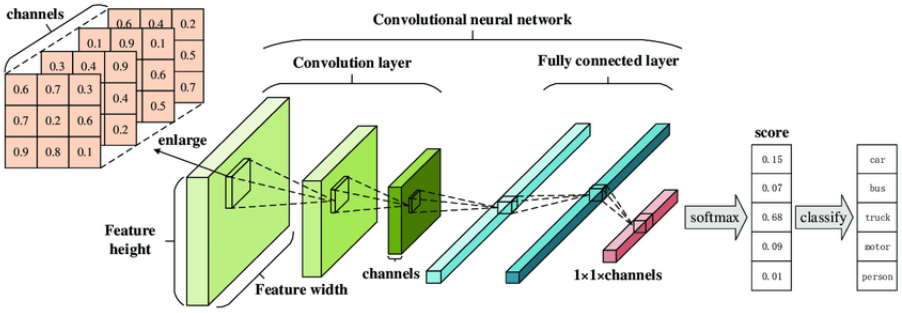


Fonte: (KAVLAKOGLU, 2020)

filtrando subáreas da visão sucessivamente por convoluções. Assim como sua inspiração, essas redes são utilizadas em aplicações voltadas a imagem, como: segmentação de imagens, classificação de imagens, análise de imagens médicas (diagnóstico), dentre outras aplicações.

Em uma CNN, os neurônios de uma camada convolucional e de sua camada posterior não estão totalmente conectados, sendo conectados seguindo um princípio de conectividade local. Como em imagens temos *pixels* dispostos espacialmente e estes possuem mais relação com *pixels* mais próximos espacialmente, não há a necessidade de conectar todos os *pixels* de uma imagem ou todas as saídas de uma camada anterior à próxima camada. Este conceito é melhor explicado na figura 2.3.

Figura 2.3: Exemplo de funcionamento de uma CNN. Nesse tipo de rede neural, existem camadas cuja saída de um dado neurônio depende apenas de um subconjunto dos neurônios da camada anterior, seguindo um princípio de conectividade local. Na imagem, podemos ver essa conectividade local representada pela projeção dos quadrados verdes (subconjunto dos neurônios de uma camada) sendo projetados em um neurônio da camada posterior.



Fonte: (KANG; SONG; SUN, 2019)

Existem diversos tipos de CNNs, cada uma com suas particularidades quanto à

arquitetura, como: número de camadas, número de neurônios por camadas ou como estão interconectadas.

2.2 Conteinerização

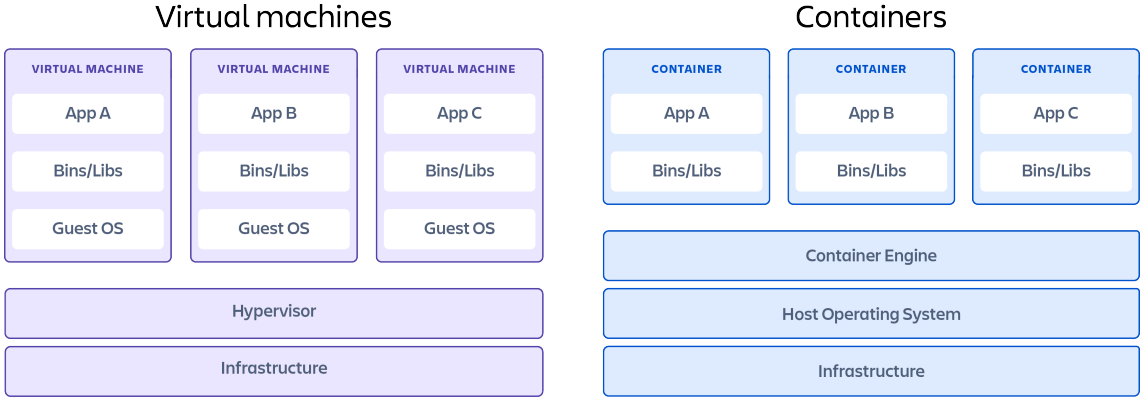
A conteinerização é o processo de empacotamento de software, de maneira que o código-fonte e todos os componentes necessários para a execução deste estejam contidos e isolados em um único objeto, o *container*.

A principal vantagem da conteinerização é a ampla facilidade de execução do código de maneira consistente e portável, isso porque as dependências do mesmo já estão contidas em um único objeto, tornando a execução independente de configurações implícitas de ambiente.

O empacotamento e isolamento de aplicações também pode ser alcançado através do uso de máquinas virtuais (VMs). No entanto, a principal diferença entre o uso de *containers* e máquinas virtuais é o custo computacional e portabilidade. Como as VMs virtualizam não somente o espaço de aplicação, mas também todo o sistema operacional, o custo em termos de desempenho e tamanho de VMs é significativamente maior que o de *containers*.

Conforme ilustrado na figura 2.4, as VMs empacotam todo o sistema operacional hospedeiro (*Guest OS*), e precisam de um *hypervisor*, que simula a infraestrutura de maneira transparente para os sistemas hospedeiros, emulando completamente recursos de hardware em baixo nível como CPU, disco e dispositivos de rede.

Figura 2.4: Diferença entre VMs e *containers*, diferentemente das VMs, os *containers* não virtualizam todo o sistema operacional, e incluem apenas as bibliotecas necessárias para a aplicação, tornando-os mais leves e portáteis.



Fonte: (BUCHANAN, 2023)

A popularidade crescente da conteinerização pode ser explicada pelos inúmeros benefícios advindos desta, os quais são sumarizados na lista a seguir.

- Otimização de recursos utilizados: É possível rodar vários *containers* "leves" em uma mesma máquina, possibilitando um melhor uso de recursos comparado ao uso de VMs.
- Automação: A padronização no empacotamento de software possibilita a automação de estágios do desenvolvimento de software, como: *Build*, *Deploy* e o escalonamento das aplicações.
- Desempenho: *Containers* podem ser inicializados, deletados ou escalados de maneira mais rápida que VMs.
- Produtividade: Como o software que roda em produção é empacotado em um objeto autocontido, isso possibilita que desenvolvedores façam o *debug* das aplicações de maneira mais assertiva, já que o uso de *containers* diminui a diferença entre os ambientes de desenvolvimento e produção.

2.3 Docker

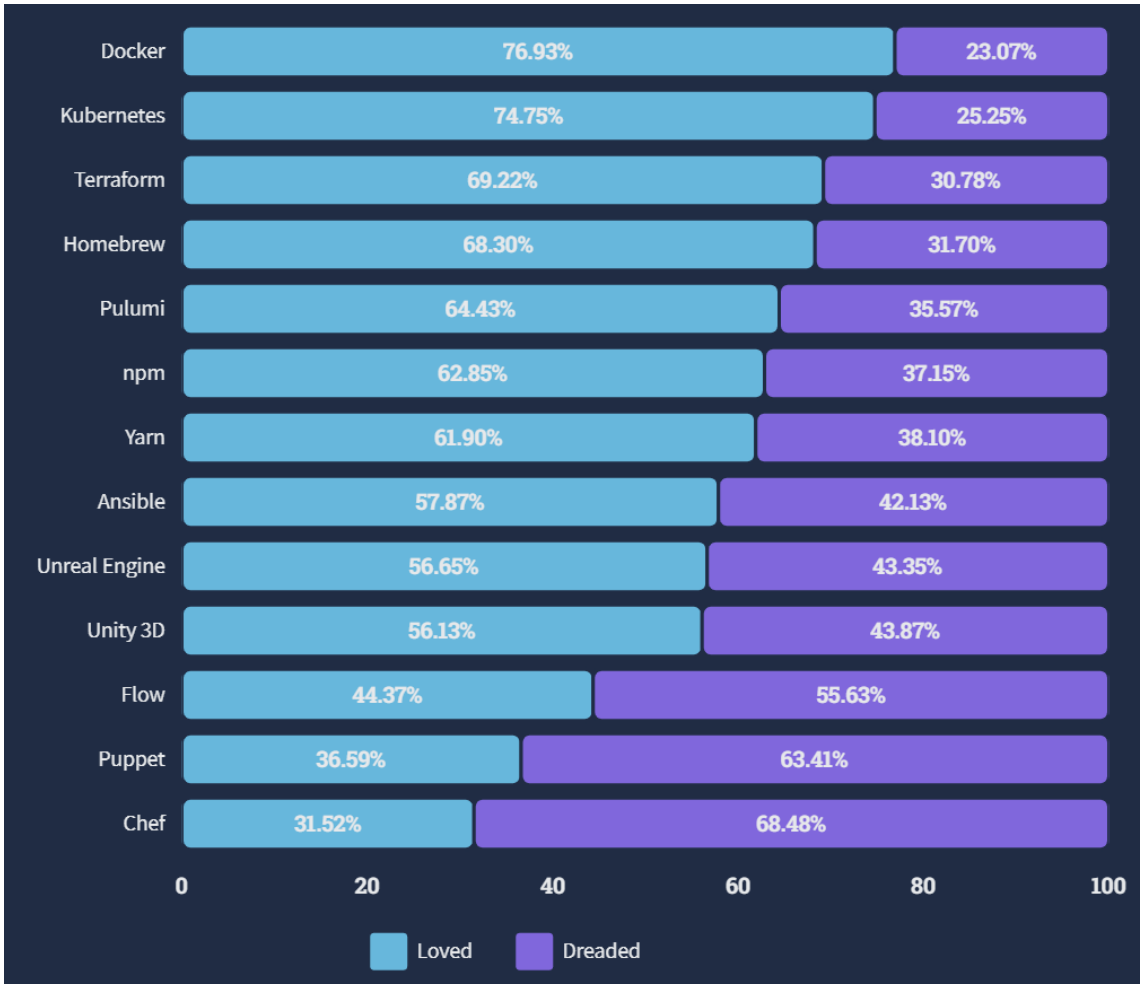
Dentre as tecnologias disponíveis para a conteinerização, o Docker é a tecnologia mais popular, sendo a ferramenta para desenvolvimento de software mais amada conforme a pesquisa *Stack Overflow Annual Developer Survey* de 2022, como mostra a figura 2.5.

Dada a grande popularidade da ferramenta e a sua utilização como tecnologia de conteinerização para a plataforma avaliada neste trabalho, o Amazon Sagemaker, é imprescindível conhecer os conceitos básicos e terminologia do Docker.

2.3.1 Conceitos Básicos

O Docker é uma plataforma *open-source* destinada a fazer o *build*, *deploy*, execução e gerenciamento de *containers*. Conforme mencionado na seção 2.2, duas características do uso de containers merecem destaque: isolamento entre containers e empacotamento de dependências. A seguir, detalham-se os conceitos fundamentais de ambas as funcionalidades da tecnologia Docker.

Figura 2.5: Tecnologias para desenvolvimento de software mais amadas/odiadas pelos desenvolvedores.



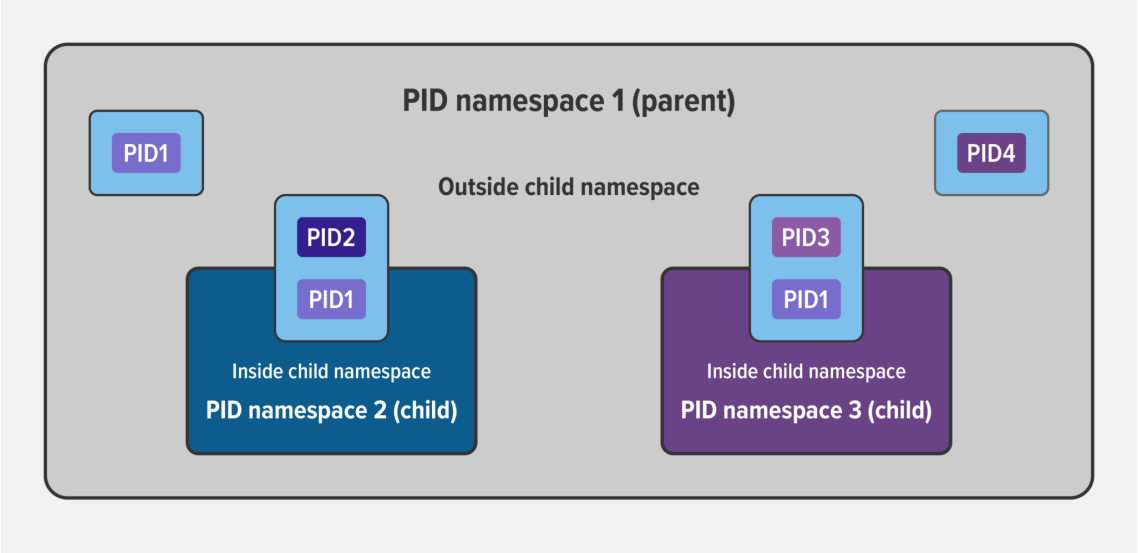
Fonte: (STACKOVERFLOW, 2022)

2.3.1.1 Isolamento

O isolamento entre aplicações, importante característica dos *containers*, é alcançado via dois conceitos fundamentais do kernel Linux: *namespaces* e *cgroups*.

Conforme explicado em Kalken (2023), *namespaces* são uma funcionalidade do kernel Linux que permitem particionar recursos do Kernel para um processo ou grupo de processos, ajudando a isolar recursos como: sistema de arquivos, rede (tabela de roteamento, endereços IP, firewall e outros) e comunicação inter-processos. Já os *cgroups* são um recurso do kernel Linux que permite a limitação e contabilidade de recursos para um único processo ou grupos de processos, ajudando a garantir a qualidade de serviço quanto a recursos como: CPU, memória, disco e rede. Ambos os conceitos são exemplificados nas figuras 2.6 e 2.7.

Figura 2.6: Exemplo de utilização de *namespaces* para limitar o grupo de PIDs visível por um processo. Na figura a seguir, ambos os processos filhos 2 (PID 2) e 3 (PID 3) não conseguem ver outros processos além de seu *namespace*, e possuem internamente PID 1 dentro dos seus respectivos *namespaces*.



Fonte: (KALKEN, 2023)

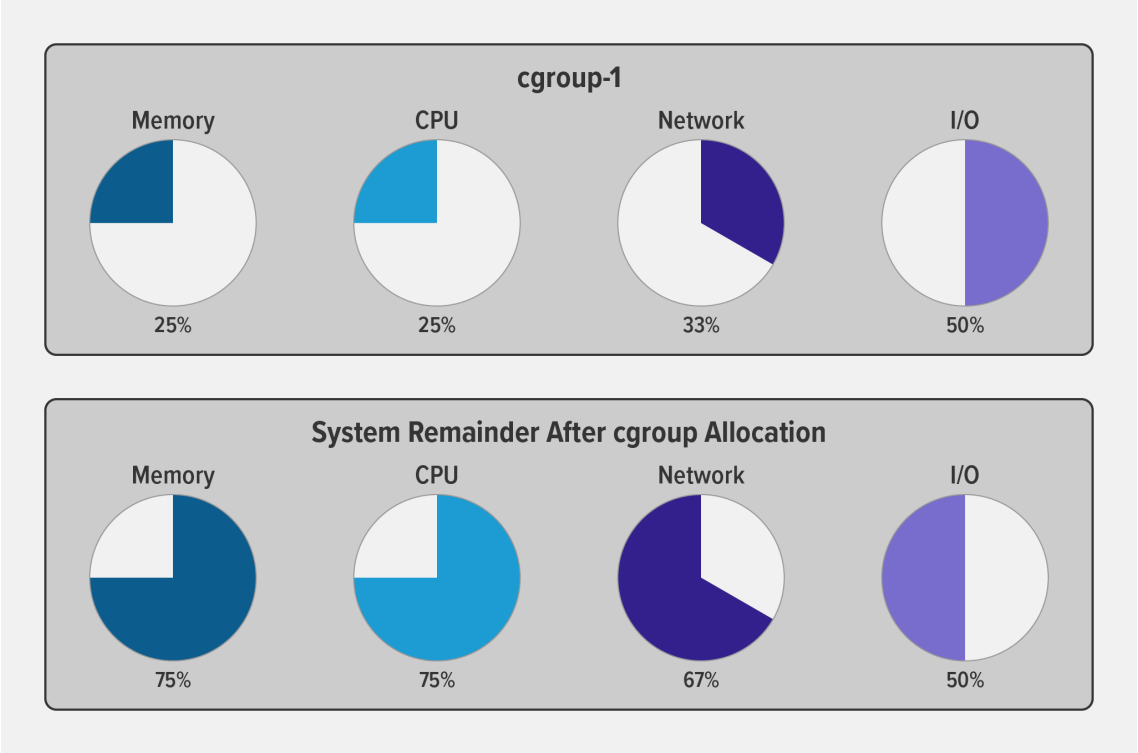
2.3.1.2 Empacotamento de dependências

No *Docker*, o empacotamento das dependências de uma aplicação é alcançado através do conceito de imagem. Análogo aos *snapshots* de máquinas virtuais, as *Docker images* são arquivos que contém instruções lidas pelo Docker *runtime* para a construção do *container*. Neste arquivo, que serve como uma espécie de *template*, estão contidos tanto o código da aplicação como bibliotecas, ferramentas e quaisquer arquivos necessários para a aplicação executar.

Uma *Docker image* é composta de camadas, representações imutáveis da imagem que possuem uma relação hierárquica de dependência com as camadas que as antecedem. Seguindo o conceito de sistema de arquivos *copy-on-write*, uma camada N possuí uma cópia atualizada apenas dos arquivos adicionados ou atualizados em relação à camada N-1. Através dessa estrutura, o processo de *build* e *download* de imagens é otimizado, já que duas imagens com camadas intermediárias iguais podem compartilhar da mesma camada. Outro benefício do uso de camadas *readonly* é a possibilidade de compartilhar a mesma imagem com diferentes instâncias de *containers* sendo executados, ajudando a reduzir o espaço em disco utilizado por várias instâncias de um mesmo *container*, conforme mostra a figura 2.8.

Conhecendo a estrutura de camadas por trás de uma imagem *Docker*, pode-se

Figura 2.7: Exemplo de utilização de *cgroups* para limitar o uso de recursos por um grupo de processos. A figura mostra a limitação de recursos como CPU, rede, memória, network e IO a ser aplicada a um grupo de processos e o restante dos recursos disponível para outros processos do sistema.

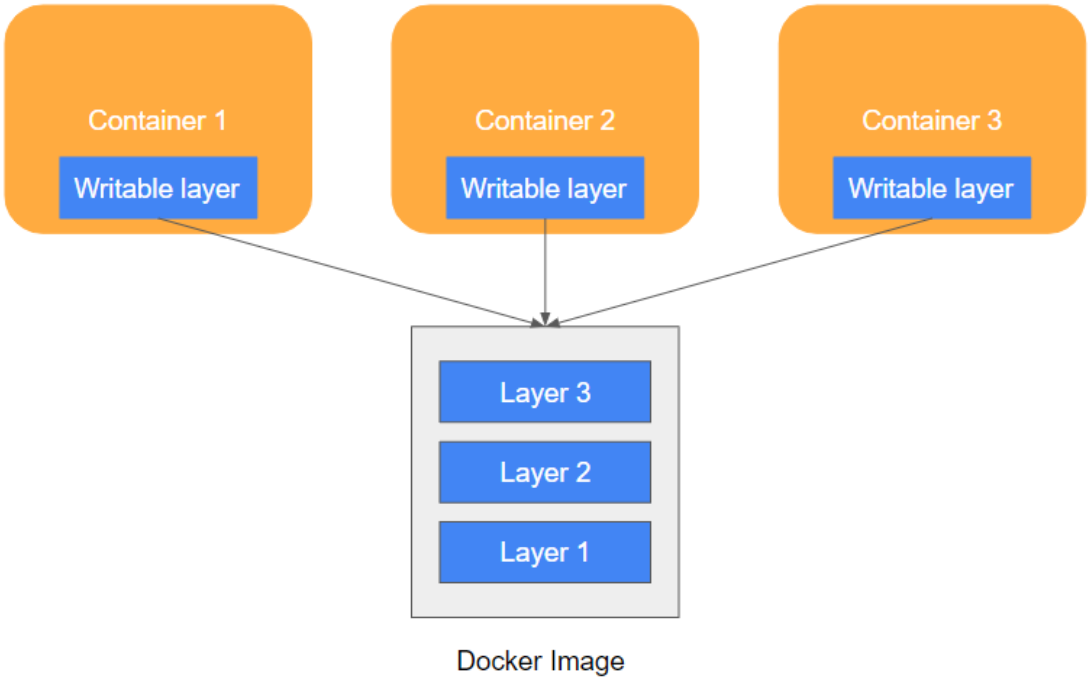


Fonte: (KALKEN, 2023)

otimizar o tempo de *build* das imagens. Como as camadas dependem diretamente das camadas que as antecedem, uma boa prática ao descrever imagens é primeiramente instalar as bibliotecas e dependências da aplicação, e só posteriormente fazer a cópia do código da aplicação propriamente dito. Desta maneira, as camadas anteriores à camada que copia o código da aplicação conseguem ser reaproveitadas da *cache*, acelerando o tempo de *build* das imagens.

Outra otimização comumente empregada é o uso de *multistage builds*, que consistem na utilização de mais de uma imagem base para realizar a build de uma imagem. Conforme mostrado na figura 2.9, primeiramente utiliza-se uma imagem base para instalar as dependências e realizar, por exemplo, a compilação do código-fonte, em seguida usa-se uma imagem base nova copiando-se apenas os componentes necessários para executar a aplicação. Esse processo reduz o tamanho das imagens e é essencial para redução de custo financeiro de repositórios de imagens e para reduzir o tempo de *deploy* de novas imagens em produção.

Figura 2.8: Uma imagem *Docker* é formada por camadas, que possuem uma dependência hierárquica entre si. Ao executar múltiplas cópias de *containers* é possível reutilizar os conteúdos da imagem, visto que a imagem é imutável.Cada *container* possui apenas uma pequena camada *writable*, que contém as mudanças em relação à imagem base.



Fonte: Elaborado pelo autor

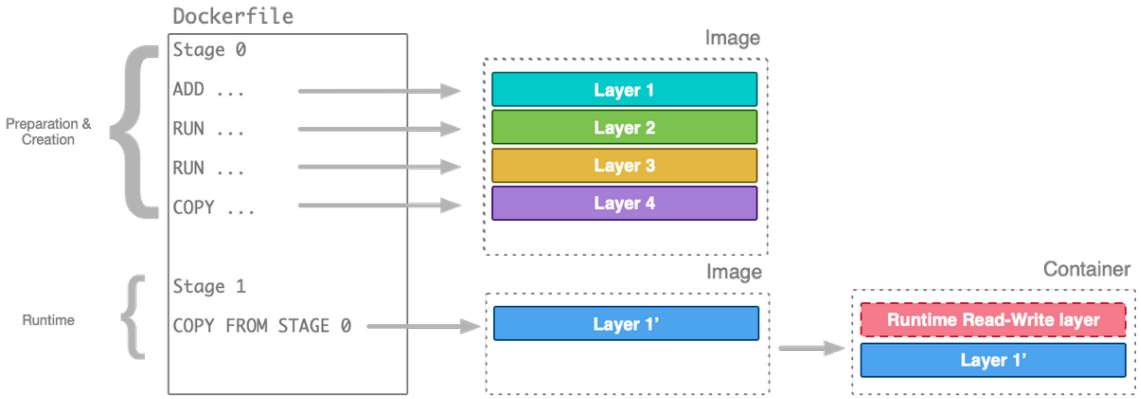
2.4 Computação em nuvem

A computação em nuvem consiste na disponibilidade de recursos computacionais, principalmente de processamento e armazenamento, sob demanda e sem o gerenciamento a cargo do usuário. Sua principal vantagem provém da facilidade em alocar recursos, sem a necessidade de gerenciá-los por completo. Assim, a computação em nuvem vem crescendo como modelo de infraestrutura usada para prover os mais diversos tipos de aplicações, principalmente em startups, onde deseja-se lançar um produto gastando o mínimo de investimento em termos financeiros e de tempo até mercado.

Baseado nos tipos de *cloud computing* definidos em Redhat (2022a), existem três principais modelos de serviço da cloud:

- **IaaS:** *Infrastructure as a Service* ou Infraestrutura como serviço, consiste em um modelo focado no oferecimento de recursos de infraestrutura sob demanda. Como um exemplo, o serviço EC2 (*Elastic Compute Cloud*) da AWS (*Amazon Web Services*), é um serviço onde os usuários conseguem alugar máquinas sob demanda,

Figura 2.9: Processo de otimização do build de uma imagem Docker através do uso de *multistage builds*. A primeira imagem base, *stage 0*, é utilizada apenas para a preparação do código-fonte e instalação de dependências, sendo a última imagem base, *stage 1*, consideravelmente menor, pois só copia da imagem base os arquivos necessários para a execução do código.



Fonte: (THAKUR, 2020)

pagando pelas horas utilizadas. Nota-se, que neste serviço, o foco é na infraestrutura, e estas são disponibilizadas sob a premissa de sua configuração ser totalmente a cargo do usuário.

- **PaaS:** *Platform as a Service* ou Plataforma como serviço, consiste em um modelo focado no oferecimento de uma plataforma completa sob demanda. Diferentemente do modelo IaaS, as unidades de computação são disponibilizadas pré-configuradas, tirando da responsabilidade do usuário a necessidade de gerenciar configurações de infraestrutura. Nesse modelo, mais valor agregado é provido ao usuário, já que o mesmo pode focar no desenvolvimento de código voltado ao domínio do problema que sua aplicação visa resolver. Um exemplo de tal plataforma é a Amazon Sagemaker, uma plataforma para *deploy* de modelos de *machine learning*, onde o provisionamento de servidores e configurações de rede são totalmente gerenciadas pela Amazon.
- **SaaS:** *Software as a Service* ou Software como serviço, consiste em um modelo onde a aplicação em sua totalidade é entregue pronta ao usuário, abstraindo todos os aspectos de desenvolvimento e provisionamento da mesma. Um exemplo de tal ferramenta é a *AWS Glue DataBrew*, que permite que usuários realizem tarefas de normalização e limpeza de dados sem escrever nenhuma linha de código.

Devido ao crescimento do uso de Computação em nuvem, existem hoje inúmeros provedores de *cloud* disponíveis, sendo as mais famosas: AWS (*Amazon Web Services*),

GCP (*Google Cloud Plataform*) e *Microsoft Azure*. Não obstante, cada empresa fornece uma grande diversidade de serviços, que estão à disposição dos desenvolvedores para diferentes tipos de aplicações e necessidades.

Como exemplo das possibilidades para *deploy* de modelos de *machine learning*, a Amazon disponibiliza uma plataforma especializada denominada *Amazon Sagemaker*, que visa abstrair conceitos de provisionamento de infraestrutura e facilitar o *deploy* de modelos. Mesmo em uma única plataforma, existem diferentes possibilidades, como o uso de *serverless functions*, aluguel de máquinas dedicadas para inferência em tempo real (*real-time inference*) e aluguel de máquinas para inferência em *batch* (*offline inference*). Cada modelo possui vantagens e desvantagens, que deverão ser analisadas segundo as necessidades de cada aplicação. A descrição da plataforma assim como os diversos serviços oferecidos estão disponíveis em Amazon (2019) e Liberty et al. (2020).

Um aspecto muito importante da utilização de recursos da *cloud* é o custo financeiro, comumente medido na unidade de dólares por hora. Normalmente, as máquinas disponibilizadas são organizadas em torno de algum recurso de computação específico como, memória, CPU ou armazenamento secundário, e também quanto à finalidade para a qual são otimizadas, como: propósito geral ou *accelerated computing*, sendo as últimas equipadas com co-processadores e GPUs capazes de acelerar alguns tipos específicos de aplicação, como *machine learning*. Em um mesmo tipo de instância o custo financeiro é diretamente proporcional a quantidade de recursos (CPU, memória, armazenamento).

2.4.1 Problemas no provisionamento de infraestrutura

Nesse contexto, é necessário escolher não somente o tipo de instância mais adequado a aplicação, mas o *grading*, tamanho em relação aos recursos, mais adequado. Dois casos devem ser evitados:

- **Under-provisioning:** Nesse cenário, alocamos menos recursos que o necessário para a aplicação, o que se traduz, por exemplo, em mais tempo de execução e pior experiência para o usuário.
- **Over-provisioning:** Nesse cenário, alocamos mais recursos que o necessário para a aplicação, o que se traduz em aumento desnecessários de custo financeiro.

Ambos os casos estão descritos em Wei et al. (2015) e podem ser melhor ilustrados na figura 2.10.

2.4.2 *Serverless computing*

Um paradigma emergente para *deploy* de aplicações na nuvem, é o uso de *serverless computing*. Nesse modelo, aplicações são executadas sem que o desenvolvedor precise provisionar e configurar qualquer tipo de infraestrutura.

É importante notar que nesse modelo de computação ainda existem servidores, no entanto, estes são abstraídos dos desenvolvedores da aplicação. Fica a cargo da provedora de computação em nuvem gerenciar os servidores, realizar a manutenção, aplicar *patches* de segurança e escalar a infraestrutura frente a um aumento da demanda.

Diferentemente de um modelo IaaS, onde a provedora de computação em nuvem fornece recursos de computação que estarão sempre à disposição do desenvolvedor, os chamados *always-on servers*, em uma arquitetura *serverless* os recursos são provisionados sob demanda, de maneira que os recursos são precificados pelo seu uso, geralmente tendo preços mais atrativos, já que o desenvolvedor deixa de pagar pelo período onde a aplicação ficaria em *stand-by*.

Conforme mencionado em Baldini et al. (2017) e Redhat (2022b), a arquitetura *serverless* apresenta inúmeros benefícios, dentre os quais:

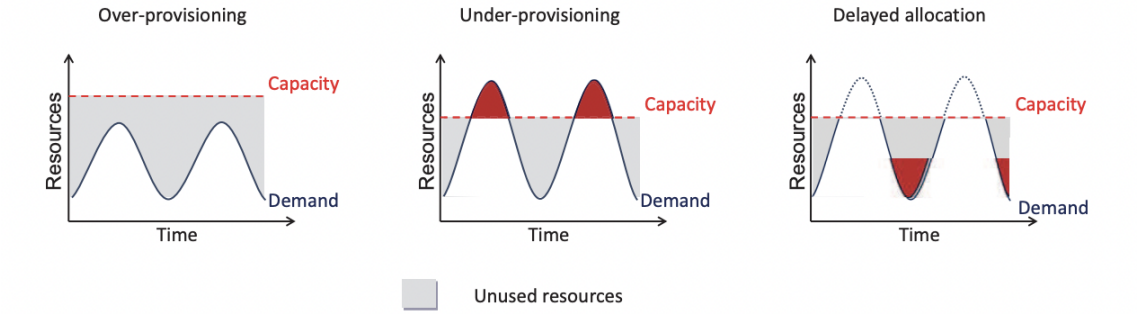
- Custo financeiro: Como os recursos são provisionados sobre demanda e o desenvolvedor só paga pelo tempo efetivamente utilizado pela aplicação, o custo é menor.
- Produtividade: Como não é necessário definir políticas de escalonamento e manutenção dos servidores, os desenvolvedores conseguem focar na lógica de negócio da aplicação.
- Diminuição da latência: Em alguns provedores de *cloud* é possível diminuir a latência para o usuário final, já que não existe um servidor único de origem e é possível configurar vários servidores mais próximos do usuário final.

No entanto, as arquiteturas *serverless* apresentam algumas desvantagens que devem ser cuidadosamente analisadas antes da sua utilização em alguma aplicação, são estas:

- Restrições: A utilização de serviços *serverless* estão sujeitos às restrições de recursos de computação, como: tempo máximo de resposta (*timeouts*), CPU e memória.
- Acoplamento com tecnologias do vendedor: Diferentes provedores de Cloud possuem diferentes restrições, que podem suscitar a utilização de tecnologias específicas de cada provedor, como: ferramentas de log, autenticação e autorização. Esse

processo pode gerar dependência explícita de tecnologias, criando acoplamento.

Figura 2.10: Diferentes problemas ao provisionar recursos na nuvem. Nos gráficos ilustrados o eixo horizontal denota a passagem do tempo e o eixo vertical a quantidade de recursos (CPU, memória ou até mesmo número de servidores), já a linha vermelha pontilhada denota a capacidade máxima provisionada para um sistema e a linha preta denota o consumo de tais recursos.



Fonte: (WEI et al., 2015)

2.5 IaC

Consoante ao descrito em Redhat (2022b), IaC (*Infrastructure as Code*) ou infraestrutura como código, é o gerenciamento de infraestrutura através de código, ao invés do uso de processo manuais. Ao invés de manualmente realizar o provisionamento e configuração de servidores, CDNs, bancos de dados e demais componentes necessários para o funcionamento de uma aplicação, no modelo IaC todos os recursos e suas configurações são descritos em arquivos, como código.

A descrição da infraestrutura como código permite que se tenha uma visão e gerenciamento melhor da infraestrutura, já que os componentes e configurações necessárias se encontram descritos em um único lugar, evitando o surgimento de mudanças não documentadas. Além disso, o uso de IaC facilita também no compartilhamento e reprodutibilidade de infraestruturas, na medida que pode-se criar *templates* reutilizáveis de padrões arquiteturais já pesquisados e testados.

Para descrever a infraestrutura como código, dois paradigmas distintos podem ser utilizados:

- **Declarativo:** No método declarativo, descrevem-se as configurações e recursos desejados para uma dada infraestrutura, sem que seja necessário especificar a dependência e ordem de provisionamento entre os recursos. Cabe a ferramenta de

IaC traduzir a declaração de recursos em um conjunto de passos sequenciais que provisione infraestrutura equivalente à declaração fornecida.

- **Imperativo:** No método imperativo, descreve-se um conjunto de passos sequenciais para se obter o estado da infraestrutura desejado. Cabe ao usuário conhecer o estado atual da arquitetura, bem como a ordem dos passos a serem executados para se chegar ao estado pretendido.

Como exemplo de ferramentas de IaC declarativas, destacam-se: Terraform, Puppet e CloudFormation. Já como exemplo de ferramentas de IaC imperativas, destacam-se: Chef e Ansible.

O provisionamento de infraestrutura através do uso de ferramentas IaC declarativas vem ganhando destaque, sendo utilizado para provisionamento de aplicações de análise de imagens médicas, como em Ivanova, Borovska and Zahov (2018) e o suporte a aplicações de HPC heterogêneas, como proposto em Vladusic and Radolovic (2020).

A descrição da infraestrutura como código de maneira declarativa traz consigo muitas vantagens alinhadas com os objetivos deste trabalho, sendo as principais:

- **Reprodutibilidade:** A especificação da infraestrutura como código permite que as infraestruturas sejam facilmente destruídas e reconstruídas, com a garantia de que os recursos serão provisionados exatamente como descritos. Isso permitirá que as diferentes infraestruturas pensadas para o *deploy* dos modelos de *machine learning* sejam facilmente reproduzíveis e testáveis por pessoas interessadas no trabalho.
- **Facilidade de provisionamento:** O uso de ferramentas de IaC permite que infraestruturas sejam provisionadas a partir de uma linguagem descritiva, assim abstraindo os passos necessários, em conjunto com a ordenação desses, para o provisionamento dos recursos. A linguagem descritiva facilita o desenvolvimento, tornando o trabalho de pesquisa e teste de alternativas mais rápido.

3 TRABALHOS RELACIONADOS

Neste capítulo, serão abordadas pesquisas relacionadas ao uso de *deep learning* para a detecção de retinopatia diabética. Conforme mostrado a seguir, a detecção de RD através do uso de *deep learning* tem se mostrado uma alternativa promissora para a problemática da escassez de mão de obra especializada para a realização dos diagnósticos.

Neste mesmo contexto, é essencial abordar trabalhos relacionados ao *deploy* de tais modelos, estágio fundamental para disponibilização dos algoritmos para o usuário final. O uso de computação em nuvem, em razão do baixo investimento inicial e possibilidade de escalabilidade, se apresenta como uma alternativa viável à disponibilização dos modelos, como será discutido a seguir.

Primeiramente, na seção 3.1 serão abordados trabalhos relacionados ao uso de *deep learning* para a detecção de RD e a possibilidade de uso de tais modelos como ferramentas para auxiliar autoridades médicas a priorizar casos de RD. Na seção 3.2 serão abordados trabalhos relacionados ao uso de computação em nuvem como alternativa para o *deploy* de modelos de *deep learning* de maneira geral. Finalmente, na seção 3.3 será abordado a importância da reprodutibilidade em pesquisas científicas, além de trabalhos que serviram de inspiração para a utilização de ferramentas destinadas ao aumento da reprodutibilidade do presente trabalho.

3.1 *Deep learning* para classificação de RD

O aprendizado de máquina vem ganhando destaque nas últimas décadas, mostrando-se cada vez mais promissor para tarefas de classificação, como o reconhecimento de objetos ou o diagnóstico de doenças. Conforme mostrado em Dai et al. (2021), a partir do sistema DeepDR, baseado em uma rede neural residual, é possível detectar a presença de diferentes graus de diabetes com área sob a curva de característica operacional do receptor de $0,916$ até $0,970$ para cada grau de RD.

Outras arquiteturas de redes neurais também podem ser utilizadas para a detecção de RD, como mostra o trabalho Moreira et al. (2020), baseado na arquitetura Inception-V3. Uma conclusão muito importante do trabalho é a respeito do impacto da resolução da arquitetura na detecção da doença, onde após um aumento da resolução das imagens foi possível aumentar a precisão das detecções em 6%.

Interessantemente, até mesmo modelos menores conseguem realizar a detecção

da retinopatia diabética com sucesso. Conforme mostrado em Pavate et al. (2020), após o treinamento da arquitetura MobileNet utilizando o *dataset* APTOS, foi possível obter um modelo com acurácia e precisão de 95%. Conforme mencionado na seção 5.2, a fim de analisar alternativas para o *deploy* dos modelos considerando diferentes tamanhos de arquitetura de rede neural, uma das arquiteturas utilizadas foi a MobileNet, devido ao seu tamanho reduzido.

O tamanho reduzido da arquitetura MobileNet já foi explorado para a detecção de RD até mesmo em aplicativos de celular, conforme reportado em Bidari et al. (2021). Neste trabalho, utilizou-se a arquitetura MobileNet de tal forma que a inferência dos modelos possa ser executada em celulares, portanto permitindo que um diagnóstico complementar seja realizado para usuários que não possuem uma conexão regular à internet, como usuários que residem na área rural.

3.2 Computação em nuvem para *deploy* de modelos de *Deep Learning*

A fim de proporcionar uma maior acessibilidade desses modelos para o usuário final, é imprescindível pensar em formas de provisionar os recursos de uma maneira performática e de baixo custo financeiro. Nesse contexto, o uso de *cloud computing* para o provisionamento da infraestrutura de um sistema de análise de imagens médicas já foi estudado em Ivanova, Borovska and Zahov (2018). Neste trabalho, foi utilizado uma plataforma baseada na provedora de computação em nuvem AWS, em conjunto da linguagem Terraform para descrever a infraestrutura seguindo os princípios de IaC (*Infrastructure as Code*).

De maneira semelhante, o uso de *cloud computing* para deploy de modelos de *deep learning* em específico já foi abordado em Ishakian, Muthusamy and Slominski (2018), onde estudou-se fazer o *deploy* de tais modelos usando o serviço de *serverless computing* da Amazon, o AWS Lambda. Muito embora o uso de funções *Lambda* simplifique em larga escala o processo de gerenciamento da infraestrutura e até mesmo reduza os custos de operação, seu uso traz consigo penalidades para o desempenho da aplicação, na forma de *cold-starts* e *warm-starts*, como mostram os autores.

Visto que em um paradigma de *serverless computing* os recursos são provisionados sob demanda, sendo destruídos após um tempo de ociosidade, as aplicações sofrem de um fenômeno chamado *cold-start*, caracterizado por uma latência elevada em sua primeira ativação. Já quando os recursos se encontram prontamente provisionados, a latência

Tabela 3.1: Tabela de latências por modelo utilizado no trabalho Bidari et al. (2021).

Modelo Usado	**Tempo levado para mostrar os resultados**
Resnet 50	4 s
MobileNet	2 s

é consideravelmente menor, configurando o que se chama de *warm-start*.

Ainda segundo mostrado em Ishakian, Muthusamy and Slominski (2018), a latência tanto para casos de *cold-start* como para casos de *warm-start* tende a diminuir, à medida que a memória aumenta. No entanto, essa diminuição é limitada, havendo casos onde o aumento da memória disponível aumenta o custo financeiro por ativação sem trazer ganhos significativos para o desempenho. Não obstante, o tamanho do modelo utilizado impacta diretamente em ambas as latências, portanto sendo importante considerar modelos de diferentes tamanhos ao avaliar otimizações e diferentes configurações.

O *deploy* de modelos de *deep learning* para a detecção de RD já foi estudado em Bidari et al. (2021), onde, conforme comentado anteriormente, os autores utilizaram a arquitetura MobileNet para prover inferências localmente em celulares, sem a necessidade de conexão à internet. Neste mesmo trabalho, também foi explorado a utilização de um modelo maior, baseado na arquitetura ResNet 50, sendo esta disponibilizada por meio de uma aplicação WEB. Embora não existam detalhes da infraestrutura utilizada para *deploy* dos modelos, segundo reportado, a latência máxima para realização da predição foi de 10 segundos para ambos os modelos. O tempo levado para exibição dos resultados para cada modelo utilizado é apresentado na tabela 3.1.

3.3 Reprodutibilidade

Um aspecto muito importante de qualquer pesquisa é sua reprodutibilidade, que diz respeito à possibilidade de reproduzir os experimentos elaborados e obter os mesmos resultados obtidos na pesquisa original. Dada a natureza complexa de *software* e *hardware*, muitas vezes a documentação dos experimentos não é suficiente para a plena reprodução da pesquisa, já que dependências de software e configurações de ambientes são difíceis de documentar em sua completude.

Conforme mostrado em Cacho and Taghva (2020), através de uma pesquisa com estudantes de graduação da Universidade de Nevada, cerca de 89% dos estudantes afirmam já ter lido um trabalho que não disponibiliza os dados utilizados na pesquisa. Conforme mencionado no trabalho, tal problemática é chamada de crise da reprodutibilidade,

e é essencial serem empregadas ferramentas e metodologias para melhorar a reprodutibilidade das pesquisas.

Ainda conforme o trabalho Cacho and Taghva (2020), uma alternativa para aumentar a reprodutibilidade de pesquisas é o uso de ferramentas que providenciem ambientes virtuais, que possam ser duplicados e executados com facilidade, como o uso de VMs ou *containers*. Essas ferramentas melhoram a reprodutibilidade das pesquisas, na medida que código, dependências de *software* e configurações de ambiente estão contidos em um único objeto.

O uso do Docker para a melhoria da reprodutibilidade de experimentos científicos já foi abordado em Boettiger (2015). Segundo o autor, o uso de Docker nos ajuda a resolver quatro problemas fundamentais que se opõem a plena reprodução das pesquisas:

1. Gerenciamento de dependências: Visto que imagens Docker mantêm em um mesmo objeto código, bibliotecas e todas as dependências necessárias para executar os experimentos, não há problemas com dependências faltantes ou incompatíveis para a replicação dos experimentos.
2. Documentação imprecisa: Visto que a construção da imagem Docker é realizada por meio de um único *script*, a Dockerfile, as dependências necessárias para a execução do projeto ficam documentadas em uma única fonte de verdade, dificultando divergências entre a documentação do projeto e as dependências realmente utilizadas.
3. Envelhecimento de dependências: Conforme as dependências de um código envelhecem, torna-se cada vez mais difícil obtê-las, o que pode gerar dificuldade na hora de reproduzir uma pesquisa. Essa problemática é evitada através do uso de binários da imagem *docker*, que contém uma cópia de todas as dependências necessárias para a execução dos experimentos e pode ser facilmente compartilhada por meio de repositórios de imagens públicos.
4. Dificuldade de uso: Uma solução técnica para o problema de reprodutibilidade deve apresentar pouca ou nenhuma dificuldade de uso, para que a mesma de fato seja utilizada. Segundo o autor, o Docker provê facilidade de uso através das seguintes características: uso em ambiente local facilitado, reuso modular, portabilidade de ambientes, versionamento e compartilhamento através de registros públicos.

Não limitado à utilização de ferramentas como Docker e VMs, o uso do paradigma IaC também contribui para a melhora da reprodutibilidade de pesquisas, como

apontado em Aguilar et al. (2022). Neste trabalho, os autores propõem um paradigma definido como *Experiments as Code*, onde os experimentos não são apenas documentados, mas também é disponibilizado todo o código necessário para o provisionamento da infraestrutura, *deploy*, gerenciamento e análise dos dados do experimento. No trabalho, são utilizadas ferramentas como: Docker, jupyter-lab e Terraform.

Conforme detalhado no capítulo 4, inspirado pelos trabalhos previamente citados, o presente trabalho utiliza o Docker e o Terraform para aumentar a reprodutibilidade da pesquisa. Não obstante, conforme comentado na seção 6.3, todas as imagens Docker utilizadas para a realização dos experimentos são disponibilizadas em um repositório de imagens público, seguindo as boas práticas definidas em Boettiger (2015).

4 TECNOLOGIAS UTILIZADAS

Neste capítulo são apresentadas as tecnologias utilizadas para a realização do *deploy* de modelos de detecção de retinopatia diabética na nuvem, assim como as tecnologias utilizadas para a realização dos experimentos destinados à avaliação dos diferentes serviços e otimizações realizadas. Começaremos com uma breve visão geral das tecnologias utilizadas, na seção 4.1. Na sequência, exploram-se as tecnologias mais importantes para o trabalho.

Na seção 4.2 abordaremos conceitos pertinentes ao Tensorflow e em especial do formato de salvamento dos modelos. Na seção 4.3 exploraremos a provedora de computação em Nuvem AWS, focando nos serviços utilizados no trabalho. Finalmente, a seção 4.4 aborda conceitos da ferramenta Terraform, utilizada para o provisionamento da infraestrutura.

4.1 Visão Geral

Para o desenvolvimento do código destinado a realizar o processo de inferência dos modelos foi utilizado a linguagem Python, junto da biblioteca TensorFlow, amplamente utilizada na área de *Machine Learning*.

Para disponibilizar os resultados das inferências via uma API, foi utilizado o framework FastAPI, um framework WEB para Python destinado à construção de WEB APIs. Além disso, também foi utilizado o servidor WEB NGINX, um popular servidor WEB Linux, comumente utilizado como *load balancer*, *proxy* reverso e cache HTTP.

Para gerenciar os processos da API Python, foram utilizados Gunicorn, um servidor Python compatível com o padrão WSGI, e Uvicorn, um servidor Python compatível com o padrão mais novo ASGI, utilizado pela biblioteca previamente mencionada FastAPI.

Para gerenciar as dependências do projeto e permitir flexibilidade na hora de avaliar diferentes modelos e versões de pacotes, foi utilizado Conda, um gerenciador de pacotes e ambientes *Open Source* e multi-plataforma.

A fim de disponibilizar a API em um *container*, portável e executável em várias plataformas diferentes, utilizou-se o Docker, uma plataforma de código aberto destinada à criação e administração de ambientes isolados via unidades denominadas *containers*. Além disso, a plataforma dispõe ferramentas para o compartilhamento de imagens, am-

plamente utilizados neste trabalho para o upload das imagens para um repositório privado na nuvem.

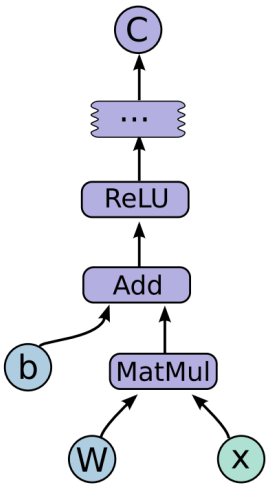
Como plataforma de computação em nuvem, utilizou-se a AWS (Amazon Web Services), uma popular plataforma, conforme mostrado na figura 4.5, para a utilização de recursos computacionais na nuvem de maneira escalável e resiliente.

Finalmente, utilizou-se a ferramenta Terraform para permitir a descrição da infraestrutura como código de maneira declarativa e facilitar a reprodutibilidade e prototipação das alternativas estudadas neste trabalho.

4.2 TensorFlow

Consoante ao definido em Abadi et al. (2016), o Tensorflow é uma interface para a definição de algoritmos de *machine learning* e a subsequente implementação de tais algoritmos. No Tensorflow, tanto computação como estado são representados por *data flow graphs*, conforme explicitado na figura 4.1. Essa representação é então convertida em computação em vários tipos de arquiteturas heterogêneas, possibilitando que desenvolvedores testem arquiteturas diferentes sem precisar refatorar código.

Figura 4.1: Exemplo de *dataflow graph* utilizado para a representação de algoritmos no TensorFlow. Em um *dataflow graph*, os nodos representam computações e as arestas dependências de dados ou resultado de outras computações. Por exemplo, a operação MatMul depende das arestas W e X, e a operação Add depende do resultado de MatMul e b.



Fonte: (ABADI et al., 2016)

De suma importância para este trabalho, a biblioteca TensorFlow disponibiliza formas para o salvamento de modelos e seus valores de parâmetro após a fase de treina-

mento. O formato SavedModel, utilizado no trabalho, é uma forma de guardar computação e valores de parâmetros de treinamento sem que seja necessário ter acesso ao código de construção e treinamento do modelo, sendo muito útil para o compartilhamento dos modelos. O formato é analisado em mais detalhes na subseção a seguir.

4.2.1 SavedModel

O formato SavedModal é salvo em disco na forma de um um diretório, contendo quatro componentes de interesse:

- `assets`: Subdiretório que contém arquivos auxiliares, como vocabulários.
- `saved_model.pb`: *SavedModel protocol buffer*, é o arquivo que contém a definição dos grafos e configuração de treinamento, seguindo o formato *protocol buffers*.
- `variables`: Esse diretório contém arquivos relacionados a checkpoints de treinamento.
- `assets.extra`: Esse diretório contém arquivos extras, não diretamente usados pelo TensorFlow, contendo, por exemplo, informações úteis aos consumidores do modelo.

Para todos os modelos do trabalho, o diretório `assets.extra` não foi utilizado. Restando então o diretório *variables* com apenas um *checkpoint* de treinamento, o checkpoint de treinamento padrão, e o arquivo `saved_model.pb`, contendo a definição do grafo que descreve a computação do modelo.

4.3 AWS

Dentre os diversos serviços oferecidos pela AWS, foram utilizados: S3, ECR e Amazon Sagemaker. Nas subseções a seguir, realiza-se uma breve descrição de cada serviço utilizado seguido do motivo para sua utilização.

4.3.1 S3

O *Simple Storage Service* é um serviço de armazenamento de objetos na nuvem. Neste serviço, a unidade básica de armazenamento são objetos, armazenados em *buckets*.

No contexto deste trabalho, o serviço foi utilizado para armazenar os artefatos dos modelos descritos na seção 4.2.1. Foi criado um único *bucket*, denominado *dr-deploy-images*, cujos objetos são arquivos tar.gz, cada qual contendo os artefatos de cada modelo comprimidos.

4.3.2 ECR

O ECR, *Elastic Container Registry*, trata-se de um serviço para o armazenamento, gerenciamento, compartilhamento e *deploy* de imagens de *containers*. Dentre os formatos de manifesto de imagem suportados incluem-se: *Docker Image Manifest V2*, *Docker Image Manifest V1* e *Open Container Initiative (OCI) Specifications*.

Neste trabalho, o serviço foi utilizado para criar um repositório privado para o armazenamento de imagens Docker, *Docker Image Manifest V2*, que implementam a inferência dos modelos de *Deep Learning* seguindo o contrato *Bring your own container* do serviço *Amazon Sagemaker*, disponível em Stroppa (2021).

Complementarmente, o serviço foi utilizado para criar um repositório de imagens público, com o intuito de disponibilizar as imagens Docker necessárias para a reprodução dos experimentos, assim como comentado na seção 6.3.

4.3.3 Sagemaker

O Amazon Sagemaker é um serviço totalmente gerenciado para a construção, treinamento e *deploy* de modelos de *machine learning*. O serviço visa facilitar o processo de construção e *deploy* dos modelos, abstraindo do usuário, se desejado, a necessidade realizar a configuração de infraestrutura necessária para a execução dos modelos.

Dentre as funcionalidades do Amazon Sagemaker, neste trabalho utilizou-se a criação de endpoints para inferência dos modelos a partir de duas modalidades: *serverless inference* e *real-time inference*.

4.3.3.1 Serviços de inferência utilizados

Na modalidade *real-time inference* deve-se escolher entre diversos tipos de instância e seus respectivos tamanhos para o provisionamento dos modelos, de maneira que o custo financeiro é proporcional ao número de horas em que a instância foi provisionada,

mantendo-se funcionando independentemente de seu uso. Esse modelo é indicado para aplicações que possuem restrições relativas à latência, não podendo suportar variações abruptas na latência.

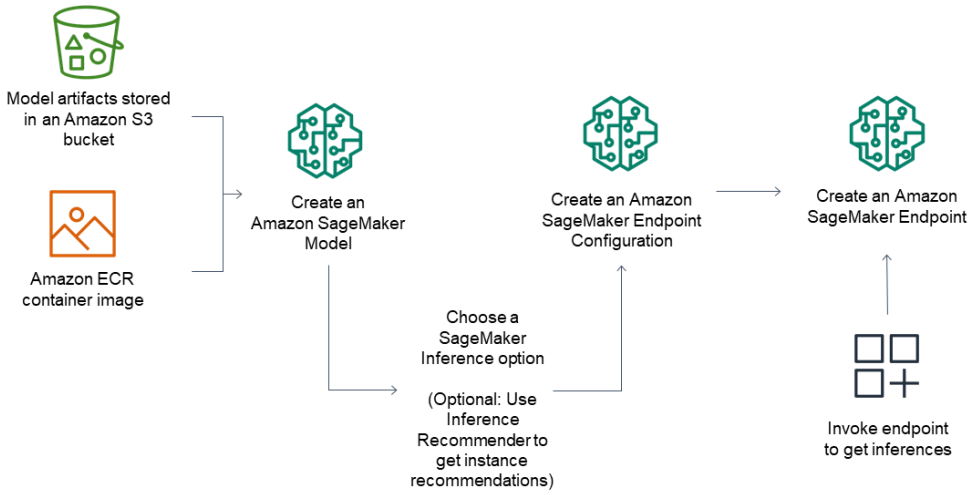
Na modalidade *serverless inference* escolhe-se apenas o tamanho de memória do *worker* dedicado a execução do modelo, sendo que a principal diferença relativa à modalidade *real-time inference* é o fato do provisionamento dos *workers* ser condicionado a demanda da aplicação. A tabela 4.1 resume os diferentes tamanhos de memória disponíveis e a relação de preço por segundo utilizado pela inferência.

Na modalidade *serverless inference*, caso a aplicação passe por longos períodos sem nenhum tráfego, os recursos provisionados são destruídos, sendo provisionados novamente apenas sob demanda. Tal modalidade é bastante atrativa para aplicações com longos períodos de espera entre cargas de trabalho, pois o custo financeiro é proporcional apenas ao tempo efetivamente utilizado para executar inferências.

4.3.3.2 Conceitos importantes

Ambos os tipos de serviço *real-time inference* e *serverless inference* utilizam os mesmos conceitos para o provisionamento de *endpoints*, demonstrados na figura 4.2 e descritos na lista a seguir:

Figura 4.2: Arquitetura do serviço Sagemaker Endpoints para o provisionamento de *endpoints*.



Fonte: (AMAZON, 2019)

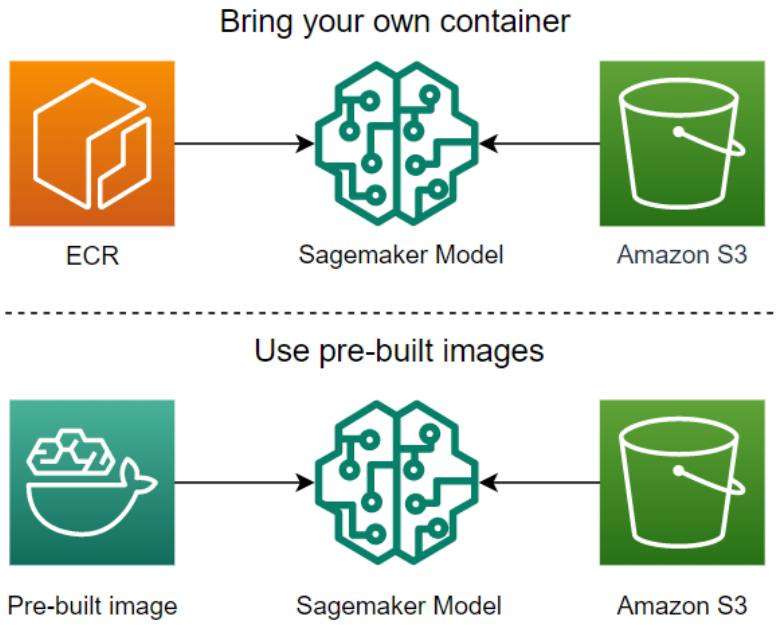
- *Model*: Um *Sagemaker Model* consiste na especificação de uma imagem de *container* que implementa a aplicação e uma URL do S3 que identifica um objeto

contendo os artefatos do modelo. Um mesmo *Sagemaker Model* pode ser utilizado para o provisionamento de diferentes endpoints.

- *Endpoint configuration*: Consiste na configuração para o provisionamento do endpoint, combinando um ou mais *Sagemaker Models* e configurações como tamanho da instância e tipo de serviço de inferência (*real-time inference* ou *serverless inference*).
- *Endpoint*: Um *Sagemaker Endpoint* é o provisionamento de um *endpoint* propriamente dito, a partir de um *endpoint configuration*. Portanto, é possível utilizar um *endpoint configuration* para provisionar várias cópias de um *endpoint* com a mesma configuração.

Ao definir um *Sagemaker Model* pode-se optar por duas modalidades: uso de imagens docker *pre-built* e *bring your own container*. A vantagem do uso da modalidade *bring your own container* é esta possibilitar a customização total do código da aplicação, além da possibilidade de uso de algoritmos de *Machine Learning* não limitados aos oferecidos pelas imagens *pre-built*. Ambas as modalidades são retratadas na figura 4.3.

Figura 4.3: Modalidades para a especificação de um *Sagemaker Model*. Na modalidade *Bring your own Container* é necessário hospedar a imagem do *container* em um registro de imagens, neste caso o ECR. Na modalidade *Use pre-built images*, isso não é necessário, utilizam-se imagens prontas disponibilizadas pela AWS.



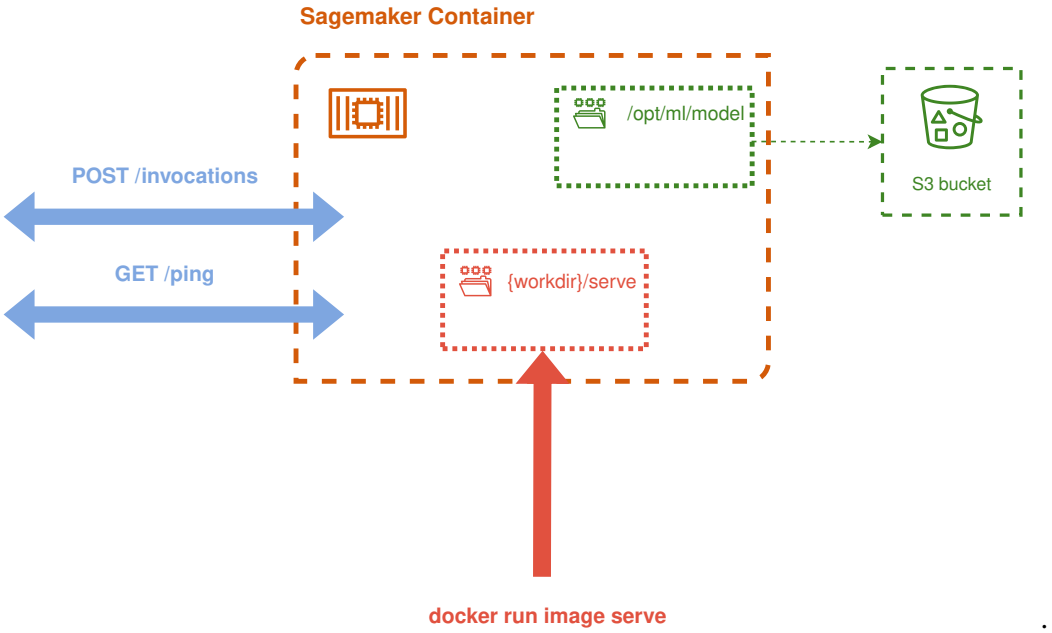
Fonte: Elaborado pelo autor

Na modalidade usada neste trabalho, *Bring your own container*, é necessário que o *container* implemente uma interface predefinida, utilizada pelo Sagemaker para fazer a

inicialização do *container*, invocação do endpoint e verificação da saúde da aplicação. A seguir, definem-se os elementos-chave da interface a ser implementada pelo *container*.

4.3.3.3 Interface do container Sagemaker

Figura 4.4: Interface definida pela plataforma Amazon Sagemaker para os serviços *real-time inference* e *serverless inference* sob a modalidade *Bring your Own Container*. Em azul, são representados os *endpoints* utilizados para retorno de predições e saúde do *container*. Em verde, é demonstrado a disponibilização dos artefatos de modelo armazenados no S3 em um dado diretório. Em vermelho, é mostrado o *entrypoint* de execução do *container*.



Fonte: Elaborado pelo autor

Assim como demonstrado na figura 4.4, primeiramente o *container* é executado utilizando o comando *docker run image serve*. A partir do argumento serve, o *Amazon Sagemaker* sinaliza que o *container* será utilizado para servir inferências do modelo a partir de um servidor HTTP, exposto na porta 80. Para a realização da inferência e checagem de saúde, dois *endpoints* HTTP precisam ser expostos:

- `GET /ping`: Este *endpoint* é invocado periodicamente a fim de verificar que o container está saudável e apto a continuar servindo requisições. O endpoint deve retornar um código de status HTTP 200, sendo qualquer outro código uma sinalização de que o *container* não está saudável e deve ser substituído.
- `POST /invocations`: Este *endpoint* é invocado para realizar as inferências a partir do modelo previamente definido. É importante notar que a entrada para a inferência é passada no *body* da requisição, e apenas *headers* explicitamente defini-

Tabela 4.1: Relação entre preço por segundo e quantidade de memória disponível para inferência do tipo *serverless*.

Memória (MB)	**Preço por segundo**
1024	0,0000200 USD
2048	0,0000400 USD
3072	0,0000600 USD
4096	0,0000800 USD
5120	0,0001000 USD
6144	0,0001200 USD

dos pelo serviço Sagemaker são repassados ao *container*, conforme detalhado em: Hudgeon and Nichol (2020a).

Além da interface previamente definida, o *container* precisa atender certos requisitos de *timeout* para a aplicação ser considerada saudável. Os requisitos são sumarizados na lista a seguir:

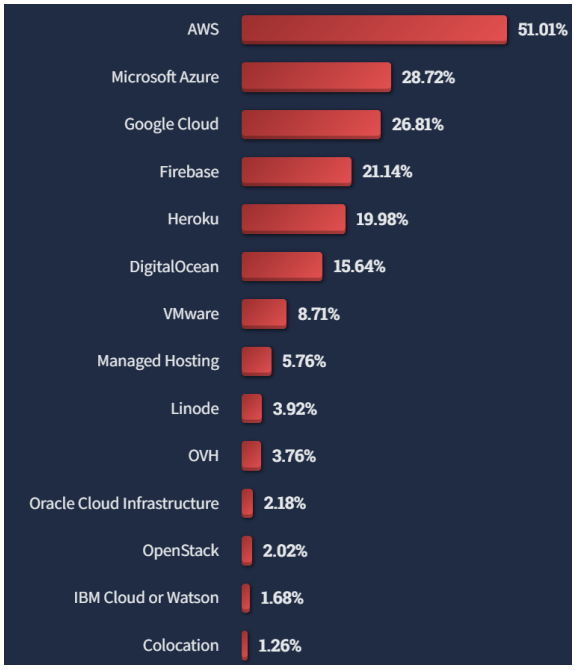
- O container deve conseguir aceitar *socket connections* em no máximo 250 milisegundos.
- O container deve conseguir responder ao *endpoint* `POST /invocations` em no máximo 60 segundos.
- O container deve conseguir responder ao *endpoint* `GET /ping` em no máximo 2 segundos.
- Durante a inicialização do container, este deve responder com sucesso a requisições ao *endpoint* `GET /ping` em no máximo 4 minutos, caso contrário o *endpoint* entrará em um estado de falha a qual não é recuperável.

4.4 Terraform

Assim como definido em HashiCorp (2021), o Terraform é uma ferramenta de infraestrutura como código *open-source* que permite o gerenciamento de recursos de computação em nuvem por meio de arquivos de configuração, permitindo o versionamento, reuso e compartilhamento de infraestrutura.

Não restrito ao provisionamento de recursos de computação de baixo nível, como instâncias para computação e recursos de armazenamento, o Terraform consegue provisionar serviços de alto nível como funcionalidades SaaS e até configurações de firewall de rede. Por exemplo, neste trabalho, o Terraform é utilizado para provisionar *endpoints* do

Figura 4.5: Plataformas de *cloud computing* mais utilizadas segundo a pesquisa anual do StackOverflow em 2022. Nota-se que a AWS é a plataforma mais utilizada.



Fonte: (STACKOVERFLOW, 2022)

Sagemaker, um serviço de alto nível para o provisionamento de inferência de modelos de *machine learning*.

Não restrito aos *endpoints* Sagemaker, o Terraform é utilizado para provisionar toda a infraestrutura do trabalho, como: bucket S3, repositórios de imagem Docker privados e públicos e configurações de permissões necessárias para fazer a integração entre os serviços citados anteriormente. A seguir, detalham-se o funcionamento, conceitos básicos e fluxo de trabalho da ferramenta.

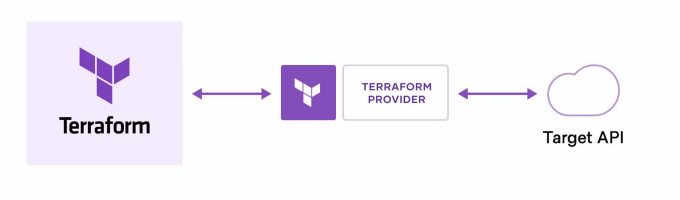
4.4.1 Funcionamento

Haja vista a natureza declarativa do Terraform, é responsabilidade da ferramenta tomar ações de maneira que o código esteja sincronizado com o estado da infraestrutura descrito. A fim de alcançar esse objetivo, a ferramenta utiliza o conceito de *state*, um arquivo que contém informações usadas para vincular recursos de infraestrutura da nuvem com os recursos definidos em código.

Tendo mapeado os recursos definidos em código com os respectivos recursos de infraestrutura provisionados, é responsabilidade do *provider* mapear as configurações para chamadas de API do serviço de computação em nuvem utilizado. Um *provider* é

como um *plug-in*, que permite ao Terraform fazer um mapeamento entre os recursos declarativos e chamadas de API imperativas de um dado serviço de computação em nuvem. A figura 4.6 ajuda a ilustrar a função do *provider*.

Figura 4.6: Papel de um *provider* do Terraform. Assim como um *plug-in*, o provider tem a função de traduzir configurações Terraform declarativas em chamadas de API de um dado provedor de computação em nuvem.



Fonte: (HASHICORP, 2021)

4.4.2 Conceitos básicos

No Terraform, os recursos de infraestrutura são descritos por meio de *resources*, os quais são blocos de configuração da infraestrutura que será gerenciada. Para permitir que tais recursos sejam agrupados em unidades coesas e possibilitar uma melhor organização de código, os *resources* podem pertencer a unidades chamadas *modules*, análogos às funções de linguagens de programação.

Semelhante às funções, os *modules* possuem entradas e saídas, que no contexto do Terraform recebem o nome de *variables* e *output values*, respectivamente. As *variables* permitem que *modules* recebam dados necessários para a configuração de uma infraestrutura, sendo que, em contrapartida, os *output values* permitem ao consumidor de um *module* acessar detalhes da infraestrutura provisionada.

Outro conceito muito importante são os *data sources*, recursos destinados a retornar informações dinamicamente de um *provider*, para tais informações poderem ser utilizadas na definição da infraestrutura.

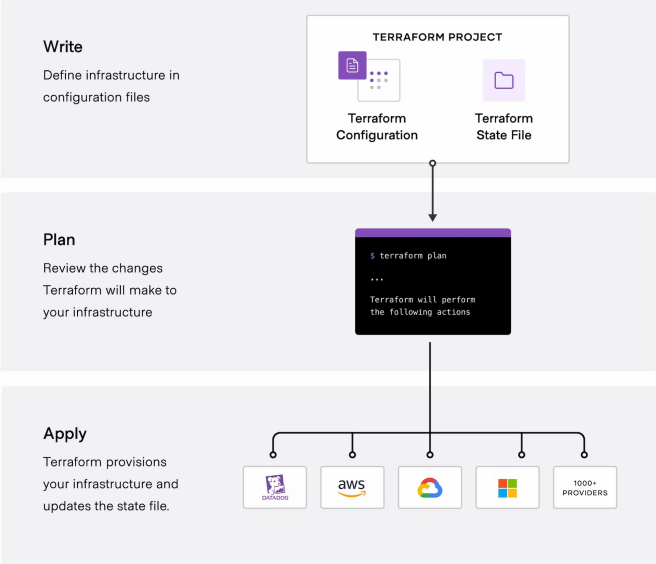
4.4.3 Fluxo de trabalho

A figura 4.7 ilustra o fluxo de trabalho para aplicar mudanças na infraestrutura através do Terraform. Inicialmente, define-se a configuração de infraestrutura via arquivos de configuração, interpretados pela ferramenta. Em seguida, gera-se um *plan* com todas

as mudanças necessárias para que a infraestrutura gerenciada esteja em sincronia com as definições presentes nos arquivos de configuração. Finalmente, após o usuário revisar e concordar com o plano de mudanças, é realizado um *apply*, para que as ações presentes no *plan* sejam de fato aplicadas à infraestrutura.

Existe uma correspondência direta entre o fluxo de trabalho e comandos da ferramenta. De modo a gerar um plano com as mudanças necessárias para sincronizar a infraestrutura e configuração desejada usamos o comando `terraform plan`. De maneira a aplicar as mudanças à infraestrutura, usamos o comando `terraform apply`.

Figura 4.7: Fluxo de trabalho ao aplicar mudanças na infraestrutura através do Terraform.



Fonte: (HASHICORP, 2021)

5 METODOLOGIA E DESENVOLVIMENTO

Este capítulo aborda os detalhes de desenvolvimento e implementação da solução para *deploy* de modelos de classificação de retinopatia diabética na *Cloud*. Não obstante, neste capítulo também é abordado a metodologia seguida para realizar os experimentos relativos aos dois serviços utilizados, *serverless inference e real-time inference*.

Na seção 5.1 abordam-se aspectos relacionados ao desenvolvimento e implementação da solução, como: estrutura de arquivos do projeto, extração e análise de dados, bem como fluxo de *deploy* dos modelos usados no trabalho. Finalmente, na seção 5.2 descreve-se a metodologia empregada para a realização dos experimentos, além dos cenários de experimento executados para cada serviço de inferência avaliado.

5.1 Desenvolvimento da solução

Primeiramente, foi necessário treinar e selecionar modelos para a detecção de retinopatia diabética. Para essa finalidade, utilizou-se como base um *notebook* da plataforma Kaggle[1], onde foram treinadas 27 arquiteturas de rede neural pré-treinadas. O dataset utilizado para treinamento é derivado do *dataset* da competição APTOS 2019, disponível em Karthik Maggie (2019). A distribuição de classes do *dataset* derivado é idêntica ao *dataset* original da competição, sendo mostrada na figura 5.1.

É importante notar que todas as imagens foram redimensionadas para a resolução 224x224, para que fossem usadas com diferentes modelos de *deep learning* pré-treinados. Além disso, aplicou-se um filtro gaussiano às imagens, de maneira a reduzir o ruído. Conforme demonstrado na figura 5.1, as amostras são classificadas segundo 5 graus de RD: sem RD, leve, moderada, grave e proliferativa.

Conforme mostrado em Ishakian, Muthusamy and Slominski (2018), o tamanho do modelo tem grande impacto tanto no tempo de inferência como latência de *cold-start*. Portanto, a fim de verificar o comportamento de modelos de diferentes tamanhos frente a diferentes arquiteturas e serviços de inferência, escolheram-se os modelos VGG19 e MobileNet. A relação entre os modelos, tamanho de artefato em disco e número de parâmetros é apresentado na tabela 5.1.

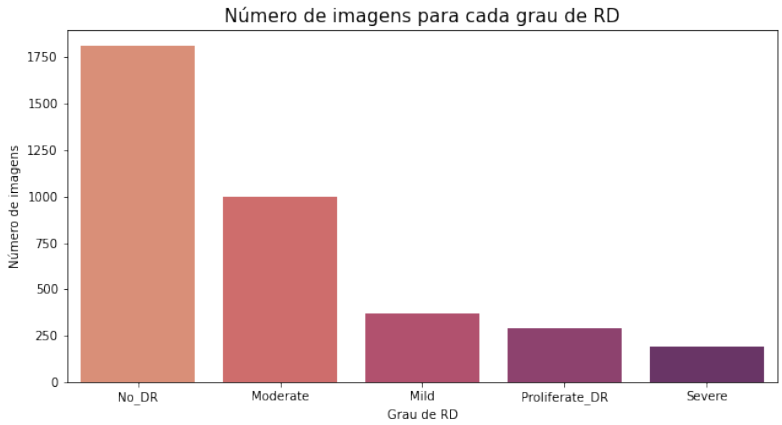
Após o treinamento dos modelos, realizou-se o salvamento dos mesmos seguindo o formato SavedModel, descrito na seção 4.2.1. Para os modelos poderem então ser

[1]Link para o *Notebook*: <https://www.kaggle.com/code/databeru/predict-diabetic-retinopathy-with-dl>

Figura 5.1: Amostras de imagens do *dataset* utilizado para o treinamento dos modelos. As imagens de fundo de olho são classificadas em 5 graus de RD: sem RD, leve, moderada, grave e proliferativa.

Fonte: Elaborado pelo autor.

Figura 5.2: Distribuição de classes do *dataset* utilizado para treinamento dos modelos.



Fonte: Elaborado pelo autor.

utilizados nos serviços Sagemaker *serverless inference* e *real-time inference*, realizou-se a compactação do modelo seguindo o formato tar.gz e fez-se upload para um bucket do S3, consoante ao descrito na subseção 4.3.3.

Tendo os artefatos do modelo hospedados em um *bucket* S3, foi então necessário desenvolver um código que realiza o carregamento do modelo e a predição de classificação de retinopatia diabética a partir de uma imagem. Não obstante, visto o alto tempo empregado para o carregamento dos modelos, foi definida uma classe *Singleton* responsável por fazer o carregamento do modelo e predições de inferência, para o modelo ser

Tabela 5.1: Relação entre modelo utilizado no trabalho, número de parâmetros e tamanho do artefato de modelo (SavedModel) armazenado em disco. Nota-se que o modelo VGG19 é cerca de $6\times$ maior que o modelo MobileNet.

Modelo	**Número de parâmetros**	**Tamanho do artefato (SavedModel)**
MobileNet	3.377.221	15,7 MB
VGG19	20.107.205	77,7 MB

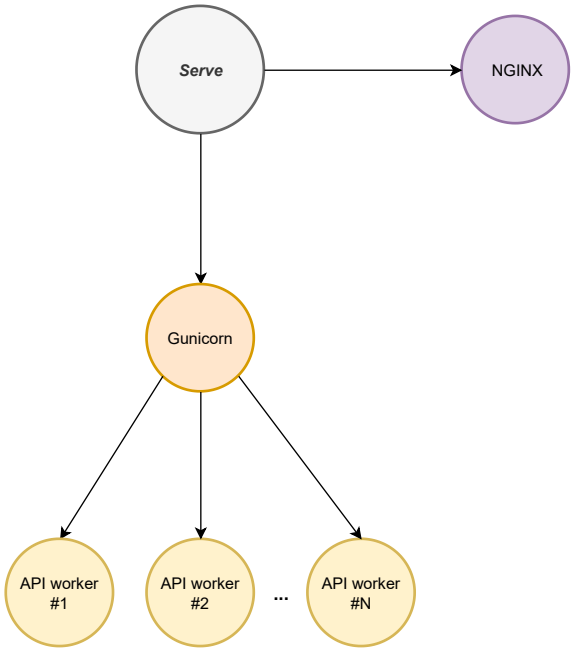
mantido em memória e reutilizado entre diferentes requisições de inferência.

A fim de satisfazer a interface do *container* esperada pelos serviços *real-time inference* e *serverless inference*, assim como mencionado na seção 4.3.3.3, foi necessário implementar uma API HTTP, utilizando para isso a biblioteca FastAPI. Esse API é implementada no arquivo web_application.py, conforme representado na figura 5.4.

Finalmente, o *entrypoint* do container foi definido em um arquivo chamado `serve`, cujo objetivo é iniciar subprocessos para o serviço NGINX e serviço Gunicorn, que, no que lhe concerne, cria e gerencia os subprocessos dos *workers* da aplicação, que rodam múltiplas instâncias de API HTTP destinada a realizar a inferência do modelo.

Após o início dos processos que servirão as requisições de inferência do Sagemaker, o processo *serve* continua esperando até o término de seus processos filhos, tendo também a responsabilidade de matar os processos do NGINX e o processo Gunicorn caso receba um sinal SIGTERM indicando que o *container* será destruído. Um diagrama mostrando a criação de subprocessos durante a inicialização do *container* é apresentado na figura 5.3.

Figura 5.3: Diagrama de criação de subprocessos durante a inicialização do *container*. O processo `serve` é o processo principal, responsável pela criação do subprocesso NGINX e subprocesso Gunicorn. Então, o subprocesso Gunicorn cria N *workers*, a depender da quantidade de vCPUs da instância utilizada para executar os *containers*.



Fonte: Elaborado pelo autor

5.1.1 Estrutura de arquivos do projeto

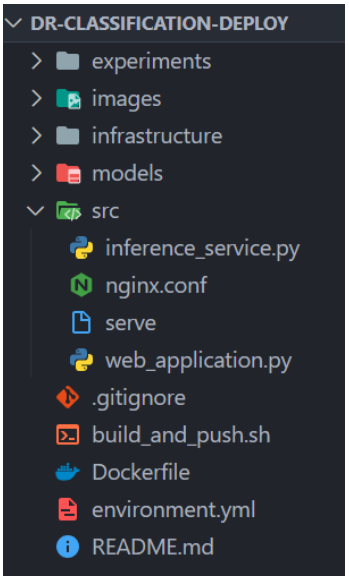
A estrutura de arquivos do projeto é mostrada na figura 5.4. A seguir, detalha-se a estrutura de diretórios em conjunto com sua finalidade e os arquivos-chave para o projeto:

- `experiments`: Neste diretório estão presentes os *scripts* destinados a executar e recolher os dados dos experimentos, armazenados na forma de arquivos CSV simples. Neste mesmo diretório, também foram definidos *scripts* para analisar os CSVs e gerar os gráficos utilizados nesse trabalho.
- `images`: Neste diretório são armazenadas imagens do *dataset* APTOS, utilizadas para enviar requisições de inferência aos *endpoints* testados neste trabalho.
- `infrastructure`: Neste diretório estão presentes os arquivos Terraform que descrevem o código necessário para provisionamento da infraestrutura analisada neste trabalho.
- `models`: Neste diretório são armazenados os artefatos dos modelos treinados para a classificação de RD.
- `src`: Neste diretório é mantido o código-fonte da aplicação, destinado a fazer a inferência a partir dos modelos, bem como inicializar os processos necessários para o funcionamento do *container* que serve as inferências a partir da API HTTP consumida pelo Sagemaker.
    - `inference_service.py`: Neste arquivo são definidas as funções responsáveis pelo carregamento do modelo e a realização da inferência dos modelos. As funções são agrupadas em uma classe *singleton*, chamada *InferenceService*.
    - `nginx.conf`: Neste arquivo são definidas as configurações necessárias para o funcionamento do NGINX. Por exemplo, neste arquivo é definido um proxy das requisições `GET /ping` e `GET /invocations` para o subprocesso Gunicorn.
    - `serve`: Neste arquivo é definido o código Python executado no momento em que o *container* é inicializado, sendo responsável por inicializar os subprocessos necessários para o funcionamento do *container*.
    - `web_application.py`: Neste arquivo é definido o código Python que descreve a API destinada a implementar a interface esperada pelo serviço Sagemaker, conforme mencionado na seção 4.3.3. Para isso, foi utilizado a

biblioteca FastAPI.

- `Dockerfile`: Neste arquivo é definida a imagem *docker* do *container*, conforme mencionado na seção 2.3.
- `environment.yml`: Neste arquivo são definidas as dependências do projeto em um formato reconhecido pelo Conda. Como exemplo de dependências incluídas nesse arquivo estão: Tensorflow, FastAPI, Gunicorn, Uvicorn e Pillow.

Figura 5.4: Estrutura de arquivos do projeto



Fonte: Elaborado pelo autor

5.1.2 Fluxo para *deploy* de endpoints

Caso sejam efetuadas atualizações na versão do *container*, contendo melhorias ou correções para o código que realiza a inferência dos modelos, é necessário seguir uma série de passos para disponibilizar tais mudanças em um novo *endpoint*, são estes:

1. *Build* da imagem Docker: Primeiramente, é necessário realizar o build da imagem contendo as novas modificações, para isso, utiliza-se o comando `docker image build`.
2. Upload da imagem Docker para o ECR: Na sequência, é necessário realizar o upload da imagem para o repositório de imagens Docker na nuvem. Com esse intuito, utilizam-se os comandos `docker login` e `docker push`.
3. Criação de um novo endpoint: É necessário criar um novo endpoint, para ser utilizada a nova versão de imagem de *container*, recém criada. Para tal fim, adicionam-

se novos *resources* ao terraform, seguido do comando `terraform apply`.

Mesmo que já existam *endpoints* ativos no *Sagemaker* apontando para uma certa imagem Docker, ao realizar atualizações no repositório de imagens é necessário destruir e recriar tais *endpoints*. Isso acontece, pois um endpoint *Sagemaker* captura a versão da imagem Docker no momento de sua criação, sendo um endpoint imutável, portanto, evitando mudanças não propositais.

Caso sejam efetuadas mudanças no artefato dos modelos, descritos em 4.2.1, é necessário seguir os seguintes passos:

1. *Upload* do novo artefato para o S3: Primeiramente, é necessário realizar o *upload* do artefato do modelo atualizado para o S3. Com essa finalidade, utiliza-se o comando `s3 sync`.
2. Criação de um novo *endpoint*: É necessário criar um novo *endpoint*, para serem utilizados os novos artefatos de modelo, recém-adicionados ao *bucket* S3. Para tal fim, adicionam-se novos *resources* ao terraform, seguido do comando `terraform apply`.

Assim como para mudanças na imagem Docker, ao realizar mudanças nos artefatos do modelo também é necessário recriar os endpoints, consoante o descrito em Hudgeon and Nichol (2020b).

5.1.3 Extração e análise de dados

A fim de analisar as diferentes otimizações e configurações utilizadas neste trabalho, foi necessário extrair métricas relativas à latência das soluções estudadas. Para atingir esse objetivo, o código utiliza a biblioteca padrão do Python `time`, para medir o tempo de execução da inferência dos modelos e retornar esse tempo como metadado de resposta da requisição. A fim de medir a latência total do endpoint, a mesma biblioteca foi utilizada, mas nos *scripts* destinados à execução dos experimentos.

Além disso, também foi utilizado o serviço CloudWatch, para a extração de métricas relativas à execução dos endpoints. Em especial, para o serviço *serverless inference*, foi utilizado a métrica *ModelSetupTime*, que diz respeito ao tempo despendido para inicializar os recursos computacionais necessários para atender às requisições de um *endpoint*.

5.2 Experimentos realizados

Dado as diferenças entre os serviços *serverless inference* e *real-time inference*, conforme discutido na seção 4.3.3.1, separaram-se os experimentos realizados em duas categorias, de tal maneira que cada experimento executado reflita as características do serviço avaliado. Por exemplo, ao avaliar inferências executadas no serviço *real-time inference* não foi utilizado nenhum cenário com intervalo maior que 10 segundos entre requisições, já que o caso *cold-start* não se aplica como para o serviço *serverless inference*. A seguir, detalham-se os experimentos realizados para cada tipo de serviço.

5.2.1 Experimentos *Serverless inference*

Como para o serviço *serverless inference* existem dois casos de latência: *warm-up* e *cold-start*, foi necessário definir dois cenários de experimento base:

- Cenário *warm-up*: Primeiramente, é realizado uma requisição sem que a mesma seja contabilizada, para que o *container* seja inicializado e as demais requisições não sofram da latência *cold-start*. Na sequência, realiza-se 25 requisições em série, com intervalos de 5 segundos entre cada requisição.
- Cenário *cold-start*: Realizam-se 10 requisições sequenciais, com intervalos de 15 minutos entre cada requisição, para que os *containers* sejam destruídos entre as execuções, ocasionando o caso de latência *cold-start* que desejamos medir.

Para ambos os cenários, extraíram-se duas métricas relativas ao tempo de execução da aplicação: latência total e tempo de predição. A latência total é a medida de tempo levado entre o envio da requisição e a resposta do servidor, englobando tanto o tempo para envio da imagem, predição do modelo e resposta do servidor. Já o tempo de predição mede apenas o tempo levado para realizar a inferência de uma imagem.

De posse dos cenários de experimento base, duas linhas de pesquisa foram seguidas: experimentação com o tamanho da imagem Docker e experimentação com a quantidade de memória disponível para o *endpoint*.

Tabela 5.2: Tabela de diferentes otimizações aplicadas à imagem base e o respectivo tamanho da imagem em disco (sem estar comprimida) e tamanho da imagem comprimida (tamanho ocupado pela imagem em registros, como ECR).

Otimização	**Tamanho imagem comprimida**	**Tamanho imagem em disco**
Nenhuma	1798.27 MB	4.97 GB
Two-step build	923.99 MB	3.32 GB
Redução dependências	517.23 MB	2.06 GB

Tabela 5.3: Cenários de experimento executados para cada combinação de imagem Docker otimizada e modelo avaliado, no total foram executados 6 cenários de experimentos.

Modelo	**Imagem 4.97 GB**	**Imagem 3.32 GB**	**Imagem 2.06 GB**
MobileNet	Cenário *cold-start*	Cenário *cold-start*	Cenário *cold-start*
VGG19	Cenário *cold-start*	Cenário *cold-start*	Cenário *cold-start*

5.2.1.1 Tamanho de imagem docker

A fim de verificar o impacto do tamanho da imagem docker na latência *cold-start* da aplicação, foram aplicadas duas otimizações. A primeira otimização aplicada foi a refatoração do `dockerfile` para utilizar o conceito de *two-step-build*, conforme comentado na seção 2.3.1.2. Na sequência, foi feito uma redução das dependências, eliminando dependências não essenciais ao funcionamento da aplicação e otimizando o tamanho de dependências existentes, como a troca da dependência tensorflow-mkl pela dependência Tensorflow. As otimizações e respectivos impactos no tamanho da imagem Docker resultante é apresentada na tabela 5.2.

Uma vez que o *container* da aplicação já está executando, o tempo de inicialização do mesmo não influência na latência da aplicação, visto que o *container* só é destruído após algum tempo de ociosidade. Por este motivo, ao avaliar o impacto das otimizações de tamanho de imagem na latência, apenas o experimento base para o cenário *cold-start* foi considerado. A tabela 5.3 sumariza os cenários de experimento executados para cada combinação de modelo e imagem *Docker* otimizada, totalizando 6 execuções de experimento.

5.2.1.2 Memória do endpoint

A fim de verificar o impacto da configuração de tamanho de memória do endpoint no tempo de predição e latência total da aplicação, foram utilizados os seguintes valores de memória: `1024 MB`, `2048 MB` e `3072 MB`. Todas as configurações de tamanho de memória disponíveis para o serviço *serverless inference* e respectivos custos são mostrados na tabela 4.1.

Tabela 5.4: Cenários de experimento executados para cada combinação de configuração de memória e modelo avaliado, no total foram executados 12 cenários de experimentos.

Modelo	**Memória: 1 GB**	**Memória: 2 GB**	**Memória: 3 GB**
MobileNet	cold-start, warm-up	cold-start, warm-up	cold-start, warm-up
VGG19	cold-start, warm-up	cold-start, warm-up	cold-start, warm-up

Tabela 5.5: Máquinas utilizadas para a execução da inferência a partir do serviço *sagemaker real-time-inference*.

Tipo de instância	**ISA**	**Processador**
`ml.m4`	x86-64	Intel Xeon E5-2676 v3
`ml.m5`	x86-64	Intel Xeon Platinum
`ml.m6g`	ARMv8.2-A	Graviton 2
`ml.c7g`	ARMv8.4-A	Graviton 3

Segundo Ishakian, Muthusamy and Slominski (2018), o aumento da configuração de memória de uma função lâmbda reflete tanto no caso *cold-start* como no *warm-up*. Por este motivo, avaliou-se tanto o cenário de experimento *cold-start*, como o cenário de experimento *warm-up*, para verificar se o mesmo comportamento se mantém para o serviço *serverless inference*. Conforme sumarizado na tabela 5.4, ao todo foram executados 12 cenários de experimento.

5.2.2 Experimentos *Real-time inference*

Como no serviço *real-time inference* são utilizadas instâncias dedicadas para a execução dos *containers*, a aplicação deixa de sofrer com o caso dos *cold-starts* descritos em Ishakian, Muthusamy and Slominski (2018). Portanto, realizou-se apenas um experimento simples composto de 25 requisições em sequência, todas contabilizadas, com intervalos de 5 segundos entre as requisições. Assim como nos experimentos para o serviço *serverless inference*, utilizaram-se apenas as métricas de tempo de predição e latência total.

A fim de avaliar diferentes arquiteturas para a realização da inferência, utilizaram-se diferentes tipos de instância, sumarizadas na tabela 5.5. Ao todo, foram avaliadas quatro instâncias, tendo duas a *ISA* x86-64 e outras duas a *ISA* ARMv8.

Visto que *Docker images* destinadas à plataforma Linux/x86-64 não podem ser executadas em plataformas ARM, foi necessário adaptar a imagem original, utilizada em outros experimentos, para que incluísse dependências compatíveis com a plataforma Linux/ARM64 e também utilizasse imagens-base compatíveis com essa plataforma. Em

especial, foi utilizado a versão do tensorflow tensorflow-cpu-aws, otimizada para os processadores Graviton da AWS, disponível em Inc (2022).

6 RESULTADOS

Neste Capítulo estão apresentados os resultados obtidos a partir da utilização dos serviços *real-time inference* e *serverless inference*, baseados nos experimentos mencionados na seção 5.2. Em conjunto com os resultados, serão realizadas análises de diferentes alternativas de configuração e otimização para cada serviço analisado, considerando o impacto no desempenho e custo financeiro dos serviços.

Na seção 6.1 são apresentados os resultados para o serviço *serverless inference*, onde serão analisados o impacto em termos de desempenho e custo financeiro de duas otimizações: otimização do tamanho de imagem *Docker* e otimização da configuração de memória dos *endpoints*. Finalmente, na seção 6.2 são apresentados os resultados para o serviço *real-time inference*, onde serão analisados o impacto da utilização de diferentes processadores para a execução da inferência em termos de desempenho e custo financeiro.

6.1 Serverless inference

A primeira otimização aplicada durante a utilização do serviço *serverless inference* foi a otimização do tamanho da imagem Docker. A redução do tamanho da imagem foi essencial para o teste de demais otimizações e versões de algoritmos utilizados no trabalho, visto que uma imagem menor acarreta menor latência de upload para registros de imagem e menor tempo de *build*. Interessantemente, a otimização de imagem Docker trouxe também redução do caso de latência *cold-start* para ambos os modelos analisados, conforme detalhado na seção 6.1.1.

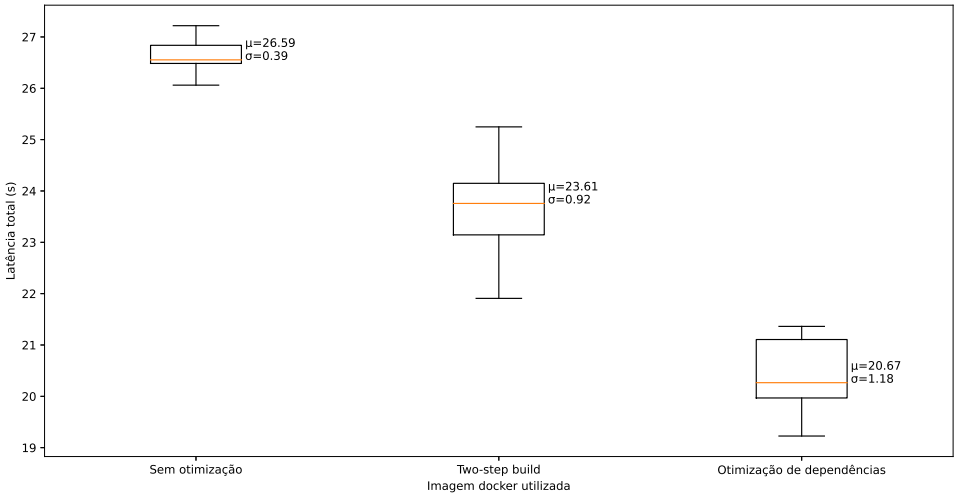
Ainda relacionado ao serviço *serverless inference*, a otimização de configuração de tamanho de memória dos endpoints trouxe ganhos importantes tanto em termos de desempenho como custo financeiro para a execução dos modelos analisados, como será detalhado na seção 6.1.2.

6.1.1 Otimização de imagem docker

Assim como mencionado na seção 5.2.1.1, o caso *cold-start* não contribui para o aumento do tempo de predição dos modelos, portanto a única métrica fundamental a ser analisada é o tempo de latência total de uma requisição.

Analisando as figuras 6.1 e 6.2 nota-se que a otimização do tamanho da imagem Docker trouxe aumento de desempenho para ambos os modelos. Comparando o tempo de latência total da maior imagem Docker (sem otimização - 1798.27 MB), com a menor imagem Docker (otimização de dependências - 517.23 MB) houve um *speed-up* de $1,2864\times$ para o modelo MobileNet e um *speed-up* de $1,3272\times$ para o modelo VGG19.

Figura 6.1: Tempo de latência total para o modelo MobileNet quando executado através do serviço serverless inference, a partir de diferentes imagens Docker. Nota-se que o tempo de latência total é reduzido a medida que são aplicadas as otimizações na imagem docker.



Fonte: Elaborado pelo autor

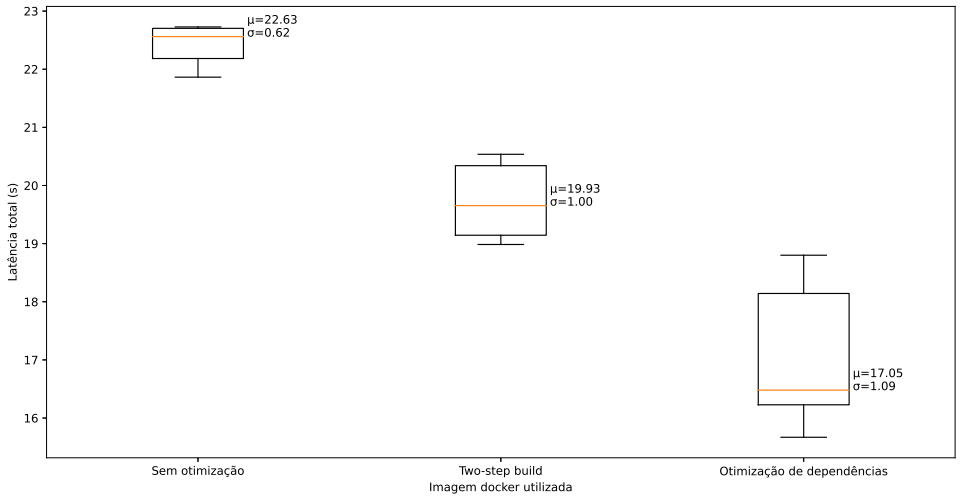
Não limitada à questão de tempo de latência *cold-start*, a otimização do tamanho da imagem Docker é essencial para a redução de tempo para *deploy*, *build* e também para a redução de custo financeiro para o armazenamento em registros de imagem Docker, assim como o serviço ECR utilizado nesse trabalho.

6.1.2 Tamanho de memória do *endpoint*

Consoante ao mencionado em Ishakian, Muthusamy and Slominski (2018), a medida que a configuração de tamanho de memória dos endpoints aumenta, percebe-se uma diminuição tanto da latência total como para o tempo de predição dos modelos.

A seguir, analisa-se o impacto do aumento de memória tanto para o cenário *warm-up* como para o cenário *cold-start*, ambos descritos na seção 5.2.

Figura 6.2: Tempo de latência total para o modelo VGG19 quando executado através do serviço *serverless inference*, a partir de diferentes imagens Docker. Nota-se que o tempo de latência total é reduzido a medida que são aplicadas as otimizações na imagem docker.



Fonte: Elaborado pelo autor

6.1.2.1 Cenário warm-up

Analisando os gráficos presentes nas figuras 6.3 e 6.4, percebemos que houve uma grande redução do tempo de predição e latência total para o modelo VGG19. Em especial, quando comparamos a configuração de 2048 MB de memória com a configuração de 1024 MB de memória, houve um *speed-up* de $2,7413\times$ analisando o tempo de predição.

Um caso muito interessante é notado quando se avalia o desempenho e despesa das diferentes configurações de memória para o modelo VGG19. Conforme os dados da tabela 6.1, o menor custo financeiro de endpoint é atingido com 2 GB de memória, mesmo que o custo por segundo desse endpoint seja o dobro da configuração de endpoint de 1 GB. Como o endpoint de 2 GB é $2,7413\times$ mais rápido que o endpoint de 1 GB, chegamos a um caso onde há tanto redução de custo financeiro como melhoria de desempenho, já que o custo financeiro de execução é diretamente proporcional ao tempo de execução. Portanto, concluímos que nem sempre um aumento na configuração de memória implica em um aumento do custo por inferência de endpoint.

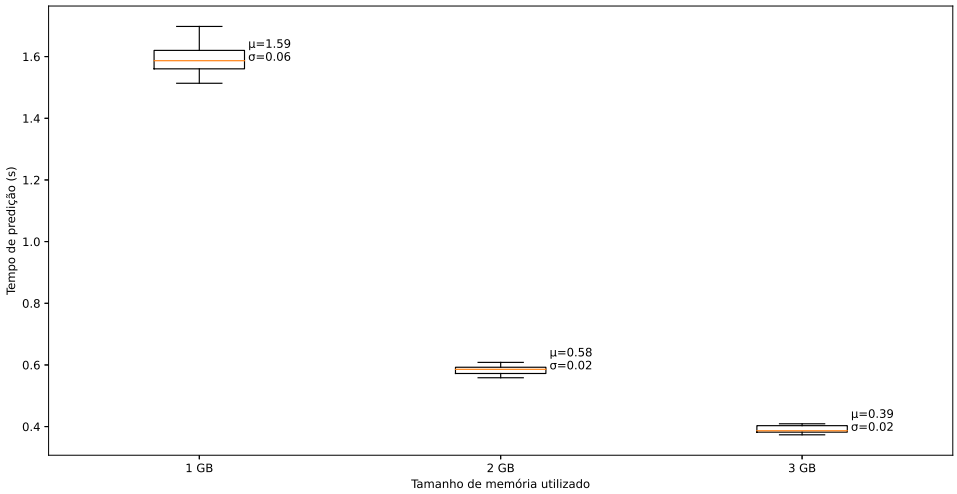
Através da seleção da configuração de memória de 2 GB para o modelo VGG19, conseguimos obter uma redução de custo financeiro de $9,4\%$, aliado ao aumento de desempenho já mencionado previamente.

Analisando os gráficos 6.5 e 6.6, notamos uma diferença de comportamento entre os modelos VGG19 e MobileNet. Considerando o tempo de predição do modelo Mobilenet, houve um aumento de desempenho considerável entre os endpoints de 1 GB e 2 GB,

Tabela 6.1: Custo médio por inferência para o modelo VGG19 e diferentes configurações de tamanho de endpoint.

Tamanho de memória do endpoint	Custo médio por inferência
1 GB	0.0000376 USD
2 GB	0.0000344 USD
3 GB	0.0000408 USD

Figura 6.3: Tempo de predição para o modelo VGG19 quando executado através do serviço *serverless inference* por tamanho de memória utilizado. Nota-se que conforme a memória aumenta, o tempo de predição diminui.



Fonte: Elaborado pelo autor

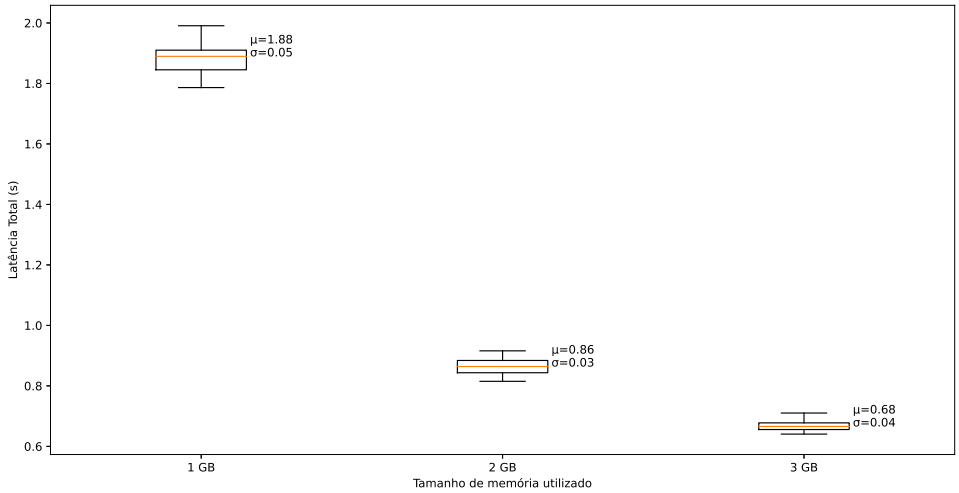
totalizando um *speed-up* de $2,3500\times$. No entanto, quando analisamos a latência total do *endpoint*, o *speed-up* diminui para $1,3807\times$.

Visto que o tempo de predição do modelo MobileNet é pequeno comparado à latência total, representando apenas 37% da latência total, os ganhos de desempenho do aumento de memória são menores quando comparamos com modelos maiores, como o VGG19. Devido a esse fato, para o modelo MobileNet, todo aumento de memória implicou em um aumento do custo financeiro, conforme mostrado na tabela 6.2.

Tabela 6.2: Custo médio por inferência para o modelo MobileNet e diferentes configurações de tamanho de endpoint.

Tamanho de memória do endpoint	Custo médio por inferência
1 GB	0.0000095 USD
2 GB	0.0000144 USD
3 GB	0.0000206 USD

Figura 6.4: Latência total para o modelo VGG19 quando executado através do serviço *serverless inference* por tamanho de memória utilizado. Nota-se que conforme a memória aumenta, a latência total diminui.



Fonte: Elaborado pelo autor

6.1.2.2 Cenário cold-start

Semelhante aos resultados do cenário *warm-up*, também houve ganhos de desempenho na latência total do *endpoint* no cenário *cold-start* conforme a memória do *endpoint* usado aumentou.

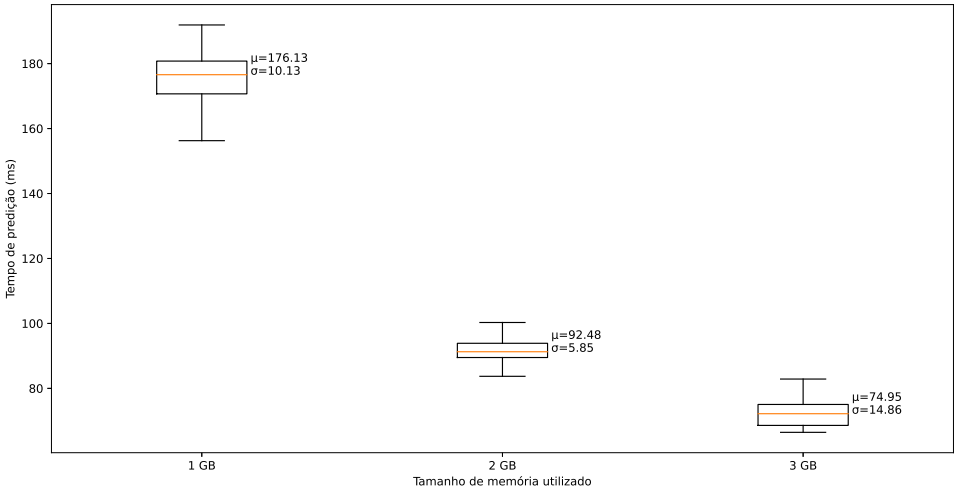
Conforme mostrado nas figuras 6.7 e 6.8, para ambos os modelos VGG19 e MobileNet houve uma grande redução na latência de *cold-start* quando comparamos as configurações de memória de 1 GB e 2 GB. Para o modelo MobileNet, houve um *speed-up* de $1,6529\times$ entre a configuração de 1 GB e 2 GB, sendo que para o modelo VGG19 o speedup foi de $1,54019\times$.

Diferentemente do cenário *warm-up*, onde a redução de latência total foi mais expressiva para o modelo VGG19, no cenário *cold-start* ambos os modelos apresentaram ganhos de desempenho semelhantes.

Conforme mencionado previamente, para modelos cujo tempo de predição é uma fração pequena do tempo de latência total, o aumento de memória não conclui em aumento de desempenho. No entanto, para o caso de latência *cold-start* a relação entre tempo de predição e latência total não tem impacto nos ganhos de desempenho.

Para explicar esse fenômeno podemos analisar a métrica *ModelSetupTime*, que diz respeito ao tempo necessário para inicializar os recursos computacionais para a execução dos *endpoints*. Conforme as tabelas 6.3 e 6.4, podemos notar que a medida que o tamanho de memória cresce o *ModelSetupTime* diminui. Visto que o *ModelSetupTime* é um com-

Figura 6.5: Tempo de predição para o modelo MobileNet quando executado através do serviço *serverless inference* por tamanho de memória utilizado. Assim como para o modelo VGG19, a medida que a memória aumenta, o tempo de predição diminui.



Fonte: Elaborado pelo autor

Tabela 6.3: ModelSetupTime médio do modelo VGG19 para cada configuração de memória utilizada. Podemos notar que conforme o tamanho de memória aumenta, o tempo gasto para inicializar os recursos computacionais dos endpoints diminui.

Tamanho de memória do endpoint	***ModelSetupTime***
1 GB	10,7879 s
2 GB	6,8852 s
3 GB	6,5223 s

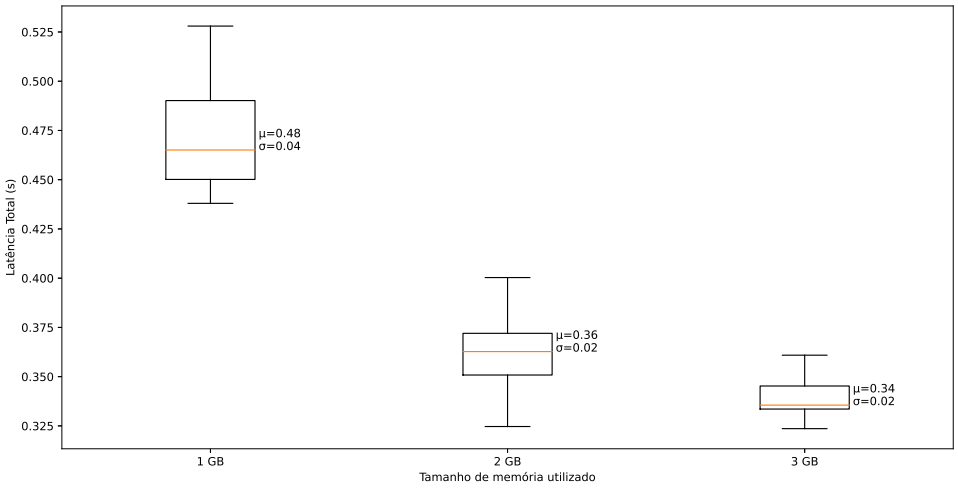
ponente adicional muito importante para latência total de uma requisição, isso explica a redução da latência de *cold-start* para ambos os modelos.

Interessantemente, para ambos os modelos não houve uma diferença considerável entre a configuração de memória `2 GB` e `3 GB`, conforme vemos nas figuras 6.7 e 6.8. Tal observação é consoante ao mostrado em Ishakian, Muthusamy and Slominski (2018), onde a latência de *cold-start* foi afetada pela configuração de memória dos *endpoints lambda*, porém a partir de certo tamanho de memória o aumento da mesma não concluiu em aumento de desempenho.

Tabela 6.4: ModelSetupTime médio do modelo MobileNet para cada configuração de memória utilizada. Podemos notar que conforme o tamanho de memória aumenta, o tempo gasto para inicializar os recursos computacionais dos endpoints diminui.

Tamanho de memória do endpoint	***ModelSetupTime***
1 GB	10,6071 s
2 GB	5,6728 s
3 GB	5,6074 s

Figura 6.6: Latência total para o modelo MobileNet quando executado através do serviço *serverless inference* por tamanho de memória utilizado.



Fonte: Elaborado pelo autor

Tabela 6.5: Latência *cold-start* por modelo utilizado antes e após otimizações. Nota-se que houve um aumento de desempenho de cerca de $2\times$.

Modelo	**Latência sem otimização**	**Latência após otimizações**
VGG19	22,63 s	11,07 s
MobileNet	26,59 s	12,48 s

6.1.3 Comparação latência *cold-start* e *warm-up*

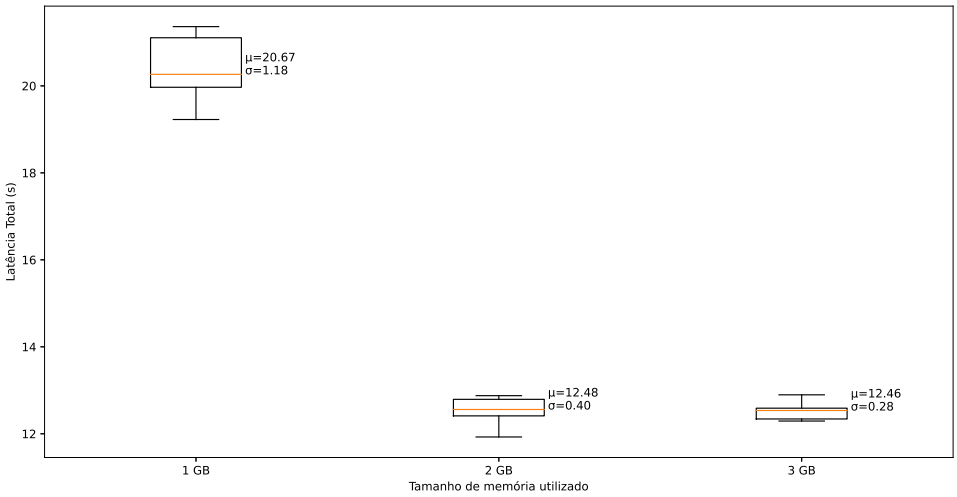
Consoante ao mencionado previamente, o uso de *serverless computing* para o *deploy* de modelos de *deep learning* apresenta latências maiores durante a inicialização dos recursos computacionais dedicados à execução da inferência, caso conhecido como latência *cold-start*.

A partir da combinação das otimizações de tamanho de imagem Docker e configuração de memória de endpoint, foi possível reduzir a latência *cold-start* em cerca de $2\times$ para ambos os modelos utilizados, conforme mostra a tabela 6.5. Para o modelo VGG19 houve um aumento de desempenho de $2,04\times$ e para o modelo MobileNet houve um aumento de desempenho de $2,13\times$.

Mesmo após a aplicação das otimizações de maneira combinada, notamos uma grande diferença entre a latência do caso *warm-up* e caso *cold-start*. Comparando a latência *warm-up* com a latência *cold-start* dos endpoints com configuração de 2 GB, a latência *cold-start* é $57,42\times$ maior para o modelo MobileNet e $19,82\times$ maior para o modelo VGG19.

Essa grande diferença pode ser impeditiva para soluções que possuam restrições

Figura 6.7: Latência total de *cold-start* para o modelo MobileNet quando executado através do serviço *serverless inference* por tamanho de memória utilizado.



Fonte: Elaborado pelo autor

quanto ao desempenho. Nesses casos, existem alternativas a serem exploradas para minimizar o impacto da latência de *cold-start*, como o uso de requisições para realizar o *warm-up* dos *containers* durante os períodos de uso da aplicação. Técnicas mais avançadas podem ser utilizadas, como o uso de previsão de séries temporais para a determinação dos períodos de uso de uma aplicação e a pré-alocação adequada dos recursos, conforme estudado em Jegannathan, Saha and Addya (2022).

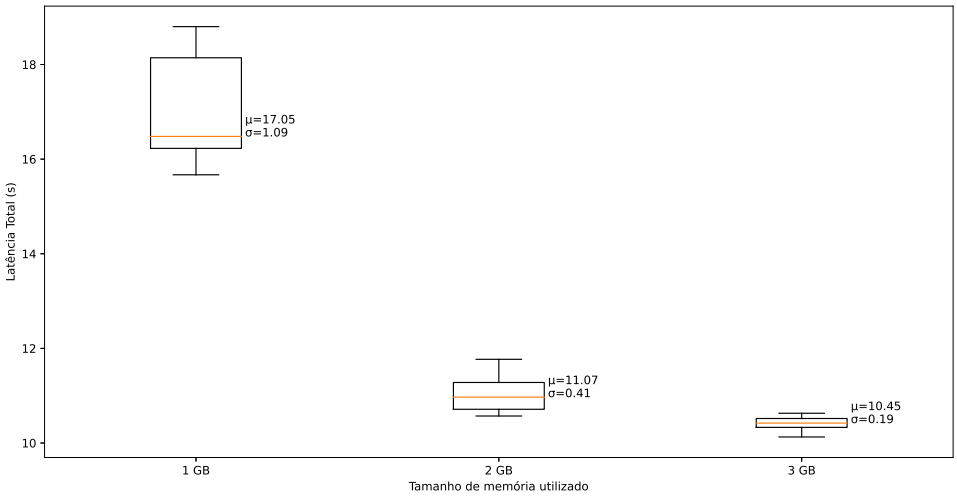
6.2 Real-time inference

O serviço *real-time inference* oferece diferentes arquiteturas e processadores para a execução da inferência dos modelos. Conforme mencionado na seção 5.2.2, neste trabalho foram utilizadas duas arquiteturas distintas para análise: x86-64 e ARM.

Foi observado que o tipo de instância utilizado tem impacto no tempo de predição em ambos os modelos, MobileNet e VGG19. No entanto, quando analisamos a latência total do *endpoint* nota-se um menor impacto para o modelo menor, MobileNet. Por esse motivo, o impacto da utilização de diferentes tipos de instância é analisado separadamente para cada métrica, sendo o tempo de predição analisado na seção 6.2.1 e a latência total analisada na seção 6.2.2.

Interessantemente, o melhor desempenho foi alcançado a partir da utilização de um processador com arquitetura x86-64. No entanto, a utilização da arquitetura ARM traz consigo vantagens de custo financeiro, principalmente em modelos cuja razão entre

Figura 6.8: Latência total de *cold-start* para o modelo VGG19 quando executado através do serviço *serverless inference* por tamanho de memória utilizado.



Fonte: Elaborado pelo autor

tempo de predição e latência total é menor, como será detalhado nas seções a seguir.

6.2.1 Tempo de predição

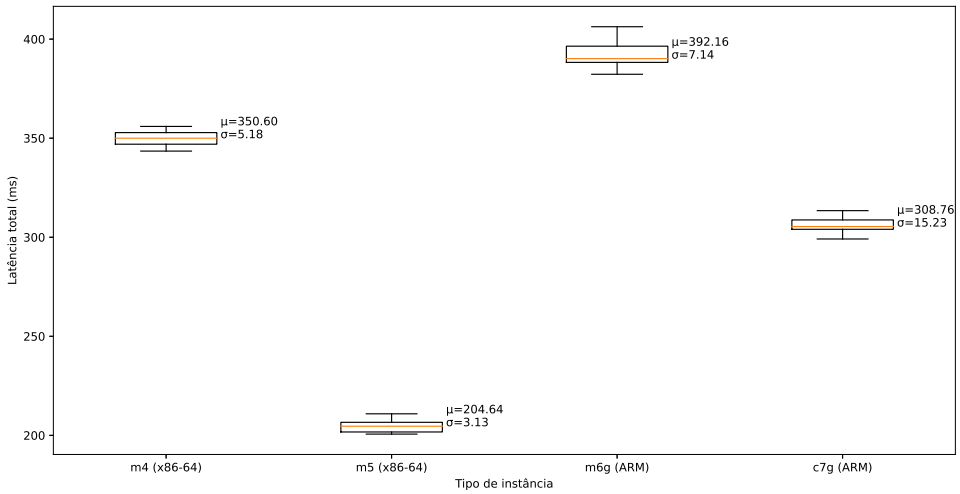
Conforme mostrado nos gráficos 6.9 e 6.10, há uma diferença considerável de desempenho entre diferentes tipos de máquina, mesmo que todas as instâncias tenham a mesma quantidade de vCPUs, conforme a tabela 6.6. Considerando o tipo de instância que apresenta o melhor desempenho, *m5*, com a máquina de pior desempenho, *m6g*, houve um *speed-up* de $1,9163\times$ para o modelo VGG19 e um *speed-up* de $1,42009\times$ para o modelo MobileNet.

Tabela 6.6: Relação de memória, número de vCPUs, ISA e custo por hora para cada tipo de instância utilizada no serviço Sagemaker *real-time inference*. Nota-se que as instâncias estão ordenadas de maneira crescente por geração. Percebe-se que o custo financeiro sempre decresce conforme há um avanço na geração da instância.

Tipo de instância	**ISA**	**Memória**	**vCPUs**	**Custo por hora**
`ml.m4.xlarge`	x86-64	16 GB	4	0,24 USD
`ml.m5.xlarge`	x86-64	16 GB	4	0,23 USD
`ml.m6g.xlarge`	ARM	16 GB	4	0,1848 USD
`ml.c7g.xlarge`	ARM	8 GB	4	0,174 USD

Podemos notar que dentro de cada arquitetura as máquinas de geração mais atual apresentaram o melhor desempenho. Para o caso da arquitetura x86-64, comparando a instância `m5` frente à instância `m4`, houve um *speed-up* de $1,7134\times$ para o modelo

Figura 6.9: Tempo de predição para o modelo VGG19 quando executado através do serviço *real-time inference* por tipo de instância utilizado. Nota-se que o tipo de instância com menor latência pertence à arquitetura x86-64: a máquina `m5.xlarge`.



Fonte: Elaborado pelo autor

VGG19 e $1,3176\times$ para o modelo MobileNet. De maneira similar para a arquitetura ARM, quando comparamos uma instância `c7g` frente a uma instância `m6g` há um *speed-up* de $1,2701\times$ para o modelo VGG19 e $1,2264\times$ para o modelo MobileNet.

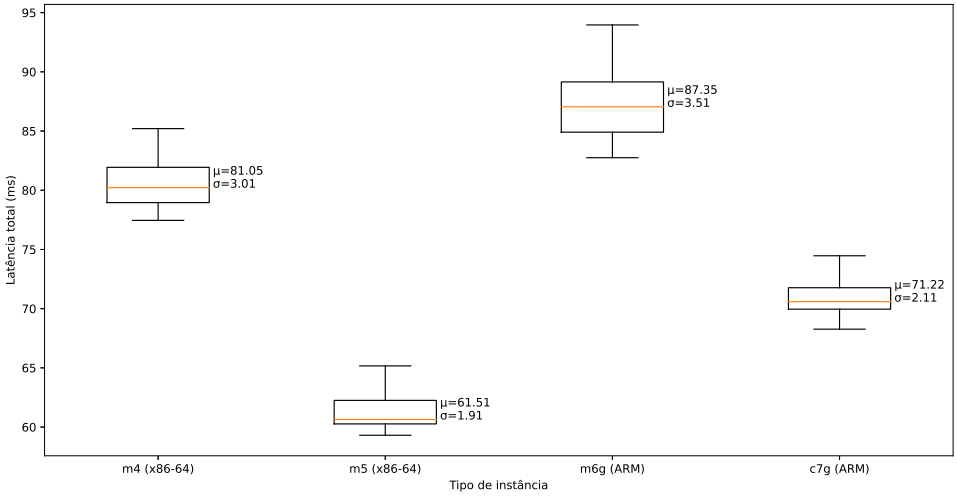
Quando se considera o custo financeiro de cada tipo de instância, expresso na tabela 6.6, percebe-se que sempre é vantajoso utilizar a versão mais recente de cada arquitetura, já que há tanto ganhos de desempenho como também redução de custo. Para o caso da arquitetura x86 existe uma redução de custo de $4,1667\%$ ao utilizar a instância mais recente, e para a arquitetura ARM existe uma redução de custo de $5,8441\%$.

6.2.2 Latência total

Embora ambos os modelos tenham apresentado diferença significativa para o tempo de predição, quando analisamos a latência total de uma requisição percebe-se uma diferença de comportamento entre os modelos VGG19 e MobileNet, como ilustrado nas figuras 6.11 e 6.12.

Para o modelo VGG19 ainda existe uma diferença significativa no tempo total de latência entre o tipo de instância mais rápido, `m5`, e o tipo de instância mais lento, `m4`, totalizando um *speed-up* de $1,5001\times$ entre as instâncias. No entanto, para o modelo MobileNet o *speed-up* entre as duas instâncias é de apenas $1,08972\times$, comparado a um *speed-up* de $1,42009\times$ quando se analisa apenas o tempo de predição.

Figura 6.10: Tempo de predição para o modelo MobileNet quando executado através do serviço *real-time inference* por tipo de instância utilizado. Assim como para o modelo VGG19, a instância mais rápida foi a `m5`.



Fonte: Elaborado pelo autor

Tabela 6.7: Relação entre razão de tempo de predição por latência total e *speed-up* máximo atingido entre diferentes tipos de instância para os modelos analisados. Nota-se que em modelos cujo tempo de predição é muito baixo a utilização de diferentes tipos de instância não se converte em otimização da latência total.

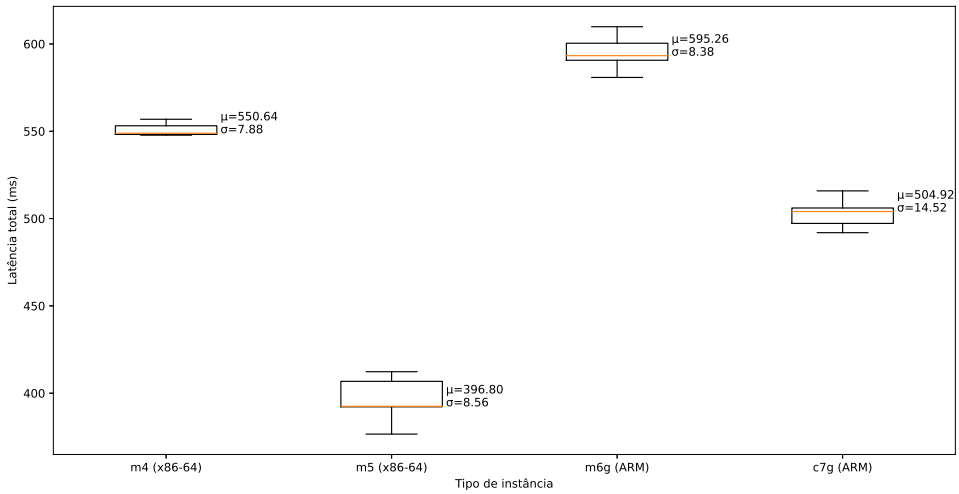
Modelo	**Tempo de predição / Latência total**	**Speed-up máximo**
VGG19	0,51572	1,5001
MobileNet	0,23605940	1,08972

A diferença da influência dos tipos de instância na latência total dos dois modelos acontece, pois o tempo de inferência do modelo MobileNet é menor que no modelo VGG19, portanto influenciando menos a latência total, calculada a partir da soma entre a latência de transferência de dados e o tempo de predição do modelo.

Portanto, conforme mostrado na tabela 6.7, em modelos cuja razão entre tempo de predição e latência total da requisição seja baixa, a utilização de diferentes tipos de instância não concluem em otimização da latência total. Nesses casos, pode ser ainda mais vantajoso utilizar instâncias `c7g`, devido ao baixo custo financeiro e pouca redução de desempenho comparada ao tipo de instância mais rápida, `m6`.

Comparando a instância mais performática (`m5`) com a instância de menor custo financeiro (`c7g`), há uma redução de desempenho de $33,72\%$ ao utilizar a instância de menor custo para a execução do modelo VGG19. Quando efetuamos a mesma comparação para o modelo MobileNet, a redução de desempenho é de $13,63\%$. Indiferente ao modelo utilizado, conseguimos reduzir o custo da execução em $24,34\%$, o que pode ser interessante para soluções com restrições de custo.

Figura 6.11: Tempo de latência total para o modelo VGG19 quando executado através do serviço *real-time inference* por tipo de instância utilizado. Nota-se que o tipo de instância com menor latência pertence à arquitetura x86-64: a máquina `m5.xlarge`.



Fonte: Elaborado pelo autor

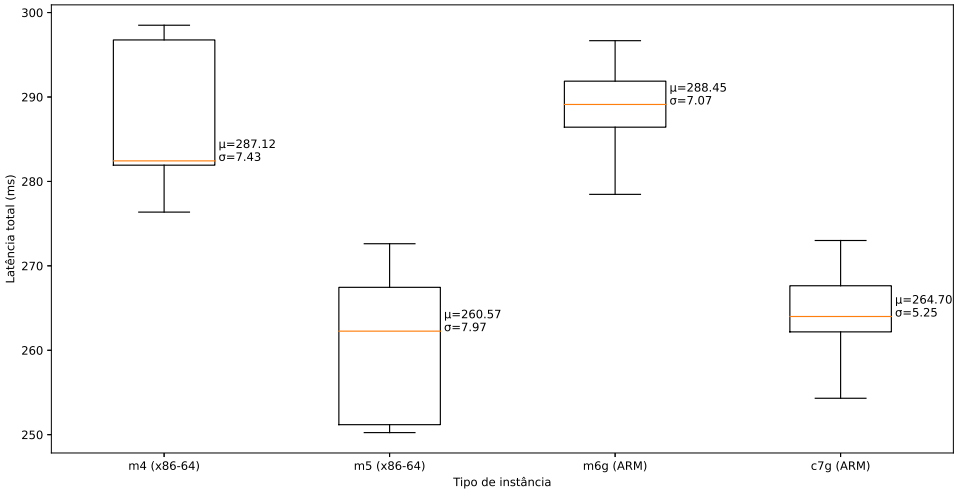
6.3 Recursos para reprodução das pesquisas

Segundo o mencionado na seção 3.3, o presente trabalho utiliza ferramentas para melhorar a reprodutibilidade das pesquisas, como Docker e Terraform. Não obstante, os seguintes recursos são disponibilizados como material complementar aos resultados:

- Repositório público com código-fonte no GitHub: No repositório[1] estão presentes os códigos necessários para o provisionamento da infraestrutura, *build* das imagens e também realização dos experimentos e análise dos mesmos. Não obstante, todos os resultados dos experimentos estão presentes no repositório, sob a forma de arquivos CSV.
- Repositório público de imagens Docker: Todas as imagens usadas para a realização dos experimentos estão presentes em um repositório público[2], de livre acesso.
- Documentação: No repositório público do GitHub há um arquivo denominado `README.md`, contendo orientações de como utilizar as ferramentas como Docker e Terraform para geração de imagens e o provisionamento das infraestruturas utilizadas neste trabalho.

[1]Link para o GitHub: <https://github.com/MatheusWoeffel/dr-detection-deploy>
[2]Link para o ECR: <https://gallery.ecr.aws/g1e7s9u4/dr-detection-deploy>

Figura 6.12: Tempo de latência total para o modelo MobileNet quando executado através do serviço *real-time inference* por tipo de instância utilizado. Nota-se que não há uma diferença significativa quanto aos diferentes tipos de instância.



Fonte: Elaborado pelo autor

7 CONCLUSÃO

Neste trabalho implementou-se uma solução para *deployment* de modelos de *deep learning* para a detecção de retinopatia diabética na nuvem. Foi utilizado o serviço Amazon *Sagemaker*, sendo consideradas duas alternativas distintas deste mesmo serviço: *serverless inference* e *real-time inference*. Conforme mostrado, as duas alternativas apresentam características distintas de desempenho e custo financeiro.

Baseado em uma alternativa *serverless*, foi possível realizar o *deployment* dos modelos em uma solução de custo financeiro reduzido, sob demanda e sem a necessidade de provisionar e configurar instâncias para a execução dos algoritmos. Consoante ao demonstrado no trabalho, essa alternativa sofre de latências elevadas em casos de *cold-start*, que chegam ser até $57\times$ maiores que a latência no caso normal.

Para mitigar este problema, mostrou-se que é possível reduzir a latência de *cold-start* a partir de otimizações para redução do tamanho das imagens Docker e por meio de otimizações da configuração de memória dos endpoints, resultando em aumento de desempenho de até $2,13\times$. Não obstante, a otimização da configuração de memória dos endpoints também trouxe ganho de desempenho para o caso de latência normal, melhorando o desempenho em até $2,16\times$ e reduzindo o custo financeiro em $8,5\%$.

Também foi analisada a alternativa *real-time*, que embora apresente um custo financeiro maior comparado à alternativa *serverless*, deixa de sofrer com o caso de latência de *cold-start* citado, portanto, sendo uma solução mais uniforme em termos de latência apresentada pela aplicação.

A partir da utilização de diferentes arquiteturas de processadores para a execução de *real-time inferences*, mostrou-se que é possível otimizar o tempo de latência total em até $1.5\times$, a partir da utilização de uma instância baseada em arquitetura x86-64. Interessantemente, foi constatado que para modelos menores não há uma diferença significativa no desempenho das arquiteturas analisadas.

A conversão do código para a utilização de imagens ARM, e a subsequente possibilidade de utilização das gerações de instâncias mais recentes, baseadas na plataforma ARM, se mostrou uma possibilidade de redução de custo financeiro. Especialmente para modelos menores, como foi comentado anteriormente, não há uma diferença considerável entre as arquiteturas, sendo, portanto, possível obter redução de custo financeiro de até $24,34\%$ sem prejuízos de desempenho.

Conforme demonstrado, as otimizações e diferentes alternativas exploradas se

comportaram de maneira diferente a depender do tamanho dos modelos utilizados. Nesse sentido, um tópico de estudo futuro é a utilização de técnicas de quantização de redes neurais e a análise das consequências em termos de desempenho e custo financeiro das soluções estudadas neste trabalho, avaliadas nos modelos otimizados. Dada o contexto médico da aplicação, também será necessário fazer uma análise cuidadosa do impacto das técnicas de quantização na acurácia e sensibilidade dos modelos.

Outro tópico de melhoria do trabalho é a utilização de diferentes arquiteturas para a execução da inferência dos modelos. Além do uso de CPUs, como pesquisado neste trabalho, seria interessante utilizar instâncias com processadores de propósito específico destinados a otimizar a execução de inferências. Como exemplo, destacam-se instâncias do tipo `inf1`, que se utilizam de *systolic arrays* para otimizar a computação de multiplicação de matrizes, muito presente na execução de inferência dos modelos.

REFERÊNCIAS

ABADI, M. et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. **arXiv preprint arXiv:1603.04467**, 2016.

AGUILAR, L. et al. Experiments as code: A concept for reproducible, auditable, debuggable, reusable, & scalable experiments. **arXiv preprint arXiv:2202.12050**, 2022.

AMAZON. **Machine learning in the AWS cloud: Add intelligence to applications with Amazon Sagemaker and Amazon Rekognition**. 2019. Disponível em: https://aws.amazon.com/sagemaker/. Acesso em: 10 de fev. de 2023.

BALDINI, I. et al. Serverless computing: Current trends and open problems. **Research advances in cloud computing**, Springer, p. 1–20, 2017.

BIDARI, I. et al. Deploying machine learning inference on diabetic retinopathy in binary and multi-class classification. In: IEEE. **2021 International Conference on Industrial Electronics Research and Applications (ICIERA)**. [S.l.], 2021. p. 1–6.

BOETTIGER, C. An introduction to docker for reproducible research. **ACM SIGOPS Operating Systems Review**, ACM New York, NY, USA, v. 49, n. 1, p. 71–79, 2015.

BUCHANAN, I. **Containers vs virtual machines**. 2023. Disponível em: <https: //www.atlassian.com/microservices/cloud-computing/containers-vs-vms>, Acesso em: 10 de mar. de 2023.

CACHO, J. R. F.; TAGHVA, K. The state of reproducible research in computer science. In: SPRINGER. **17th International Conference on Information Technology–New Generations (ITNG 2020)**. [S.l.], 2020. p. 519–524.

DAI, L. et al. A deep learning system for detecting diabetic retinopathy across the disease spectrum. **Nature communications**, Nature Publishing Group, v. 12, n. 1, p. 1–11, 2021.

DICKSON, B. **What are Artificial Neural Networks (ann)?** 2019. Disponível em: <https://bdtechtalks.com/2019/08/05/what-is-artificial-neural-network-ann/>, Acesso em: 05 de mar. de 2023.

FEDERATION, I. D. **IDF Diabetes Atlas**. Brussels, Belgium: International Diabetes Federation, 2021.

HASHICORP. **What is terraform: HashiCorp developer**. 2021. Disponível em: <https://developer.hashicorp.com/terraform/intro>. Acesso em: 09 de mar. de 2023.

HUDGEON, D.; NICHOL, R. **Amazon Sagemaker API Reference**. [S.l.]: Manning Publications Co., 2020. Disponível em <https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_runtime_InvokeEndpoint.html> Acesso em: 02 de mar. de 2023.

HUDGEON, D.; NICHOL, R. **Best practices for deploying models on SageMaker Hosting Services**. [S.l.]: Manning Publications Co., 2020. Disponível em: <https://docs.aws.amazon.com/sagemaker/latest/dg/deployment-best-practices.html>. Acesso em: 10 de mar. de 2023.

INC, G. **Tensorflow-CPU-AWS**. 2022. Disponível em: <https://pypi.org/project/tensorflow-cpu-aws/>. Acesso em: 07 de mar. de 2023.

ISHAKIAN, V.; MUTHUSAMY, V.; SLOMINSKI, A. Serving deep learning models in a serverless platform. In: IEEE. **2018 IEEE International Conference on Cloud Engineering (IC2E)**. [S.l.], 2018. p. 257–262.

IVANOVA, D.; BOROVSKA, P.; ZAHOV, S. Development of paas using aws and terraform for medical imaging analytics. In: AIP PUBLISHING LLC. **AIP Conference Proceedings**. [S.l.], 2018. v. 2048, n. 1, p. 060018.

JEGANNATHAN, A. P.; SAHA, R.; ADDYA, S. K. A time series forecasting approach to minimize cold start time in cloud-serverless platform. In: IEEE. **2022 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)**. [S.l.], 2022. p. 325–330.

KALKEN, S. V. **What are namespaces and cgroups, and how do they work?** 2023. Disponível em: <https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work>. Acesso em: 21 de fev. de 2023.

KANG, X.; SONG, B.; SUN, F. A deep similarity metric method based on incomplete data for traffic anomaly detection in iot. **Applied Sciences**, MDPI, v. 9, n. 1, p. 135, 2019.

KARTHIK MAGGIE, S. D. **APTOS 2019 Blindness Detection**. Kaggle, 2019. Available from Internet: <https://kaggle.com/competitions/aptos2019-blindness-detection>.

KAVLAKOGLU, E. **AI vs. Machine Learning vs. Deep Learning vs. neural networks: What's the difference?** 2020. Disponível em: <https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks>. Acesso em: 16 de mar. de 2023.

LIBERTY, E. et al. Elastic machine learning algorithms in amazon sagemaker. In: **Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data**. [S.l.: s.n.], 2020. p. 731–737.

LIEW, G.; MICHAELIDES, M.; BUNCE, C. A comparison of the causes of blindness certifications in england and wales in working age adults (16–64 years), 1999–2000 with 2009–2010. **BMJ open**, British Medical Journal Publishing Group, v. 4, n. 2, p. e004015, 2014.

MOREIRA, F. et al. Impacto da resolução na detecção de retinopatia diabética com uso de deep learning. In: **Anais do XX Simpósio Brasileiro de Computação Aplicada à Saúde**. Porto Alegre, RS, Brasil: SBC, 2020. p. 494–499. ISSN 2763-8952. Available from Internet: <https://sol.sbc.org.br/index.php/sbcas/article/view/11546>.

PAVATE, A. et al. Diabetic retinopathy detection-mobilenet binary classifier. **Acta Sci Med Sci**, v. 4, n. 12, p. 86–91, 2020.

REDHAT. **Types of cloud computing**. 2022. Disponível em: <https://www.redhat.com/en/topics/cloud-computing/public-cloud-vs-private-cloud-and-hybrid-cloud>. Acesso em: 07 de set. de 2022.

REDHAT. **What is serverless?** [S.l.]: Redhat, 2022. Disponível em: <https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless>. Acesso em: 05 de mar. de 2023.

STACKOVERFLOW. **Stack overflow developer survey 2022**. 2022. Disponível em: <https://survey.stackoverflow.co/2022/>. Acesso em: 15 de jan. de 2023.

STROPPA, D. **Overview of containers for Amazon sagemaker**. 2021. Disponível em: <https://sagemaker-workshop.com/custom/containers.html>. Acesso em: 03 de mar. de 2023.

THAKUR, V. **How to build optimal docker images**. 2020. Disponível em: <https://www.metricfire.com/blog/how-to-build-optimal-docker-images/>. Acesso em: 22 de jan. de 2023.

VLADUSIC, D.; RADOLOVIC, D. Infrastructure as code for heterogeneous computing. In: IEEE. **2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)**. [S.l.], 2020. p. 1–2.

WEI, L. et al. Towards efficient resource allocation for heterogeneous workloads in iaas clouds. **IEEE Transactions on Cloud Computing**, IEEE, v. 6, n. 1, p. 264–275, 2015.