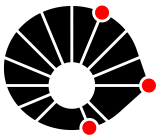

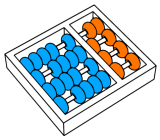

Simulador MobFogSim com Suporte a Treinamento Federado e Aprendizado de Máquina

L. L. Martins *L. F. Bittencourt* *D. M. Gonçalves*

Relatório Técnico - IC-PFG-24-27
Projeto Final de Graduação
2024 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Simulador MobFogSim com Suporte a Treinamento Federado e Aprendizado de Máquina

L. L. Martins* L. F. Bittencourt D. M. Gonçalves

Resumo

O *Federated Learning* (FL) é uma abordagem emergente em *Machine Learning* (ML) que permite o treinamento colaborativo de modelos sem a necessidade de centralizar os dados. Essa técnica tem ganhado destaque na literatura devido à sua capacidade de preservar a privacidade e possibilitar o treinamento em ambientes distribuídos, como dispositivos móveis conectados à borda da rede.

O aumento do interesse por FL reflete a relevância de explorar soluções descentralizadas para problemas que envolvem grandes volumes de dados e restrições de privacidade. O Federated Learning (FL) é uma boa solução porque permite que o pré-processamento e o treinamento dos dados sejam realizados diretamente nos dispositivos locais, preservando a privacidade ao evitar a transferência de dados brutos para servidores centrais. Em vez disso, apenas os parâmetros atualizados dos modelos (como pesos e gradientes) são compartilhados pela rede, reduzindo significativamente o tráfego de dados e os riscos associados à exposição de informações sensíveis.

O simulador MobFogSim é reconhecido por sua capacidade de replicar ambientes distribuídos, simulando a mobilidade de dispositivos móveis executando aplicações conectadas a servidores em arquitetura de computação na borda. Essas características tornam o simulador um ambiente adequado para estudos de treinamento federado, dada sua habilidade em representar cenários complexos de sistemas distribuídos e interações dinâmicas entre dispositivos e servidores.

Logo, neste projeto, tivemos como objetivo viabilizar a integração de modelos de *Machine Learning* (ML) ao simulador MobFogSim, ampliando suas funcionalidades para suportar estudos de treinamento federado em cenários com dispositivos móveis e dados descentralizados. Para alcançar esses objetivos, o projeto foi estruturado em três módulos principais: 1) o simulador foi adaptado para gerar logs detalhados por dispositivo, possibilitando a coleta de dados para treinamento federado utilizando o framework Flower; 2) um modelo de *Machine Learning* foi treinado de forma colaborativa e descentralizada; e 3) uma API, desenvolvida com FastAPI e PyTorch, foi implementada para integrar o modelo treinado ao simulador, permitindo a tomada de decisões dinâmicas durante as simulações.

Ao adicionar suporte a FL e ML no MobFogSim, possibilitamos que o simulador tome decisões baseadas em modelos de ML em tempo real, tornando-o uma opção para pesquisas que envolvem treinamento federado.

*Instituto de Computação, Universidade Estadual de Campinas

1 Introdução

O avanço da computação em névoa (*fog computing*) tem como objetivo estender os recursos da nuvem para a borda da rede, aproximando serviços e recursos de computação dos usuários finais. Essa proximidade traz diversos benefícios, como redução de latência, diminuição do consumo de largura de banda e maior contexto situacional, melhorando a experiência do usuário [1]. Entretanto, a mobilidade dos usuários representa um desafio significativo, pois a variação na proximidade entre dispositivos móveis e os recursos de névoa pode impactar negativamente o desempenho. Uma possível solução para esse problema é a migração de máquinas virtuais (VMs) ou contêineres entre nós de névoa, acompanhando a movimentação dos usuários para manter baixa latência e alta disponibilidade [1].

Nesse contexto, o **MobFogSim** [1] foi desenvolvido como uma extensão do simulador **iFogSim** para permitir o estudo de mobilidade de dispositivos e migração de VMs em ambientes de computação em névoa. Ele foi inicialmente proposto e detalhado no trabalho *MobFogSim: Simulation of mobility and migration for fog computing* [1], que apresentou suas principais funcionalidades e a capacidade de simular aspectos como padrões de mobilidade de usuários, eventos de *handoff* (troca da antena na qual o usuário está conectado) entre pontos de acesso e estratégias de migração de recursos. Esse simulador auxilia pesquisadores a avaliar o impacto de diferentes políticas e técnicas em cenários com dispositivos móveis. O MobFogSim [1] também permite a simulação de aplicações distribuídas e cenários de carga de trabalho realistas, como os padrões de mobilidade do *Luxembourg SUMO Traffic* (LuST) [2].

Com a crescente popularidade do treinamento federado (FL), uma técnica emergente em *Machine Learning*, surge a oportunidade de integrar essa abordagem ao MobFogSim [1] para explorar sua aplicação em ambientes distribuídos e móveis. O treinamento federado permite a construção de modelos de forma colaborativa, mantendo os dados localizados em dispositivos individuais, o que preserva a privacidade e reduz a necessidade de transferência de dados sensíveis. Essa característica torna o FL especialmente relevante em cenários que demandam privacidade e eficiência no uso da rede [3].

Dessa forma, a estrutura descentralizada e móvel do MobFogSim [1] o torna um ambiente viável para a incorporação e estudo do treinamento federado, permitindo simular e avaliar como essa técnica pode ser aplicada em sistemas distribuídos, como a computação em névoa, para tomada de decisões em tempo real.

Este trabalho tem como objetivo possibilitar a integração de modelos de ML ao MobFogSim [1] e explorar o treinamento federado no contexto da simulação. Para isso, três módulos principais foram desenvolvidos:

1. O **simulador MobFogSim** foi adaptado para gerar logs detalhados por dispositivo, criando um conjunto de dados realista para experimentos em aprendizado federado.

2. O **treinamento federado** foi implementado utilizando o framework **Flower** [5], que permite a colaboração entre dispositivos simulados para treinar modelos de ML de forma descentralizada.

3. Uma **API de integração**, desenvolvida com **FastAPI** [6] e **PyTorch** [7], expõe os modelos treinados para que possam ser utilizados em tempo real pelo simulador, possibilitando decisões dinâmicas e inteligentes durante as simulações.

A combinação desses três módulos possibilitou validar conceitos de treinamento federado, além de demonstrar a viabilidade da integração de ML com o simulador.

2 Referencial Teórico

2.1 Computação em Névoa

A computação em névoa (*fog computing*) é um paradigma que expande os recursos computacionais da nuvem em direção à borda da rede, aproximando os serviços de processamento e armazenamento dos usuários finais. Diferentemente da computação em nuvem, que centraliza esses recursos em data centers remotos, a computação em névoa distribui a infraestrutura em múltiplas camadas hierarquicamente organizadas, localizadas entre os usuários e a nuvem. Essas camadas podem variar em número e composição, dependendo do domínio da aplicação e de seus requisitos específicos [1].

Geralmente, a camada superior é representada pela nuvem, enquanto a camada inferior é composta por dispositivos ou *cloudlets* próximos geograficamente ao usuário, como pontos de acesso de primeira conexão. Essa organização permite que aplicações habilitadas para computação em névoa utilizem *cloudlets* para armazenar e processar dados, idealmente na unidade mais próxima ao usuário. Para manter essa proximidade, especialmente em cenários de mobilidade, os dados e processos associados ao usuário devem migrar junto com ele, a fim de minimizar a latência e preservar a qualidade do serviço [1].

Esses aspectos tornam a computação em névoa uma solução promissora para lidar com os desafios impostos por aplicações que exigem baixa latência, como serviços em tempo real, IoT e realidade aumentada, especialmente em cenários de alta mobilidade. Ao mesmo tempo, a descentralização oferecida pela névoa é altamente alinhada com paradigmas emergentes, como o treinamento federado, que requerem suporte para computação distribuída e preservação de privacidade [1].

2.2 Treinamento Federado

O treinamento federado é uma abordagem emergente no *Machine Learning* que permite a construção de modelos de forma descentralizada, mantendo os dados localizados nos dispositivos de origem. Em vez de centralizar os dados em um único servidor para o treinamento, o aprendizado federado utiliza atualizações de modelo provenientes de cada dispositivo para criar um modelo global [3]. Essa estratégia é particularmente relevante em contextos que envolvem privacidade de dados, como dispositivos IoT, serviços de saúde e aplicações financeiras.

O aprendizado federado enfrenta desafios como a heterogeneidade de dados e dispositivos. A heterogeneidade de dados refere-se a distribuições estatísticas diferentes em cada dispositivo, enquanto a heterogeneidade de sistemas está relacionada à diversidade em capacidades computacionais, armazenamento e conectividade. Técnicas como *Federated Averaging* (FedAvg) e suas variantes, como FedProx, são usadas para mitigar esses problemas, agregando atualizações de modelos locais de forma eficiente [4].

Neste trabalho, utilizamos o framework **Flower** [5] para implementar o treinamento federado. O Flower é uma ferramenta flexível e escalável, projetada para experimentos de aprendizado federado em larga escala. Ele permite que dispositivos simulados (no caso, clientes federados representados pelos logs do MobFogSim [1]) colaborem com um servidor central para treinar modelos globais, mantendo a descentralização e a privacidade dos dados. A arquitetura do Flower suporta diversos algoritmos de treinamento e integrações com bibliotecas como PyTorch, proporcionando uma solução robusta para experimentação em aprendizado federado.

2.3 MobFogSim

O **MobFogSim** [1] é uma extensão do simulador **iFogSim**, projetado para estudar a computação em névoa, a mobilidade de dispositivos e a migração de máquinas virtuais (VMs) em ambientes distribuídos. A computação em névoa é um paradigma que se situa entre a nuvem e os dispositivos de borda, aproximando recursos de processamento e armazenamento dos usuários finais. Isso reduz a latência e o consumo de largura de banda, além de melhorar a segurança, a privacidade e a continuidade dos serviços [1].

No MobFogSim [1], dispositivos móveis, conhecidos como *fog nodes*, conectam-se a pontos de acesso e migram entre eles conforme os usuários se movem. Para otimizar a alocação de recursos e manter baixa latência, o simulador permite simular eventos de handoff e estratégias de migração de VMs. Esses cenários são fundamentais para avaliar a eficácia de políticas de migração e o impacto da mobilidade em ambientes de computação em névoa [1].

A relevância do MobFogSim [1] para este trabalho está na sua capacidade de simular dispositivos móveis descentralizados, criando um ambiente ideal para experimentos de treinamento federado. A geração de logs detalhados por dispositivo fornece dados realistas para o treinamento de modelos de *Machine Learning*, que podem ser usados para suportar decisões de migração no simulador.

2.4 Integração com Modelos de ML

Para conectar o MobFogSim [1] a modelos de *Machine Learning* (ML), desenvolvemos uma API utilizando **FastAPI** [6], um framework moderno para a construção de APIs de alta performance em Python. A API serve como um intermediário, expondo os modelos treinados e permitindo que o simulador tome decisões baseadas em ML em tempo real. As principais características da API incluem:

- **Carregamento de Modelos**: A API permite carregar modelos treinados em frameworks como PyTorch, que é amplamente utilizado para aprendizado profundo.
- **Inferência em Tempo Real**: A API recebe dados de entrada (por exemplo, parâmetros do estado do dispositivo) e retorna decisões baseadas no modelo, como o momento ideal para migração.
- **Modularidade**: A estrutura da API suporta facilmente a substituição do modelo por versões mais complexas, possibilitando futuras melhorias.

O framework **PyTorch** [7] foi utilizado para treinar e implementar os modelos de ML. Ele é reconhecido por sua flexibilidade e suporte extensivo a redes neurais profundas. Com PyTorch, os modelos podem ser otimizados para diferentes tipos de dados e implementados diretamente na API, facilitando a integração com o MobFogSim [1].

2.5 Integração dos Módulos

Os três módulos principais deste trabalho—simulador, treinamento federado e API— interagem de forma coesa. O MobFogSim [1] gera logs detalhados por dispositivo, que são utilizados como dados para treinar modelos de ML no ambiente federado do framework Flower. Após o treinamento, os modelos são carregados na API, que se comunica com o simulador para tomar decisões em tempo real durante as simulações. Essa integração permite não apenas validar o treinamento federado em um ambiente realista, mas também demonstrar como decisões baseadas em ML podem aprimorar o desempenho de sistemas de computação em névoa.

A arquitetura proposta é escalável e modular, abrindo caminho para experimentos futuros com modelos de maior complexidade e cenários mais desafiadores em computação distribuída.

3 Metodologia

Neste projeto, realizamos uma série de atividades para integrar o simulador **MobFogSim** [1] com modelos de Machine Learning (ML), explorando o uso de treinamento federado e sua integração com a simulação de computação em névoa. Toda a implementação apresentada neste trabalho está disponível no repositório público no GitHub.[1] O repositório contém documentações detalhadas sobre cada módulo desenvolvido, incluindo o simulador, os ajustes para coleta de logs, a API de integração e o fluxo de treinamento federado. A seguir, apresentamos um resumo das principais etapas:

- Realizamos ajustes no **MobFogSim** [1] para incluir logs detalhados por dispositivo em cada execução da simulação. Esses logs servem como dados de entrada para o treinamento de modelos de ML. Além disso, implementamos uma nova estratégia de migração (*migration strategy*) que utiliza os dados locais de cada dispositivo para realizar chamadas à API que contém o modelo de ML.

- Desenvolvemos uma API em **FastAPI**, que faz o carregamento de modelos de ML treinados utilizando **PyTorch**. Essa API expõe o modelo em tempo real por meio de uma rota HTTP, permitindo que o simulador consulte o modelo durante as simulações para suportar decisões de migração.

- Desenvolvemos um modelo de ML treinado com aprendizado federado, utilizando o framework **Flower**. Este modelo foi projetado com simplicidade, uma vez que o objetivo principal era validar a viabilidade de integração com o simulador. O modelo

[1] Repositório GitHub: *https://github.com/leolivrare/MobFogSim-with-ML*

foi criado para realizar uma tarefa específica e direta: determinar se um dispositivo deve ou não realizar uma migração, utilizando dados previamente computados como *features* para a tomada de decisão.

A Figura 1 ilustra a interação entre os diferentes módulos do projeto, detalhando como os dados fluem entre o simulador, o treinamento federado e a API.

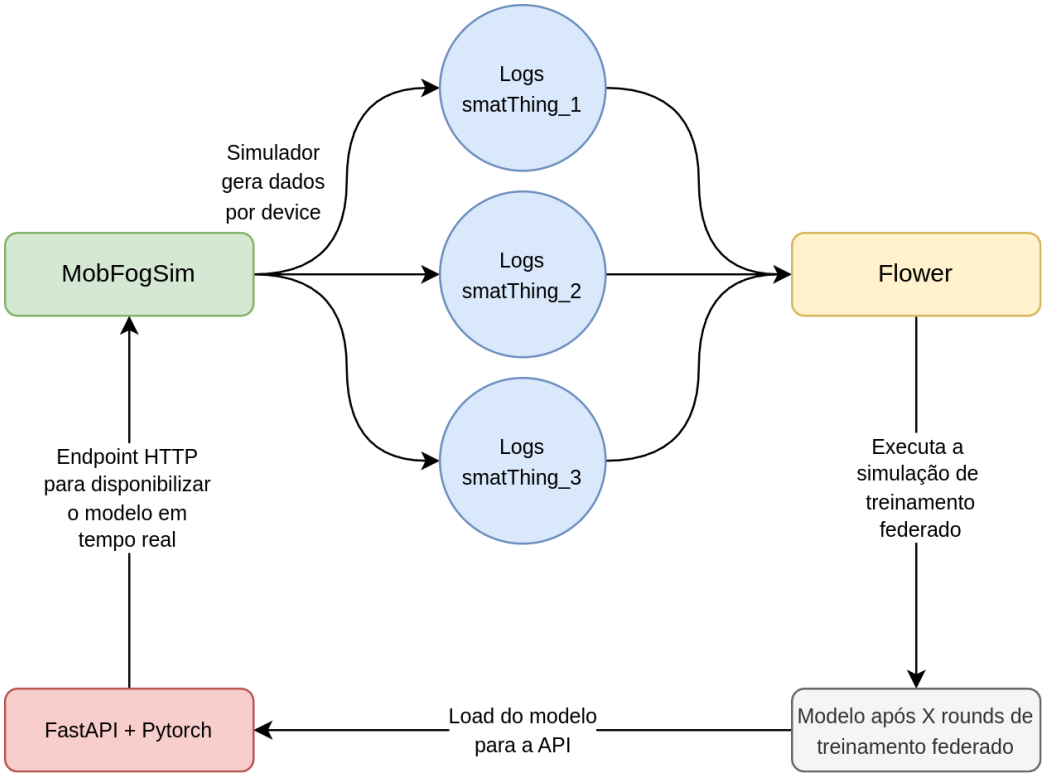

Figura 1: Interação entre os módulos do projeto: MobFogSim, Flower e a API FastAPI.

Nos próximos subtópicos, detalharemos cada um dos módulos desenvolvidos, apresentando as modificações realizadas, os métodos aplicados e a forma como eles interagem para alcançar os objetivos do projeto.

3.1 Ajustes no MobFogSim para Geração de Logs Detalhados

Para possibilitar o uso dos dados do simulador no treinamento de modelos, foi preciso modificar o *MobFogSim* [1] para que ele pudesse gerar logs detalhados por dispositivo durante a simulação. Essas informações são essenciais para compreender o comportamento dos dispositivos móveis e, principalmente, para fornecer dados ricos que possam ser utilizados no treinamento de modelos de *Machine Learning*. Dois tipos de logs principais foram implementados:

3.1.1 Logs de Latência dos Dispositivos (*logs_device_latency*)

Essa funcionalidade foi desenvolvida para capturar detalhes abrangentes sobre a latência experimentada por cada dispositivo ao longo da simulação. Os dados são armazenados em arquivos CSV, organizados por dispositivo, contendo os seguintes campos:

- **Index**: Identificador único de cada registro.
- **Time**: Tempo do evento em relação ao início da simulação.
- **Latency**: Latência observada no momento do registro.
- **AppId**: Identificador da aplicação em execução no dispositivo.
- **SmartThingMyId**: Identificador único do dispositivo móvel.
- **ServerCloudletName**: Nome da *cloudlet* que hospeda a aplicação do dispositivo.
- **PosX** e **PosY**: Coordenadas do dispositivo no ambiente simulado.
- **Direction** e **Speed**: Direção e velocidade do dispositivo.
- **SourceAp**: Ponto de acesso (*Access Point*) associado ao dispositivo no momento do registro.

Esses dados detalham o estado do dispositivo e suas interações com a infraestrutura de *fog computing*, sendo valiosos para análises e modelos preditivos, como decisões de migração.

3.1.2 Logs de Decisão de Migração (*logs_migration_decision*)

Para capturar informações mais específicas relacionadas às decisões de migração, foi implementada uma funcionalidade adicional que gera logs em momentos críticos de decisão para cada dispositivo. Esses dados são armazenados em arquivos CSV organizados por dispositivo, contendo os seguintes campos:

- **Time**: Tempo do evento em relação ao início da simulação.
- **DeviceId**: Identificador único do dispositivo.
- **PosX** e **PosY**: Coordenadas do dispositivo no momento do registro.
- **Direction** e **Speed**: Direção e velocidade do dispositivo.
- **SourceAp**: Ponto de acesso atual do dispositivo.
- **DistanceToSourceAp**: Distância do dispositivo ao seu ponto de acesso atual.
- **MigrationTime**: Tempo estimado para realizar a migração.
- **ShouldMigrate**: Decisão de migração (sim ou não).

- **NextServerCloudlet** e **NextAp**: Próxima *cloudlet* e próximo ponto de acesso, respectivamente, caso a migração seja aprovada.
- **Reason**: Justificativa para a decisão de migração.
- **LocalCloudlet** e **DistanceToLocalCloudlet**: Nome e distância para a *cloudlet* local do dispositivo.
- **ClosestCloudlet** e **DistanceToClosestCloudlet**: Nome e distância para a *cloudlet* mais próxima do dispositivo.
- **IsMigPoint** e **IsMigZone**: Indicadores binários que verificam se o dispositivo está em um ponto ou zona de migração.

Esses logs foram projetados para capturar todos os fatores relevantes que influenciam a decisão de migração, oferecendo uma visão completa sobre os critérios de decisão no contexto da simulação.

3.1.3 Implementação Técnica

Os ajustes realizados no *MobFogSim* [1] incluíram a modificação da classe **MyStatistics**, onde foram adicionados dois novos métodos principais: **putLatencyFileValueCSV** e **logMigrationMetrics**.

Registro de Dados de Latência: O método **putLatencyFileValueCSV** foi implementado para registrar os dados detalhados de latência experimentada por dispositivos durante a simulação. Ele realiza a gravação dessas informações em arquivos no formato CSV, específicos para cada dispositivo. Os dados registrados incluem o tempo do evento, a latência observada, o identificador do aplicativo em execução, as coordenadas de posição do dispositivo, a direção e velocidade do movimento, além do ponto de acesso e da *cloudlet* conectados no momento. Esse método também incrementa contadores individuais para cada dispositivo, assegurando a organização e integridade dos registros. Em caso de erro durante o processo de escrita, uma exceção é tratada e registrada.

Registro de Decisões de Migração: O método **logMigrationMetrics** foi criado para armazenar informações detalhadas relacionadas às decisões de migração dos dispositivos. Ele registra dados como o tempo do evento, as coordenadas do dispositivo, a direção e velocidade de movimento, a distância para o ponto de acesso atual, o tempo estimado para realizar a migração, a decisão de migração (se ela ocorreu ou não), a *cloudlet* e o ponto de acesso previstos para a próxima conexão, e a justificativa para a decisão tomada. O método também inclui informações adicionais sobre as distâncias para outras *cloudlets* relevantes e indicadores binários para determinar se o dispositivo está em um ponto ou zona de migração. Antes de registrar os dados, o método verifica se os arquivos de registro para o dispositivo já existem; caso contrário, ele cria novos arquivos e adiciona os cabeçalhos apropriados. Caso ocorram erros no processo de escrita, uma exceção é tratada para evitar interrupções na simulação.

Integração nas Estratégias de Migração: Para integrar os métodos de registro ao simulador, foram realizadas alterações na classe **Actuator**, que agora chama o método **putLatencyFileValueCSV** para registrar dados de latência durante a execução da simulação. O método é acionado sempre que um dispositivo interage com a infraestrutura da *fog computing*, assegurando que cada interação relevante seja registrada.

Além disso, estratégias de migração, como a **LowestLatency**, foram ajustadas para utilizar o método **logMigrationMetrics**. Esse método é chamado no momento da decisão de migração para registrar todos os fatores que influenciam a escolha. Os dados coletados incluem a distância entre o dispositivo e o ponto de acesso, o tempo estimado para completar a migração e informações sobre o ambiente local, como o nome e a distância para as *cloudlets* mais próximas. A estratégia também registra se o dispositivo está em um ponto ou zona de migração, garantindo que todas as condições sejam documentadas.

Objetivo e Impacto das Alterações: Essas modificações ampliam significativamente a capacidade do *MobFogSim* [1] de capturar dados detalhados sobre o comportamento dos dispositivos e suas interações com a infraestrutura de computação em névoa. Com os dados coletados, torna-se possível realizar análises aprofundadas sobre o impacto de diferentes estratégias de migração, bem como treinar e validar modelos de *Machine Learning* que utilizem os registros como base para predições. Dessa forma, o simulador foi transformado em uma ferramenta ainda mais robusta e versátil, permitindo experimentos avançados em *fog computing* e mobilidade.

3.2 Criação de uma Nova Estratégia de Migração Baseada em API

Para demonstrar a integração do *MobFogSim* [1] com modelos de *Machine Learning*, desenvolvemos uma nova estratégia de migração chamada **DecisionMigrationAPI**. Essa estratégia faz uso de uma API para delegar parte da lógica de decisão de migração a um modelo de ML, expondo o simulador a um modelo externo, treinado e hospedado separadamente.

3.2.1 Funcionamento da Estratégia *DecisionMigrationAPI*

A *DecisionMigrationAPI* coleta diversos dados do dispositivo móvel no momento em que uma decisão de migração deve ser tomada. Esses dados incluem:

- Coordenadas do dispositivo (**PosX**, **PosY**);
- Direção e velocidade do dispositivo (**Direction**, **Speed**);
- Distâncias relevantes:
  - Distância até o ponto de acesso atual (**DistanceToSourceAp**);
  - Distância até a *cloudlet* local (**DistanceToLocalCloudlet**);
  - Distância até a *cloudlet* mais próxima que não seja a local (**DistanceToClosestCloudlet**);

- Indicadores binários sobre a zona e o ponto de migração (**IsMigZone**, **IsMigPoint**).

Esses dados são encapsulados em um objeto JSON e enviados à API em uma requisição para o endpoint `should_migrate` (a URL base da API pode mudar se estiver executando localmente ou em outro ambiente). A API utiliza um modelo de ML para analisar os dados e retorna uma decisão, indicando se o dispositivo deve ou não migrar.

Se a API aprovar a migração, a estratégia continua avaliando as condições locais para determinar a próxima *cloudlet* e o próximo ponto de acesso (*Access Point, AP*). Caso contrário, a migração é rejeitada, e os motivos são registrados nos logs.

3.2.2 Flexibilidade e Expansibilidade

Embora a *DecisionMigrationAPI* tenha sido implementada como um exemplo simples para validar a integração com a API, sua arquitetura é flexível o suficiente para acomodar modelos mais sofisticados. Por exemplo:

- Um modelo de ML poderia recomendar diretamente a próxima *cloudlet* e o próximo AP em vez de apenas aprovar ou rejeitar a migração.
- Poderiam ser integrados modelos que consideram múltiplos fatores, como carga nos servidores, previsões de mobilidade, ou consumo de energia.

3.2.3 Implementação Técnica

A nova estratégia foi implementada como uma classe chamada **DecisionMigrationAPI**, que segue a interface *DecisionMigration*. Seu método principal, denominado `shouldMigrate`, realiza a tomada de decisão de migração com base em dados coletados do dispositivo e da infraestrutura. O método segue os seguintes passos:

1. **Coleta de Dados:** Inicialmente, são coletados dados relevantes do dispositivo e da infraestrutura utilizando métodos fornecidos pelo simulador. Esses dados incluem informações como posição ($PosX$, $PosY$), direção e velocidade do dispositivo, distância até o ponto de acesso atual ($SourceAp$), distância para a *cloudlet* local e para a *cloudlet* mais próxima, além de indicadores binários que verificam se o dispositivo está em um ponto ou zona de migração.
2. **Envio de Dados à API:** Após a coleta, os dados são organizados em formato JSON e enviados por meio de uma requisição para a API. A API processa essas informações e retorna uma decisão sobre a necessidade de migração.
3. **Processamento da Resposta da API:** A resposta da API é avaliada para determinar a próxima ação:
    - Se a resposta indicar que a migração não é necessária, a decisão é registrada nos logs com uma justificativa adequada.
    - Caso a resposta indique que a migração deve ser realizada, a estratégia utiliza métodos adicionais do simulador para calcular a próxima *cloudlet* e o próximo ponto de acesso (*Access Point, AP*) para o dispositivo.

Comunicação com a API: A comunicação com a API é realizada por meio de um método dedicado que encapsula toda a lógica de preparação e envio da requisição, bem como o processamento da resposta. Esse método segue os seguintes passos:

1. **Preparação dos Dados:** Um objeto contendo todas as informações necessárias sobre o estado do dispositivo e da infraestrutura é criado. Este objeto é transformado em uma string JSON, garantindo que os dados estejam no formato apropriado para serem interpretados pela API.

2. **Envio da Requisição HTTP:** A string JSON é enviada para a API por meio de uma conexão HTTP. A URL utilizada para essa comunicação aponta para o endpoint específico da API responsável por processar os dados e retornar uma decisão de migração.

3. **Recebimento e Validação da Resposta:** A resposta da API, se bem-sucedida, contém uma string que indica a decisão de migração. Caso a resposta seja nula ou um erro ocorra durante o envio, uma mensagem de erro é gerada e a migração é automaticamente rejeitada. Se a resposta for válida, ela é processada para interpretar o valor retornado.

4. **Interpretação da Resposta:** O valor retornado pela API é convertido em um resultado booleano, indicando se a migração deve ou não ocorrer. Essa interpretação é essencial para que a estratégia tome a decisão adequada.

Garantia de Confiabilidade e Registro de Erros: O método de comunicação foi projetado para lidar com possíveis falhas durante o envio ou recebimento da requisição HTTP. Caso um erro seja detectado (como a ausência de resposta da API ou problemas na comunicação), uma exceção é tratada e registrada nos logs do simulador. Isso garante que, mesmo em situações adversas, o simulador continue a operar de maneira robusta, evitando interrupções na execução.

Objetivo e Impacto da Integração: A implementação dessa estratégia demonstra a capacidade do simulador *MobFogSim* [1] de integrar-se a sistemas externos, como uma API baseada em modelos de *Machine Learning*. Essa abordagem não apenas valida a viabilidade de usar decisões baseadas em inteligência artificial no contexto de *fog computing*, mas também amplia as possibilidades de pesquisa futura, permitindo a integração de modelos mais complexos e avançados. Assim, o simulador evolui como uma ferramenta versátil para experimentos que combinam mobilidade, computação em névoa e *Machine Learning*.

3.2.4 Integração com o *MobFogSim*

Para que o simulador suportasse a nova estratégia, foi necessário modificar o comportamento do `AppExample`, adicionando o suporte a um novo parâmetro para configurar a estratégia de migração. Agora, o quarto parâmetro de entrada do simulador pode assumir os seguintes valores:

- **0**: Migração baseada na menor latência (*Lowest Latency*);
- **1**: Migração baseada na menor distância entre o dispositivo e a *cloudlet* (*Lowest Distance Between SmartThing and ServerCloudlet*);
- **2**: Migração baseada na menor distância entre o dispositivo e o AP (*Lowest Distance Between SmartThing and AP*);
- **3**: Migração baseada na decisão da API (*DecisionMigrationAPI*).

Exemplo de parâmetros do simulador com a nova estratégia:

```
1 290538 0 3 11 0 0 0 61
```

3.2.5 Resultados e Benefícios

A adição da *DecisionMigrationAPI* demonstra a capacidade do *MobFogSim* [1] de integrar-se a modelos de *Machine Learning* externos, abrindo caminho para experimentos mais complexos. Além disso, a nova estratégia facilita o desenvolvimento e a validação de modelos mais complexos, possibilitando uma avaliação detalhada de suas decisões no contexto de *fog computing*.

Como o simulador lida com IoTs, ele oferece um ambiente ideal para validações de treinamento federado. Cada dispositivo móvel gera seus próprios dados de forma independente, permitindo o treinamento de modelos no formato federado. Com um modelo treinado, torna-se simples integrá-lo ao simulador através da *DecisionMigrationAPI*, possibilitando uma validação eficiente e dinâmica do modelo em cenários simulados de computação em névoa.

3.3 Fluxo de Treinamento Federado com Flower

O treinamento federado foi implementado utilizando a biblioteca *Flower (FL)*, com o objetivo de simular um ambiente onde cada cliente representa um dispositivo (ou veículo) no simulador *MobFogSim* [1]. Este treinamento permitiu desenvolver um modelo global de *Machine Learning* a partir de dados locais, distribuídos por diferentes clientes. O foco foi demonstrar a integração desse modelo federado com o *MobFogSim* [1]. A seguir, apresentamos o fluxo detalhado do treinamento federado.

3.3.1 Conceitos Fundamentais de Treinamento Federado

O treinamento federado é uma abordagem distribuída para *Machine Learning*, em que os dados permanecem localizados nos dispositivos, e apenas os parâmetros dos modelos (pesos e gradientes) são compartilhados com o servidor central. O processo é composto pelas seguintes etapas:

1. **Inicialização do Modelo Global:** O servidor central inicia com um modelo global (com pesos aleatórios ou pré-treinados) e o envia aos dispositivos participantes.

2. **Treinamento Local nos Dispositivos:** Cada dispositivo utiliza seus próprios dados locais para treinar o modelo recebido do servidor.
3. **Agregação no Servidor:** O servidor central recebe os parâmetros atualizados dos dispositivos e os combina para atualizar o modelo global.
4. **Iteração:** Esse ciclo é repetido por várias rodadas (*rounds*) até que o modelo global alcance a performance desejada.

3.3.2 Etapas do Fluxo de Treinamento Federado

Configuração do Ambiente e Dependências Inicialmente, o ambiente foi configurado para realizar o treinamento. As bibliotecas utilizadas incluíram:

- `flwr`: Para gerenciamento do treinamento federado.
- `torch` e `torchvision`: Para construção e treinamento local do modelo.

O treinamento foi configurado para suportar tanto `CPU` quanto `GPU`, garantindo flexibilidade e eficiência para diferentes ambientes de execução.

Exploração e Pré-processamento dos Dados Os dados utilizados foram extraídos do *MobFogSim* [1], contendo logs gerados por dispositivos móveis. Cada dispositivo foi tratado como um cliente federado com seu próprio subconjunto de dados. Esses dados foram processados e simplificados para conter apenas duas *features*: `IsMigPoint` e `IsMigZone`. Essa simplificação permitiu focar na validação do fluxo de integração, mas não comprometeu a possibilidade de utilizar modelos mais complexos futuramente.

Definição do Modelo de Aprendizado de Máquina O modelo utilizado no treinamento federado foi um *Perceptron Multicamadas (MLP)* simples, implementado em `PyTorch`. Este modelo possui:

- Uma camada de entrada configurada para receber duas *features*;
- Uma camada oculta com quatro neurônios ativados pela função ReLU;
- Uma camada de saída projetada para realizar uma predição binária que indica se o dispositivo deve migrar (`ShouldMigrate`).

Implementação dos Clientes Federados Os clientes federados foram desenvolvidos utilizando a interface `NumPyClient`, fornecida pela biblioteca *Flower*. Cada cliente é responsável por:

1. Receber os parâmetros do modelo global enviados pelo servidor central.
2. Realizar o treinamento local utilizando seu subconjunto de dados.
3. Avaliar o modelo em um conjunto de validação local, gerando métricas que são enviadas ao servidor.

Esse ciclo é repetido durante cada rodada de treinamento federado.

Agregação no Servidor Federado O servidor foi configurado para utilizar a estratégia **FedAvg (Federated Averaging)**, que realiza a combinação dos parâmetros recebidos dos clientes de forma ponderada, considerando o número de amostras disponíveis em cada cliente. Além disso, uma funcionalidade adicional foi implementada para salvar o modelo global ao final de cada rodada de treinamento, permitindo rastreabilidade e a possibilidade de reutilização do modelo.

Execução do Treinamento Federado O treinamento federado foi conduzido por 10 rodadas (*rounds*). Em cada rodada, o servidor coordenava o treinamento distribuído, enviando o modelo global para os clientes, recebendo as atualizações locais e realizando a agregação para formar um novo modelo global. Após cada rodada, o modelo global atualizado era salvo para análises posteriores.

Validação do Modelo Global Após a conclusão do treinamento federado, o modelo global final foi avaliado utilizando casos específicos para verificar sua capacidade de realizar predições corretas. O modelo demonstrou habilidade em aprender a lógica das decisões baseadas nas duas *features* fornecidas (`IsMigPoint` e `IsMigZone`), validando assim o fluxo de integração entre o treinamento federado e o simulador *MobFogSim* [1].

O modelo utilizado no treinamento foi simplificado, com o objetivo principal de validar a integração entre o treinamento federado e o *MobFogSim* [1]. A estrutura estabelecida, no entanto, pode ser facilmente adaptada para treinar modelos mais complexos utilizando todos os logs disponíveis no simulador, explorando mais os cenários reais de *fog computing* ou qualquer outra aplicação distribuída.

3.4 API para Inferência em Tempo Real

Para possibilitar a integração do modelo treinado com o simulador *MobFogSim* [1], desenvolvemos uma API utilizando a biblioteca *FastAPI*. Essa API expõe um endpoint capaz de receber os dados de um dispositivo, realizar inferência em tempo real e retornar a decisão do modelo quanto à necessidade de migração. A seguir, descrevemos o funcionamento e os principais componentes dessa API.

3.4.1 Estrutura e Carregamento do Modelo

A API utiliza o framework *PyTorch* para gerenciar o modelo de *Machine Learning*. O modelo treinado é salvo como um arquivo de `checkpoint` no formato `.pth` e carregado no momento da inicialização da API. O processo de carregamento segue os seguintes passos:

1. O arquivo de `checkpoint` é carregado utilizando a função `torch.load()`, que contém os pesos do modelo e a dimensão de entrada utilizada no treinamento.

2. Um objeto da classe `MigrationModel` é instanciado com a mesma arquitetura usada durante o treinamento.

3. Os pesos do modelo são carregados no objeto instanciado por meio do método `load_state_dict()`.

4. O modelo é configurado para o modo de avaliação (`eval()`), garantindo que as camadas de *dropout* e normalização se comportem corretamente durante a inferência.

Esse processo garante que o modelo treinado esteja pronto para ser utilizado em inferências logo após o início da API.

3.4.2 Estrutura do Endpoint e Processamento da Requisição

A API expõe um único endpoint, `/should_migrate`, que aceita requisições HTTP POST contendo as informações necessárias para a decisão de migração. A estrutura do endpoint é descrita a seguir:

Entrada A API recebe os seguintes parâmetros, organizados no corpo da requisição como um objeto JSON:

- **PosX, PosY**: Coordenadas do dispositivo móvel.
- **Direction**: Direção de movimento do dispositivo.
- **Speed**: Velocidade do dispositivo.
- **DistanceToSourceAp**: Distância até o ponto de acesso atual.
- **DistanceToLocalCloudlet**: Distância até a *cloudlet* local.
- **DistanceToClosestCloudlet**: Distância até a *cloudlet* mais próxima que não seja a local.
- **IsMigPoint, IsMigZone**: Indicadores binários que representam se o dispositivo está em um ponto ou zona de migração.

Esses parâmetros são validados utilizando a biblioteca `Pydantic`, garantindo que a API receba apenas dados consistentes e no formato esperado.

Processamento da Inferência Após receber os dados da requisição:

1. Os indicadores `IsMigPoint` e `IsMigZone` são convertidos em valores numéricos (`0` ou `1`).

2. Esses valores são organizados em um vetor de entrada, transformados em um tensor do *PyTorch* e enviados para o modelo carregado.

3. O modelo realiza a inferência e retorna um valor *logit*, que é transformado em probabilidade pela função `sigmoid`.

4. A probabilidade é comparada com um limiar (`0.5`) para determinar se o dispositivo deve migrar (`True`) ou não (`False`).

Saída A API retorna um objeto JSON contendo a decisão:

- `shouldMigrate`: Um valor booleano que indica se o dispositivo deve migrar (`true`) ou não (`false`).

3.4.3 Registro de Logs

A API registra todas as requisições e decisões em um arquivo de logs, permitindo uma análise detalhada do comportamento do modelo e da API. Os logs contêm informações como:

- Dados recebidos na requisição, incluindo coordenadas, direção, velocidade e distâncias.
- Decisão do modelo (`shouldMigrate`).
- Mensagens de erro, caso ocorram falhas no processamento ou carregamento do modelo.

Essa funcionalidade é essencial para o monitoramento e depuração da API, além de possibilitar estudos futuros sobre os padrões de migração.

3.4.4 Flexibilidade da API

Embora o exemplo implementado utilize um modelo simples com apenas duas *features* (`IsMigPoint` e `IsMigZone`), a estrutura da API é altamente flexível. Para utilizar modelos mais complexos:

- Basta alterar as *features* enviadas no corpo da requisição.
- O modelo carregado deve ser adaptado para considerar essas novas *features*.

Essa flexibilidade permite que a API suporte modelos de *Machine Learning* mais sofisticados, como redes neurais profundas ou modelos baseados em *time series*, expandindo sua aplicabilidade.

3.4.5 Conexão com o Simulador *MobFogSim*

A API conecta-se diretamente à estratégia de migração `DecisionMigrationAPI` no *MobFogSim* [1]. Durante a execução do simulador, a estratégia envia os dados de um dispositivo para a API, que retorna a decisão de migração em tempo real. Essa integração demonstra a capacidade do *MobFogSim* [1] de operar em conjunto com modelos externos de *Machine Learning*, permitindo experimentos avançados e flexíveis no contexto de computação em névoa.

3.4.6 Implementação Técnica

A API foi implementada utilizando a biblioteca *FastAPI* e integra um modelo de *Machine Learning* treinado com o framework *PyTorch*. A seguir, são descritas as principais etapas envolvidas no carregamento do modelo e no processamento das requisições.

Carregamento do Modelo O carregamento do modelo treinado é realizado no momento em que a API é inicializada. O processo consiste em:

1. O *checkpoint* do modelo, contendo os pesos treinados e a dimensão de entrada, é carregado utilizando uma função específica do *PyTorch*.
2. Um objeto da classe do modelo é instanciado, configurado com a dimensão de entrada especificada no *checkpoint*.
3. Os pesos salvos no *checkpoint* são carregados no objeto do modelo instanciado, assegurando que o estado do modelo treinado seja restaurado corretamente.
4. O modelo é configurado para o modo de inferência, garantindo que operações específicas, como *dropout*, sejam desativadas durante a execução.

Essas etapas garantem que o modelo treinado esteja pronto para ser utilizado em tarefas de inferência assim que a API estiver em execução. Além disso, o processo inclui mecanismos de registro de log para confirmar o carregamento bem-sucedido ou relatar eventuais erros.

Processamento da Requisição e Inferência A API expõe um endpoint chamado `should_migrate`, responsável por receber os dados de entrada e determinar se um dispositivo deve realizar a migração. O fluxo de processamento da requisição é descrito a seguir:

1. A API recebe os dados enviados pelo cliente em formato JSON. Esses dados incluem informações relevantes sobre o estado do dispositivo, como indicadores binários que representam se o dispositivo está em um ponto ou zona de migração.
2. Os valores binários são convertidos em números inteiros (`0` ou `1`), normalizando os dados para serem usados no modelo.
3. Esses valores são organizados em um vetor de entrada e transformados em um tensor, que é o formato requerido pelo *PyTorch* para operações de inferência.
4. O tensor é passado para o modelo, que realiza a inferência em modo de desativação de gradientes (isto é, sem atualizar os pesos do modelo).
5. O modelo retorna um *logit*, que é transformado em uma probabilidade utilizando a função sigmoidal (*sigmoid*). Essa probabilidade é comparada com um limiar de `0.5` para decidir se a migração deve ser realizada.

Após a inferência, a API retorna a decisão ao cliente em formato JSON, indicando se a migração deve ocorrer (`True`) ou não (`False`). Além disso, o sistema registra os dados recebidos e as decisões tomadas em um arquivo de log, permitindo o monitoramento e a auditoria do comportamento do modelo.

Esses componentes asseguram que a API processe as requisições de maneira eficiente e consistente, conectando diretamente o simulador *MobFogSim* [1] ao modelo de *Machine Learning* treinado, e facilitando a tomada de decisões baseada em inferência em tempo real.

4 Resultados

Os resultados deste trabalho demonstram o sucesso na integração de modelos de Machine Learning (ML) com o simulador *MobFogSim* [1] e na aplicação do treinamento federado utilizando o framework *Flower*. A seguir, são apresentados os principais resultados, organizados em seções que destacam as evidências geradas a partir dos logs do simulador, a eficácia da nova estratégia de migração baseada em API, os registros da API e o impacto no desempenho do simulador.

4.1 Resumo dos Resultados

Foi possível realizar com sucesso a integração de modelos de ML ao simulador *MobFogSim* [1], permitindo decisões baseadas em inferência em tempo real através de uma API. Além disso, foi possível explorar os conceitos de treinamento federado utilizando o framework *Flower* no contexto de sistemas distribuídos.

Os experimentos demonstraram que:

- A coleta de logs detalhados pelo simulador é viável e fornece informações ricas para análise e treinamento de modelos de ML.
- A API desenvolvida respondeu com eficiência às requisições do simulador, suportando decisões dinâmicas em tempo real.
- A estratégia de migração baseada em API foi capaz de integrar as predições do modelo treinado, permitindo o uso de ML nas decisões do simulador.

4.2 Evidências dos Logs para Uso em Treinamento Federado

Os logs gerados pelo simulador durante a execução validam a qualidade dos dados fornecidos para experimentos em treinamento federado. As próximas subseções incluem exemplos de logs gerados pelo simulador:

4.2.1 Latência dos Dispositivos

Os logs detalham a latência observada por cada dispositivo, fornecendo informações como posição, direção, velocidade e servidor conectado. Esses dados podem ser utilizados para prever o comportamento dos dispositivos em diferentes cenários. Exemplo:


```
{
  "Index": 1,
  "Time": 3014.9479970104117,
  "Latency": 14.947997010411655,
  "AppId": "MyApp_vr_game0",
  "SmartThingMyId": 0,
  "ServerCloudletName": "ServerCloudlet76",
  "PosX": 7473,
```

```
  "PosY": 4967,
  "Direction": 5,
  "Speed": 0,
  "SourceAp": "AccessPoint76"
}
```


4.2.2 Decisão de Migração

Os logs de decisão de migração capturam critérios críticos utilizados para a tomada de decisão, como distância para o ponto de acesso, tempo estimado de migração e estado do dispositivo em relação a zonas de migração. Exemplo:


```
{
  "Time": 4000.0,
  "DeviceId": 0,
  "PosX": 7473,
  "PosY": 4967,
  "Direction": 5,
  "Speed": 0,
  "SourceAp": "AccessPoint76",
  "DistanceToSourceAp": 646.623538080698,
  "MigrationTime": 0.0,
  "ShouldMigrate": false,
  "NextServerCloudlet": null,
  "NextAp": null,
  "Reason": "Device is stationary",
  "LocalCloudlet": "ServerCloudlet76",
  "DistanceToLocalCloudlet": 646.623538080698,
  "ClosestCloudlet": "ServerCloudlet64",
  "DistanceToClosestCloudlet": 883.7250703697389,
  "IsMigPoint": false,
  "IsMigZone": false
}
```


4.3 Resultados da Estratégia de Migração Baseada em API

A nova estratégia de migração, que utiliza uma API para decisões baseadas em ML, foi avaliada com sucesso. Durante as simulações, a API foi consultada a cada cenário de decisão, retornando se a migração deveria ou não ser realizada. As próximas subseções incluem exemplos de logs gerados pelo simulador e pela API:

4.3.1 Logs do Simulador com API


```
{
  "Time": 4000.0,
```

```
  "DeviceId": 0,
  "PosX": 7473,
  "PosY": 4967,
  "Direction": 5,
  "Speed": 0,
  "SourceAp": "AccessPoint76",
  "DistanceToSourceAp": 646.623538080698,
  "MigrationTime": 101524.5165877775,
  "ShouldMigrate": false,
  "Reason": "Migration rejected by API"
}
```


4.3.2 Logs da API


```
{
  "Timestamp": "2024-12-08 18:58:29",
  "Level": "INFO",
  "Message": "Received request",
  "Details": {
    "PosX": 7473.0,
    "PosY": 4967.0,
    "Direction": 5.0,
    "Speed": 0.0,
    "DistanceToSourceAp": 646.62,
    "DistanceToLocalCloudlet": 646.62,
    "DistanceToClosestCloudlet": 883.72,
    "IsMigPoint": false,
    "IsMigZone": true
  }
}
{
  "Timestamp": "2024-12-08 18:58:29",
  "Level": "INFO",
  "Message": "Decision",
  "Details": {
    "Decision": false,
    "Request": {
      "PosX": 7473.0,
      "PosY": 4967.0,
      "Direction": 5.0,
      "Speed": 0.0
    }
  }
}
```

4.4 Impacto no Desempenho do Simulador

Os experimentos realizados avaliaram o impacto da integração com a API que contém o modelo de *Machine Learning* (ML) no desempenho do simulador **MobFogSim** [1]. Para isso, dois cenários foram executados na mesma máquina, utilizando os mesmos parâmetros e configurações, alterando apenas a estratégia de migração:

- **Cenário A:** Utiliza a estratégia *LowestLatency*, que busca minimizar a latência entre o dispositivo móvel e a *cloudlet*, sendo inteiramente executado dentro do simulador (sem acionar nada externo).
- **Cenário B:** Adota a estratégia *DecisionMigrationAPI*, que faz chamadas à API para tomar decisões baseadas em um modelo de ML. Esta estratégia *DecisionMigrationAPI* foi construída para se comportar exatamente como a estratégia *LowestLatency*, com o modelo de ML tomando parte da decisão.

Ambos os cenários foram configurados com os seguintes parâmetros no simulador [1]:

- **Cenário A:** 1 210512 0 0 3 11 0 0 0 61
- **Cenário B:** 1 210512 0 3 3 11 0 0 0 61

Os parâmetros configuram o comportamento da simulação, sendo descritos a seguir:

- **Primeiro parâmetro**: 0/1 - Indica se as migrações estão desativadas (0) ou permitidas (1).
- **Segundo parâmetro**: Número inteiro positivo - Define a semente para geração de números aleatórios.
- **Terceiro parâmetro**: 0/1 - Abordagem do ponto de migração é fixa (0) ou baseada na velocidade do usuário (1).
- **Quarto parâmetro**: 0/1/2/3 - Estratégia de migração utilizada: menor latência (0), menor distância entre o usuário e a cloudlet (1), menor distância entre o usuário e o ponto de acesso (2), ou decisão baseada em uma API com Machine Learning (3).
- **Quinto parâmetro**: Número inteiro positivo - Número de usuários na simulação.
- **Sexto parâmetro**: Número inteiro positivo - Largura de banda base da rede entre as cloudlets.
- **Sétimo parâmetro**: 0/1/2 - Política de migração: Migração completa de VM ou fria (0), Migração completa de contêiner (1) ou Migração ao vivo de contêiner (2).
- **Oitavo parâmetro**: Número inteiro não negativo - Previsão de mobilidade do usuário, em segundos.
- **Nono parâmetro**: Número inteiro não negativo - Imprecisão na previsão de mobilidade do usuário, em metros.

- **Décimo parâmetro**: Número inteiro positivo ou negativo - Latência base da rede entre as cloudlets.

A única diferença entre os parâmetros dos dois cenários está no quarto valor, que indica a estratégia de migração utilizada.

Análise do Tempo de Execução O gráfico na Figura 2 apresenta a comparação dos tempos de execução para os dois cenários. O Cenário B, que utiliza a estratégia com a API, apresentou um tempo de execução ligeiramente superior ao Cenário A devido à sobrecarga introduzida pela comunicação com a API. No entanto, essa diferença é muito pequena, demonstrando que a integração com a API é viável e não torna a simulação significativamente mais lenta.

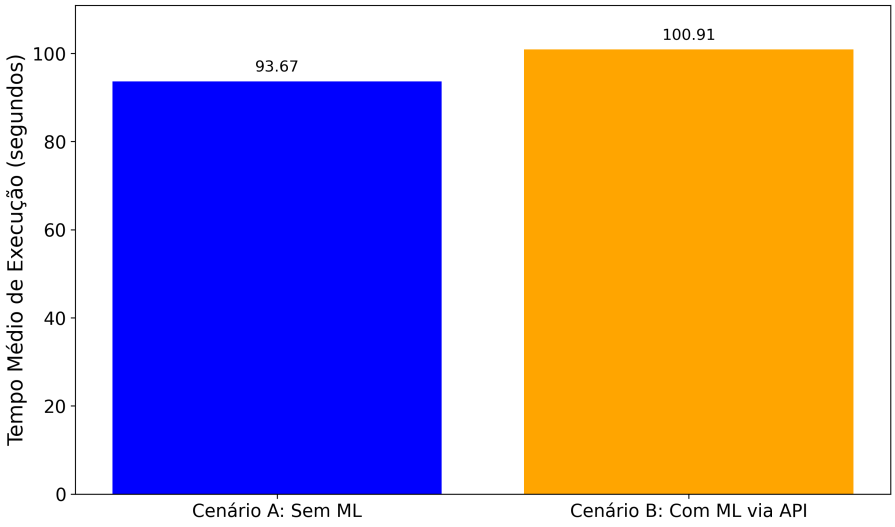

Figura 2: Comparação dos tempos de execução entre os Cenários A e B.

Resultados de Desempenho Os resultados gerais para os dois cenários são apresentados abaixo. Ambos os cenários mantiveram os mesmos aspectos de desempenho no simulador, confirmando que a integração com a API não alterou o comportamento das decisões de migração:

- **Total de migrações:** 8
- **Total de handoffs:** 24
- **Cloudlets diferentes alcançadas:** 24
- **Tempo médio sem conexão:** 1046.39 unidades de tempo
- **Atraso médio após reconexão:** 452305.89 unidades de tempo

- **Tempo médio de migração:** 98272.90 unidades de tempo
- **Percentual de pacotes perdidos:** Cenário A: 7.42% / Cenário B: 7.42%

Os resultados mostram que ambos os cenários apresentaram o mesmo comportamento em relação às decisões de migração e métricas de desempenho. O pequeno aumento no tempo de execução do Cenário B é justificado pela integração com a API, mas não impacta significativamente a viabilidade de utilizar ML para decisões de migração no simulador.

Esses resultados reforçam que a conexão entre o simulador e a API é eficiente e mantém a execução do simulador em níveis aceitáveis, tornando a abordagem viável para futuros experimentos com modelos mais complexos de *Machine Learning*.

5 Conclusão e Próximos Trabalhos

Este trabalho teve como objetivo integrar modelos de Machine Learning (ML) ao simulador *MobFogSim* [1], ampliando suas funcionalidades para suportar estudos de treinamento federado em cenários de computação em névoa com dispositivos móveis. Para alcançar esse objetivo, foram desenvolvidos três módulos principais: ajustes no *MobFogSim* [1] para geração de logs detalhados, implementação de um treinamento federado utilizando o framework *Flower* e criação de uma API que permite a integração de modelos treinados ao simulador para tomada de decisões dinâmicas durante as simulações.

Os resultados demonstraram que foi possível realizar a integração entre o *MobFogSim* [1] e os modelos de ML, validando a viabilidade do uso de treinamento federado para o desenvolvimento de modelos globais de aprendizado em um ambiente distribuído. Além disso, a API desenvolvida provou ser eficaz ao conectar o simulador a modelos externos, possibilitando a utilização de inferências em tempo real para decisões baseadas em ML.

Com essas modificações, o *MobFogSim* [1] está agora preparado para receber modelos mais complexos de *Machine Learning*, ampliando seu potencial como ferramenta de pesquisa em computação em névoa, mobilidade e aprendizado federado. Pesquisadores podem utilizar essa estrutura para explorar novos cenários e validar modelos federados em contextos variados, aprofundando os estudos sobre privacidade, eficiência e desempenho em sistemas distribuídos.

Este trabalho abre caminho para mais linhas de pesquisa futuras, como a aplicação de estratégias otimizadas para migração de recursos em *fog computing* e a validação de diferentes abordagens de treinamento federado, fortalecendo a conexão entre *Machine Learning* e simulação de mobilidade em sistemas distribuídos.

Referências

[1] Carlo Puliafito, Diogo M. Gonçalves, Márcio M. Lopes, Leonardo L. Martins, Edmundo Madeira, Enzo Mingozzi, Omer Rana, and Luiz F. Bittencourt, *MobFogSim: Simulation of mobility and migration for fog computing*, Simulation Modelling Practice and Theory, vol. 101, pp. 102062, Elsevier (2020).

[2] L. Codeca, R. Frank, T. Engel, *Luxembourg SUMO Traffic (LuST) Scenario: 24 Hours of Mobility for Vehicular Networking Research*, Proceedings of the IEEE Vehicular Networking Conference (VNC), 2015, pp. 1–8. Available at: *https://doi.org/10.1109/VNC.2015.7385539*.

[3] Chen Zhang, Yu Xie, Hang Bai, Bin Yu, Weihong Li, and Yuan Gao, *A survey on federated learning*, Knowledge-Based Systems, vol. 216, pp. 106775, Elsevier, 2021.

[4] Thiago dos Santos Solera, Pedro Strambeck Nogueira, Allan Mariano de Souza, Joahannes Bruno Dias da Costa, and Luiz Fernando Bittencourt, *Análise de performance e viabilidade de algoritmos de aprendizado federado utilizando Flower*, Institute of Computing, University of Campinas, Technical Report IC-PFG-24-06, July 2024.

[5] Flower Framework, *Documentation*, accessed on December 8, 2024. Available at: *https://flower.dev/docs/*.

[6] FastAPI, *Documentation*, accessed on December 8, 2024. Available at: *https://fastapi.tiangolo.com/*.

[7] PyTorch, *Documentation*, accessed on December 8, 2024. Available at: *https://pytorch.org/docs/*.