



PROTOTIPAGEM DE UM CLIENTE EMBARCADO PARA  
EXECUÇÃO DE APRENDIZADO FEDERADO EM VEÍCULOS:  
UM ESTUDO DE CASO EM BATERIAS ELÉTRICAS

Vitor Rayol Taranto

Projeto de Graduação apresentado ao Curso de Engenharia de Computação e Informação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Miguel Elias Mitre Campista

Rio de Janeiro

Março de 2026

PROTOTIPAGEM DE UM CLIENTE EMBARCADO PARA  
EXECUÇÃO DE APRENDIZADO FEDERADO EM VEÍCULOS:  
UM ESTUDO DE CASO EM BATERIAS ELÉTRICAS

Vitor Rayol Taranto

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO  
DE ENGENHARIA DE COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA PO-  
LITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO  
PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU  
DE ENGENHEIRO DE COMPUTAÇÃO E INFORMAÇÃO

Autor:

---

Vitor Rayol Taranto

Orientador:

---

Prof. Miguel Elias Mitre Campista, D.Sc.

Examinador:

---

Prof. Rodrigo de Souza Couto, D.Sc.

Examinador:

---

Prof. Luís Henrique Maciel Kosmowski Costa, Dr.

Rio de Janeiro

Março de 2026

## Declaração de Autoria e de Direitos

Eu, *Vitor Rayol Taranto*, autor da monografia *Prototipagem de um Cliente Embarcado para Execução de Aprendizado Federado em Veículos: Um Estudo de Caso em Baterias Elétricas*, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.
2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e idéias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.
3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.
4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.
5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.
6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.
7. Por ser verdade, firmo a presente declaração.

---

Nome do aluno

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

Escola Politécnica - Departamento de Eletrônica e de Computação

Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária

Rio de Janeiro - RJ CEP 21949-900

Este exemplar é de propriedade da Universidade Federal do Rio de Janeiro, que poderá incluí-lo em base de dados, armazenar em computador, microfilmear ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es).

## AGRADECIMENTO

A elaboração deste Trabalho de Conclusão de Curso representou uma etapa de intenso aprendizado e superação. Ao longo de toda essa trajetória, contei com o suporte indispensável de pessoas extraordinárias, a quem direciono a minha mais profunda gratidão.

Em primeiro lugar, expresso o meu amor e agradecimento infinito aos meus pais, Gisele Barbosa Vianna Rayol Taranto e Glauco Nery Taranto. Todo o apoio incondicional, os ensinamentos diários e a base familiar que vocês me proporcionaram foram o alicerce para que eu pudesse alcançar esta conquista. Este trabalho reflete diretamente os valores que aprendi em casa.

Ao professor Miguel Elias Mitre Campista, deixo o meu sincero reconhecimento por toda a excelência na orientação, pela paciência e pelos incentivos constantes. Sou imensamente grato pela confiança que depositou em meu potencial desde a minha primeira iniciação científica, além das oportunidades em que atuei como seu monitor durante três períodos letivos. Ter construído essa parceria acadêmica prévia, sob o escopo deste mesmo grande projeto, foi crucial para o meu desenvolvimento como pesquisador e para a maturidade desta pesquisa.

Por fim, estendo meus agradecimentos aos amigos, aos colegas de graduação e a todos os professores que cruzaram o meu caminho durante o curso. As trocas de conhecimento, os momentos de descontração e o companheirismo nas horas de dificuldade foram fundamentais para tornar esta jornada inesquecível e possível.

## RESUMO

O crescente aumento da frota de veículos elétricos traz à tona a necessidade de monitoramento contínuo da Saúde da Bateria (*State of Health* – SOH). Tradicionalmente, a predição desse parâmetro baseia-se na transmissão de métricas veiculares sensíveis para processamento em nuvem, o que gera sobrecarga nas redes de comunicação e sérias vulnerabilidades de privacidade. Visando contornar esses problemas, este trabalho concentra-se na prototipagem e na demonstração prática de um cliente embarcado de Aprendizado Federado. O foco central do projeto reside no desenvolvimento de uma interface visual autônoma, concebida para fornecer *feedback* em tempo real ao condutor de forma totalmente passiva. Durante a pesquisa, diversas abordagens de integração foram exploradas, evidenciando as rigorosas restrições arquiteturais das plataformas automotivas comerciais. Como solução viável, o sistema foi implementado localmente utilizando hardware de baixo custo da família Raspberry Pi, integrado de forma transparente ao ecossistema do Android Auto. Essa arquitetura descentralizada viabilizou a criação de um painel de telemetria minimalista e com recarregamento automático, garantindo que o motorista não sofra distrações com interações manuais. A demonstração funcional valida a eficácia da interface desenvolvida, na qual o treinamento ocorre no próprio veículo e os dados brutos permanecem retidos fisicamente, comprovando a viabilidade técnica de aliar inteligência artificial distribuída e segurança cibernética em painéis multimídia veiculares.

Palavras-Chave: Aprendizado Federado. Android Auto. Interface Visual. Prototipagem. Veículos Elétricos.

## ABSTRACT

The growing increase in the electric vehicle fleet highlights the need for continuous monitoring of Battery State of Health (SOH). Traditionally, predicting this parameter relies on transmitting sensitive vehicular metrics for cloud processing, which creates communication network overload and severe privacy vulnerabilities. Aiming to bypass these issues, this work focuses on the prototyping and practical demonstration of an embedded Federated Learning client. The core focus of the project lies in the development of an autonomous visual interface, designed to provide real-time feedback to the driver in a completely passive manner. During the research, several integration approaches were explored, exposing the strict architectural restrictions of commercial automotive platforms. As a viable solution, the system was implemented locally using low-cost hardware from the Raspberry Pi family, seamlessly integrated into the Android Auto ecosystem. This decentralized architecture enabled the creation of a minimalist telemetry dashboard with automatic reloading, ensuring that the driver is not distracted by manual interactions. The functional demonstration validates the efficacy of the developed interface, in which training occurs within the vehicle itself and raw data remains physically retained, proving the technical viability of combining distributed artificial intelligence and cybersecurity in vehicular multimedia dashboards.

Key Words: Federated Learning. Android Auto. Visual Interface. Prototyping. Electric Vehicles.

## SIGLAS

AAOS : Android Automotive OS

AOSP : Android Open Source Project

AVADiP : Aplicações Veiculares com Aprendizado Distribuído e Manutenção de Privacidade

BLE : Bluetooth Low Energy

BMS : Sistema de Gerenciamento da Bateria (do inglês, Battery Management System)

CAN : Controller Area Network

CPU : Unidade Central de Processamento (do inglês, Central Processing Unit)

CSS : Cascading Style Sheets

FedAvg : Federated Averaging

GATT : Generic Attribute Profile

GPS : Sistema de Posicionamento Global (do inglês, Global Positioning System)

GPU : Unidade de Processamento Gráfico (do inglês, Graphics Processing Unit)

GTA : Grupo de Teleinformática e Automação

HMI : Interface Homem Máquina (do inglês, Human Machine Interface)

HTML : HyperText Markup Language

IEEE : Institute of Electrical and Electronics Engineers

IID : Independentes e Identicamente Distribuídas (do inglês, Independent and Identically Distributed)

MSE : Erro Quadrático Médio (do inglês, Mean Squared Error)

NPU : Neural Processing Unit

OBD II : On Board Diagnostics

RAM : Random Access Memory

SOH : Estado de Saúde (do inglês, State of Health)

UFRJ : Universidade Federal do Rio de Janeiro

USB : Universal Serial Bus

VRU : Usuários Vulneráveis da Via (do inglês, Vulnerable Road Users)

# Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Estado da Arte</b>	<b>4</b>
2.1	Aprendizado Federado Automotivo . . . . .	4
2.2	Implementação de Telemetria e Computação de Borda . . . . .	5
<b>3</b>	<b>Fundamentação Teórica</b>	<b>8</b>
3.1	Computação de Borda . . . . .	8
3.2	Aprendizado Federado: Conceitos e Arquitetura . . . . .	9
3.3	Baterias de Veículos Elétricos e Parâmetros de Saúde (SOH) . . . . .	11
3.4	Integração de Aplicações de SOH em Interfaces Veiculares . . . . .	12
<b>4</b>	<b>Metodologia e Evolução da Arquitetura</b>	<b>15</b>
4.1	Fase 1: Arquitetura Baseada em Gateway Móvel (Bluetooth) . . . . .	16
4.1.1	Arquitetura de Comunicação e Fluxo de Dados do Sistema . . . . .	17
4.1.2	Implementação do Aplicativo Gateway . . . . .	18
4.1.3	Limitações e Motivação para Evolução . . . . .	18
4.2	Fase 2: Arquitetura Embarcada Nativa (Android Auto) . . . . .	18
4.2.1	Seleção da Plataforma de Software . . . . .	18
4.2.2	Desafios de Implementação no Raspberry Pi Zero 2 W . . . . .	19
4.2.3	Migração para Raspberry Pi 4 e Consolidação do Hardware . . . . .	20
4.2.4	Configurações Críticas de Sistema e Rede . . . . .	20
4.3	Fase 3: O Desafio da Aceleração e a Consolidação do Ambiente . . . . .	22
4.3.1	Incompatibilidade Arquitetural e Falha na Virtualização . . . . .	22
4.3.2	O Pivô para Predição de SOH (Battery Health) . . . . .	22
4.3.3	Desafios de Software em Ambiente Legado . . . . .	23

4.3.4	Desacoplamento da Interface de Usuário (HMI Web) . . . . .	24
<b>5</b>	<b>Desenvolvimento e Implementação</b>	<b>28</b>
5.1	Configuração do Ambiente e Otimização para Hardware Heterogêneo	28
5.2	Implementação dos Clientes de Aprendizado Federado . . . . .	31
5.2.1	Aquisição e Particionamento dos Dados de Telemetria . . . . .	32
5.3	Desenvolvimento da Interface de Telemetria . . . . .	38
5.4	Orquestração da Comunicação e Estratégia de Agregação . . . . .	39
<b>6</b>	<b>Resultados e Análise Experimental</b>	<b>42</b>
6.1	Cenário de Testes e Infraestrutura . . . . .	42
6.2	Validação Funcional da Arquitetura de Rede . . . . .	43
6.3	Validação da Interface Web em Tempo Real . . . . .	45
<b>7</b>	<b>Conclusão</b>	<b>47</b>
7.1	Considerações Finais . . . . .	47
7.2	Trabalhos Futuros . . . . .	48
	<b>Bibliografia</b>	<b>50</b>
	<b>A Código Fonte do Cliente Federado</b>	<b>53</b>
	<b>B Código Fonte do Servidor de Orquestração</b>	<b>60</b>

# Lista de Figuras

3.1	Arquitetura padrão do Aprendizado Federado. Fonte: GeeksforGeeks [1] . . . . .	10
4.1	Exemplo do protótipo inicial descartado: <i>bounding boxes</i> para detecção de VRUs. Fonte: Canal do Youtube Grupo de Teleinformática e Automação - GTA [2] . . . . .	16
4.2	Fluxo de dados da arquitetura com smartphone como intermediário. . . . .	17
4.3	Montagem física do dispositivo de borda com display touchscreen . . . . .	21
4.4	Flask + Flower rodando simultaneamente . . . . .	26
4.5	Painel de telemetria servido localmente via rede sem fio . . . . .	27
5.1	Arquitetura de software e hardware do sistema embarcado . . . . .	29
5.2	Disposição física dos dispositivos de borda e infraestrutura de rede local. . . . .	30
5.3	Fluxo de processamento do dado: da telemetria local à extração de parâmetros federados. . . . .	32
6.1	Evolução do Erro Quadrático Médio (MSE) e convergência da predição de Saúde da Bateria (SOH) para ambos os clientes . . . . .	43
6.2	Média e desvio padrão do consumo de CPU e RAM durante 5 execuções do ciclo de treinamento nos dispositivos de borda . . . . .	44
6.3	Logs de execução simultânea: convergência do MSE e estabilização do SOH nos dois clientes (Raspberry Pi 4 e Pi Zero) . . . . .	45
6.4	Estados da interface web durante o ciclo de vida do treinamento federado . . . . .	46

# Capítulo 1

## Introdução

A transição global para a mobilidade sustentável impulsionou a adoção em massa de veículos elétricos na última década [3]. O componente central que dita a viabilidade, o custo e a segurança desses veículos é o pacote de baterias de íons de lítio. Devido ao estresse químico e térmico, essas baterias sofrem degradação natural com o passar do tempo e dos ciclos de uso, tornando o monitoramento do Estado de Saúde (*State of Health* – SOH) uma tarefa crítica para a engenharia automotiva. Para calcular o SOH de forma precisa e contornar as limitações dos métodos empíricos tradicionais, a literatura recente demonstra que os sistemas modernos utilizam cada vez mais algoritmos de inteligência artificial [4, 5]. Esses modelos preditivos aprendem a partir dos dados gerados pelo próprio automóvel, englobando perfis exatos de tensão, corrente e temperatura capturados durante as sessões de recarga e de condução.

Historicamente, a construção e a atualização desses modelos preditivos exigem o envio contínuo de toda a telemetria veicular para processamento em centros de dados centralizados. Essa abordagem baseada em nuvem impõe desafios à infraestrutura de comunicação, pois o volume de dados pode sobrecarregar as redes celulares, exigindo conectividade ininterrupta. Mais alarmante, entretanto, é a questão da privacidade. As grandezas elétricas trafegam atreladas às coordenadas de posicionamento global e aos horários de rotina do motorista. A literatura recente sobre segurança em veículos conectados alerta que transmitir essas informações brutas para servidores de terceiros expõe a intimidade do condutor e cria vulnerabilidades críticas de segurança cibernética, sujeitando o sistema a interceptações e vazamen-

tos [6].

Diante desse cenário restritivo, a comunidade acadêmica tem buscado arquiteturas descentralizadas que preservem a privacidade das fontes de dados. Toda a motivação e o desenvolvimento desta pesquisa inserem-se no escopo do projeto Aplicações Veiculares com Aprendizado Distribuído e Manutenção de Privacidade (AVADiP)<sup>1</sup>, conduzido pelo Grupo de Teleinformática e Automação (GTA) da Universidade Federal do Rio de Janeiro. O projeto AVADiP tem como premissa viabilizar ecossistemas veiculares inteligentes capazes de aprender padrões complexos de forma colaborativa, utilizando o Aprendizado Federado para garantir que os dados brutos de telemetria e a rotina dos condutores nunca sejam disponibilizados a terceiros. O Aprendizado Federado, introduzido originalmente por pesquisadores do Google [7], consolida-se como a solução técnica ideal para esse impasse. Trata-se de um paradigma que inverte o fluxo tradicional de processamento: em vez de enviar os dados dos clientes para o servidor, o sistema transmite o modelo matemático para ser treinado localmente nos dispositivos. No contexto de redes móveis e de borda, pesquisas recentes conduzidas no próprio GTA comprovaram que essa arquitetura é fundamental para manter o desempenho preditivo ao mesmo tempo em que contorna os gargalos de comunicação [8].

Alinhado diretamente a essa visão, o presente trabalho concentra esforços na engenharia de software e hardware necessária para embarcar um cliente de aprendizado federado diretamente no veículo. O objetivo principal consiste em desenvolver, integrar e demonstrar a viabilidade prática desse sistema embarcado, o qual treina modelos preditivos localmente e atua como uma prova de conceito real das premissas do projeto AVADiP. Para alcançar esse propósito, o projeto estabeleceu metas claras: investigar as limitações de desenvolvimento impostas por plataformas automotivas comerciais, como o Android Auto; estruturar um ambiente de execução local utilizando microcomputadores da família Raspberry Pi; e desenvolver uma aplicação veicular provida de uma interface gráfica minimalista que traduza as predições do modelo em um painel de telemetria, fornecendo feedback valor do SOH do treinamento em tempo real sem gerar distrações visuais ou exigir interações manuais do motorista.

---

<sup>1</sup>Página oficial do projeto: <https://gta.ufrj.br/avadip/>.

A relevância desta pesquisa reside na entrega de uma solução tecnológica abrangente, que vai desde a concepção da arquitetura de software até a validação experimental de um protótipo físico. Por um lado, a aplicação do Aprendizado Federado soluciona o dilema da privacidade, retendo os dados sensíveis no veículo e viabilizando o treinamento de modelos preditivos diretamente na borda. Por outro lado, a materialização do sistema proposto em um hardware de baixo custo comprova a viabilidade de integrar essa inteligência distribuída de forma transparente ao ecossistema do Android Auto. Os resultados experimentais obtidos atestam a eficácia da abordagem, demonstrando que o ambiente automotivo pode executar processamento complexo de inteligência artificial de forma concorrente a painéis de diagnóstico seguros e visualmente integrados, eliminando em definitivo a dependência contínua de processamento na nuvem ou em dispositivos móveis intermediários.

Para apresentar o desenvolvimento e a validação desta solução, este documento está estruturado da seguinte forma:

- o Capítulo 2 revisa o estado da arte referente ao monitoramento de baterias e redes federadas automotivas,
- o Capítulo 3 estabelece a fundamentação teórica sobre os conceitos tecnológicos abordados,
- o Capítulo 4 detalha a metodologia de pesquisa e a arquitetura do sistema proposto,
- o Capítulo 5 descreve a implementação técnica do protótipo, englobando o software e o hardware,
- o Capítulo 6 apresenta os resultados experimentais e a demonstração prática da interface,
- o Capítulo 7 conclui o trabalho, resumizando as contribuições e apontando caminhos para desenvolvimentos futuros.

# Capítulo 2

## Estado da Arte

Este capítulo revisa a literatura pertinente ao monitoramento de baterias veiculares e à aplicação do Aprendizado Federado no contexto automotivo. O objetivo é contextualizar o presente trabalho frente aos avanços recentes e evidenciar a lacuna tecnológica referente à implementação prática em dispositivos de borda embarcados.

### 2.1 Aprendizado Federado Automotivo

A aplicação do Aprendizado Federado na indústria automotiva tem ganhado destaque como uma solução para o dilema entre a necessidade de dados massivos e a garantia de privacidade dos condutores. Pesquisas recentes demonstraram a viabilidade de treinar modelos preditivos precisos sem a necessidade de transferir informações sensíveis de telemetria para servidores em nuvem [9, 10].

Estudos comparativos na literatura avaliam o desempenho de arquiteturas centralizadas em contraste com abordagens federadas para a estimação da Saúde da Bateria (SOH) [9]. Os resultados indicam que o treinamento federado consegue preservar a privacidade dos dados mantendo-os localmente e, ainda assim, alcançar métricas de precisão muito próximas às da abordagem centralizada. Essa equivalência demonstra que a perda de desempenho é mínima em face do ganho expressivo em segurança cibernética e soberania de dados. Paralelamente, a literatura recente explora estratégias otimizadas de agregação de pesos para manter a convergência do modelo mesmo diante da alta heterogeneidade dos dados de condução presentes em frotas veiculares [10].

Um estudo de relevância direta para este trabalho foi conduzido pelo Grupo de Teleinformática e Automação [11], que propôs um framework federado focado na predição de SOH em veículos elétricos reais. Os autores enfrentaram o desafio da heterogeneidade estatística, em que cada bateria sofre degradação sob condições distintas devido aos diferentes perfis de condução e rotinas de recarga dos usuários. O estudo comprovou que a orquestração federada consegue manter a convergência do modelo, mesmo em cenários extremos em que até 85% dos pares de clientes possuem distribuições de dados fortemente heterogêneas.

Apesar dos avanços, validações experimentais na área, incluindo o estudo de [11], frequentemente concentram a execução dos modelos em *clusters* de servidores equipados com unidades de processamento gráfico (GPUs). Nesses cenários, os nós da rede simulam o treinamento local utilizando recursos computacionais que não refletem as limitações de processamento e energia de um automóvel. Para preencher essa lacuna, a principal contribuição deste projeto consiste em adaptar essas arquiteturas de alto desempenho à realidade dos dispositivos com recursos limitados. Essa adaptação é viabilizada por meio da adoção de algoritmos de baixo custo computacional — priorizando modelos lineares em detrimento de redes neurais profundas — e do desacoplamento de processos, desenvolvendo um cliente leve o suficiente para operar de forma nativa e sem gargalos no hardware embarcado do veículo..

## 2.2 Implementação de Telemetria e Computação de Borda

Os sistemas tradicionais de gerenciamento de baterias (*Battery Management System* – BMS) operam de forma isolada dentro do automóvel. Historicamente, para que algoritmos de inteligência artificial pudessem prever a degradação a longo prazo, esses dados brutos precisavam ser transmitidos continuamente através de redes celulares para os centros de dados das montadoras. Como consolida a literatura do setor [9], essa abordagem centralizada gera gargalos de conectividade e preocupações severas quanto à privacidade, expondo o rastreamento da rotina de carregamento dos usuários.

A adoção da Computação de Borda Veicular (*Vehicular Edge Computing* –

VEC) propõe trazer o processamento da nuvem para o próprio automóvel. Esse paradigma permite que o veículo execute tarefas de análise de dados localmente, mitigando a dependência de uma conexão estável. Pesquisas desenvolvidas na academia brasileira [12] ressaltam que aplicações para veículos conectados exigem tolerância à conectividade intermitente e adaptação aos recursos computacionais limitados presentes na borda da rede.

A integração de centrais multimídia e computadores de placa única, como os dispositivos da família Raspberry Pi utilizados nesta pesquisa, permite a aplicação prática da Computação de Borda Veicular. Dessa forma, o veículo deixa de ser um mero remetente de informações e passa a atuar como um nó de processamento local. Ao associar essa arquitetura de hardware a métodos de Aprendizado Federado, o sistema viabiliza o treinamento local de modelos que, futuramente, subsidiam a aplicação de diagnóstico do SOH, colaborando com parâmetros globais sem a necessidade de expor dados brutos. Essa transição para o processamento embarcado demonstra ser uma alternativa viável, respeitando as restrições de consumo energético e os limites de orçamento comuns na indústria automotiva [13].

Além dos desafios de infraestrutura e comunicação, a própria modelagem e predição da Saúde da Bateria (SOH) representam um obstáculo técnico significativo. A dificuldade reside na natureza altamente não-linear da degradação das células de íons de lítio, que varia drasticamente conforme os perfis de condução, rotinas de recarga e flutuações de temperatura enfrentados por cada veículo. Para lidar com essa complexidade temporal e realizar a inferência do SOH, o estado da arte tem adotado arquiteturas complexas de aprendizado profundo (*Deep Learning*). Como exemplo, o próprio estudo supramencionado do Grupo de Teleinformática e Automação [11] propôs o modelo SOH-MPRSE, um preditor de múltiplos passos que extrai características temporais profundas dos perfis de tensão, corrente e temperatura. A arquitetura utiliza uma rede recorrente GRU (*Gated Recurrent Unit*) na fase intra-ciclo e uma rede LSTM (*Long Short-Term Memory*) na fase inter-ciclo para processar as dependências de degradação a longo prazo.

Embora modelos profundos alcancem excelente precisão na captura do envelhecimento da bateria, eles exigem um volume expressivo de poder computacional e uso de memória. Esse cenário reforça a motivação deste trabalho: enquanto a

literatura busca redes neurais cada vez mais densas para a previsão de SOH, a implementação real na borda veicular (como em dispositivos Raspberry Pi emulando o *Android Auto*) impõe diversas restrições de processamento. Portanto, a simplificação algorítmica intencional, substituindo redes recorrentes pesadas por modelos de regressão linear eficientes, combinada ao desacoplamento de processos em tela e plano de fundo, consolida-se como o caminho pragmático para viabilizar previsões locais contínuas sem comprometer os limitados recursos do painel do automóvel.

# Capítulo 3

## Fundamentação Teórica

Como estabelecido anteriormente, o objetivo central deste trabalho consiste em desenvolver um cliente embarcado de Aprendizado Federado capaz de operar diretamente no hardware do veículo, realizando a predição da saúde da bateria de forma descentralizada e exibindo os resultados em uma interface gráfica. Para fundamentar a engenharia dessa solução, este capítulo apresenta as bases teóricas e tecnológicas que sustentam o projeto. A estruturação dos conceitos abrange desde os paradigmas de aprendizado de máquina distribuído até as métricas de degradação das baterias elétricas, culminando nas estratégias de integração visual e no desacoplamento de interfaces para o painel do veículo.

### 3.1 Computação de Borda

O paradigma computacional predominante na última década baseou-se fortemente na centralização de recursos em infraestruturas de nuvem. Nesse modelo, os dispositivos terminais atuam primariamente como coletores de dados, transferindo grandes volumes de informações através da rede para serem processados por servidores remotos de alta capacidade. Contudo, a proliferação de dispositivos da Internet das Coisas e a evolução dos sistemas veiculares evidenciaram as limitações críticas dessa arquitetura centralizada, fundamentalmente no que tange à privacidade. A transferência contínua de dados brutos de telemetria exige que informações sensíveis sejam armazenadas e manipuladas em *datacenters* de terceiros. Muito além de eventuais vulnerabilidades de interceptação durante a transmissão, é o armaze-

namento persistente e o uso irrestrito desses dados brutos diretamente na nuvem que levantam as mais graves preocupações de segurança e soberania, motivando de forma definitiva a transição para métodos de processamento local e descentralizado.

Para mitigar esses gargalos, a Computação de Borda propõe tirar o processamento dos servidores e trazê-los para os nós da rede [14]. Em vez de enviar os dados para onde o poder computacional reside, a inteligência é deslocada para as extremidades da rede, ou seja, para próximo de onde o dado é gerado. No contexto veicular, o próprio carro assume o papel de um nó de processamento na borda.

Ao processar as informações de telemetria localmente no hardware embarcado do veículo, a Computação de Borda reduz drasticamente a latência de resposta e diminui a dependência de conexões ininterruptas. Mais importante ainda, manter os dados retidos fisicamente no cliente é o fundamento essencial para garantir a preservação de privacidade exigida pelo Aprendizado Federado.

## **3.2 Aprendizado Federado: Conceitos e Arquitetura**

O Aprendizado Federado emergiu como uma solução arquitetural promissora para conciliar o treinamento de modelos de inteligência artificial com os rigorosos requisitos de privacidade do mundo moderno [15]. Em sua essência, trata-se de um paradigma de aprendizado de máquina descentralizado que permite o treinamento colaborativo entre múltiplos dispositivos sem a necessidade de compartilhar dados locais. Quando aplicado ao escopo deste trabalho, essa abordagem permite que uma frota de veículos atue de forma conjunta, garantindo que os dados brutos de uso da bateria nunca deixem o automóvel.

A arquitetura padrão do Aprendizado Federado opera através de ciclos iterativos de comunicação entre um servidor coordenador e múltiplos dispositivos clientes, conforme ilustrado na Figura 3.1.

O fluxo de execução obedece a uma mecânica rigorosa: cada cliente treina um modelo de aprendizado de máquina localmente utilizando exclusivamente o seu próprio conjunto de dados retido em armazenamento físico e, em seguida, transmite apenas os parâmetros do modelo treinado para o servidor central.

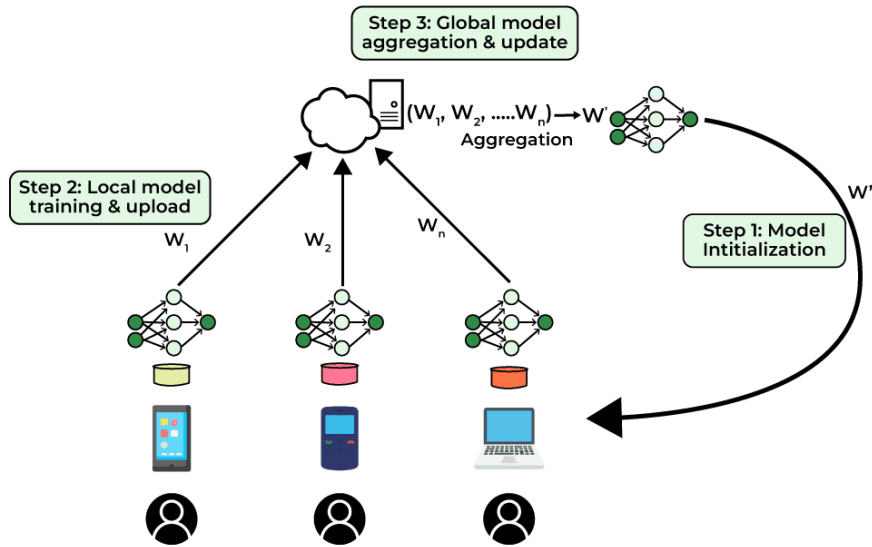


Figura 3.1: Arquitetura padrão do Aprendizado Federado. Fonte: GeeksforGeeks [1]

Após o recebimento das contribuições individuais, o servidor agrega esses parâmetros para produzir um modelo global aprimorado, o qual é prontamente redistribuído aos clientes para o início da próxima rodada de treinamento. A estratégia de agregação mais consolidada na literatura é a FedAvg (*Federated Averaging*), que calcula a média ponderada dos parâmetros recebidos. Nessa abordagem, a contribuição de cada cliente é ponderada de acordo com o volume de dados utilizado em seu treinamento local, ou seja, clientes com mais amostras de dados possuem maior peso na atualização do modelo global [7].

O principal obstáculo técnico enfrentado por essa arquitetura no ambiente automotivo reside na natureza dos dados gerados. Além de serem volumosos e sensíveis em termos de privacidade, os dados coletados pelos veículos são não independentes e identicamente distribuídos (não-IID). Hábitos de condução distintos, variações climáticas regionais e protocolos de recarga variados produzem conjuntos de dados heterogêneos entre os clientes [11], exigindo que o algoritmo federado possua robustez para alcançar a convergência sem causar o esquecimento catastrófico do modelo global.

Para viabilizar a implementação prática dessa arquitetura, este trabalho utiliza o *framework* Flower [16]. Trata-se de uma biblioteca de código aberto projetada

para estruturar e escalar sistemas de Aprendizado Federado. A principal vantagem técnica do Flower é a sua independência estrutural, permitindo a integração com ferramentas consolidadas de aprendizado de máquina, como PyTorch ou TensorFlow. No escopo deste projeto, a plataforma destaca-se por suportar a execução em hardwares de borda, como o Raspberry Pi, facilitando a transição de um ambiente de simulação em bancada para a implantação em dispositivos reais de forma eficiente.

### 3.3 Baterias de Veículos Elétricos e Parâmetros de Saúde (SOH)

O desempenho da frota global de veículos elétricos depende diretamente de seus pacotes de baterias de íons de lítio. Devido à complexidade das reações eletroquímicas internas, esses componentes sofrem degradação contínua e irreversível a cada ciclo de carga e descarga, impactando a autonomia do automóvel.

Nesse cenário, estimar com precisão o Estado de Saúde (SOH) é um requisito técnico necessário. O SOH fornece uma medida quantitativa da condição atual da bateria em relação ao seu estado original. Na prática, a forma mais comum de calcular essa métrica é pela razão direta entre a capacidade de armazenamento atual e a capacidade nominal de fábrica:

$$SOH = \frac{C_{atual}}{C_{nominal}} \times 100\%$$

onde  $C_{atual}$  representa a capacidade máxima de carga que a bateria consegue armazenar no seu estado corrente, e  $C_{nominal}$  é a capacidade original de projeto.

A definição do limite aceitável de SOH é arbitrária e varia conforme a criticidade da aplicação [17]. Enquanto sistemas estacionários menos exigentes podem operar com baterias a 50% de saúde, o primeiro ciclo de vida em veículos elétricos geralmente encerra-se quando o SOH atinge a faixa de 70% a 80%. Abaixo desse limiar, a bateria passa a sofrer quedas instantâneas de tensão devido ao aumento de sua resistência interna, o que impede a eletrônica de potência do automóvel de sustentar a operação com segurança.

A necessidade de uma arquitetura distribuída e intensiva em dados surge do

próprio desafio de monitoramento veicular: a Saúde da Bateria (SOH) não é uma grandeza física mensurável diretamente por sensores, mas sim uma variável de estado interna. Como a sua determinação não é uma medida exata, mas sim uma estimativa, modelos preditivos exigem um volume massivo e diversificado de dados operacionais para aprender os padrões de degradação. Para viabilizar o acesso seguro a esses dados e a implementação prática da arquitetura, este trabalho utiliza o *framework* Flower [16]. Trata-se de uma biblioteca de código aberto projetada para estruturar e escalar sistemas de Aprendizado Federado. A principal vantagem técnica do Flower é a sua independência estrutural, permitindo a integração com ferramentas consolidadas de aprendizado de máquina, como PyTorch ou TensorFlow. No escopo deste projeto, a plataforma destaca-se por suportar a execução em *hardwares* de borda, como o Raspberry Pi, facilitando a transição de um ambiente de simulação em bancada para a implantação em dispositivos reais de forma eficiente.

### 3.4 Integração de Aplicações de SOH em Interfaces Veiculares

Para que as previsões de SOH detalhadas na seção anterior sejam processadas localmente e exibidas ao condutor, é necessária uma infraestrutura computacional robusta no automóvel. Historicamente projetada para funções básicas de rádio, a central multimídia transformou-se no principal concentrador de dados da cabine. Essa evolução converteu o painel em um sofisticado centro de processamento e interface veicular. Veículos modernos, como o Tesla Model 3, são equipados com processadores de múltiplos núcleos, dezenas de gigabytes de memória RAM e unidades de processamento neural dedicadas, formando uma arquitetura capaz de executar trilhões de operações por segundo.

Apesar dessa alta capacidade embarcada, a implementação de aplicações mais complexas, como o treinamento de modelos preditivos da bateria, esbarra em barreiras impostas pelo próprio sistema operacional. Atualmente, o desenvolvimento de aplicativos para painéis veiculares é fortemente padronizado por gigantes de tecnologia. Embora a Apple ofereça o *Apple CarPlay* sob um modelo de projeção

amplamente adotado pelo mercado, este trabalho foca nas soluções do Google, que se dividem em duas abordagens principais. O *Android Auto* tradicional opera sob um modelo de projeção. É importante destacar que isso não significa apenas espelhar a tela do celular; o sistema utiliza o processamento do smartphone do usuário para renderizar uma interface gráfica nativa e adaptada especificamente para a tela do carro. Em contrapartida, o *Android Automotive OS* (AAOS) é um sistema operacional completo instalado diretamente na central multimídia, executando os aplicativos de forma independente.

O presente trabalho foca no ambiente do *Android Auto*, utilizando o Crankshaft executado em um Raspberry Pi. O Crankshaft é um sistema operacional de código aberto (baseado em GNU/Linux) projetado especificamente para transformar hardwares compactos em centrais multimídia, permitindo emular a central multimídia de um veículo na bancada de testes e garantir controle total sobre a execução da interface.

Embora essas plataformas do Google facilitem a integração, elas impõem diretrizes de segurança altamente restritivas [18]. Visando mitigar a distração do motorista, a documentação de qualidade (*Car App Quality Guidelines*) proíbe a criação de interfaces gráficas livres. Os aplicativos devem utilizar a biblioteca *Android for Cars App Library*, que limita o design a componentes estruturais rígidos, como o `ListTemplate` e o `GridTemplate`. Além disso, a regra estrutural SA-1 proíbe a exibição de elementos gráficos animados ou vídeos com o veículo em movimento. Essa restrição torna inviável, por exemplo, a implementação de aplicações que exibam o fluxo de vídeo de câmeras frontais com caixas delimitadoras (*bounding boxes*) em tempo real. As diretrizes também impõem o *Task Step Limit*, que bloqueia fluxos de navegação com mais de cinco interações consecutivas [18] com o objetivo de gerar menos distrações ao motorista.

Essas severas limitações de interface e as restrições do sistema operacional para processamento contínuo em segundo plano motivam a arquitetura deste projeto. O uso de um hardware independente contorna essas barreiras. Essa abordagem permite executar o treinamento do aprendizado federado sem interrupções, garantindo o cálculo preciso da saúde da bateria sem comprometer a estabilidade da central multimídia do veículo ou infringir as regras de segurança impostas pelas

montadoras.

# Capítulo 4

## Metodologia e Evolução da Arquitetura

Este trabalho adotou uma metodologia de prototipagem exploratória e incremental, visando a validação funcional de um cliente de Aprendizado Federado no contexto do projeto AVADiP. O desenvolvimento foi guiado pela busca de soluções que equilibrassem a facilidade de implementação em bancada com a representatividade de um ambiente veicular real.

Originalmente, a proposta arquitetural visava a construção de um sistema de assistência visual avançado. O objetivo era processar o fluxo de vídeo em tempo real e projetar, diretamente na tela do painel do veículo, caixas delimitadoras (*bounding boxes*) sobrepostas aos pedestres ou ciclistas detectados, criando uma experiência similar à realidade aumentada para alertar o condutor sobre a presença de VRUs, conforme ilustrado na Figura 4.1.

Entretanto, durante a análise de viabilidade técnica, identificou-se uma barreira arquitetural crítica na plataforma *Android Auto*. Conforme detalhado na Seção 3.4, as diretrizes de qualidade do Google impõem o uso exclusivo de modelos visuais rígidos (*templates*) e a regra estrutural SA-1 proíbe estritamente a exibição de vídeos ou elementos gráficos dinâmicos com o veículo em movimento [18].

Diante dessas imposições do sistema operacional, o projeto necessitou adaptar-se. A impossibilidade de alteração visual direta redirecionou o foco para métodos alternativos de alerta e, posteriormente, para a validação da arquitetura federada através de dados de sensores não visuais.



Figura 4.1: Exemplo do protótipo inicial descartado: *bounding boxes* para detecção de VRUs. Fonte: Canal do Youtube Grupo de Teleinformática e Automação - GTA [2]

Este capítulo descreve a evolução do protótipo ao longo de três fases distintas, detalhando as decisões de projeto como a escolha de sistemas *open-source* prontos para uso — e os desafios técnicos (restrições de software legado e incompatibilidade de hardware) que emergiram dessas escolhas.

## 4.1 Fase 1: Arquitetura Baseada em Gateway Móvel (Bluetooth)

Diante das restrições impostas para a exibição de alertas visuais na unidade de bordo, a primeira abordagem arquitetural investigada buscou explorar a modalidade auditiva como canal primário de interface homem-máquina (*Human-Machine Interface* – HMI). A premissa era utilizar o sistema de som do veículo para emitir alertas sonoros sem a necessidade de modificação no hardware do carro.

Nesta arquitetura, utilizou-se o **Raspberry Pi Zero 2 W**, executando a distribuição padrão **Raspberry Pi OS** (baseada em Debian). O dispositivo atuava exclusivamente como unidade de processamento de bordo e, por operar sem uma tela acoplada nesta fase, delegava toda a interface com o usuário para um dispositivo móvel intermediário (o *smartphone* do condutor).

### 4.1.1 Arquitetura de Comunicação e Fluxo de Dados do Sistema

A arquitetura foi projetada para operar em três estágios da transmissão do alerta, criando uma ponte entre o Raspberry Pi e o sistema de áudio do veículo ilustrado na Figura 4.2.

1. **Detecção e Sinalização (Raspberry Pi → Smartphone):** A unidade embarcada processa o fluxo de vídeo capturado por uma câmera conectada ao sistema. Ao realizar a inferência visual e detectar uma condição de risco, o Raspberry Pi emite um sinal de controle via *Bluetooth Low Energy* (BLE), operando como um servidor GATT (*Generic Attribute Profile*) que expõe uma característica de notificação para o celular.
2. **Processamento do Alerta (Smartphone):** Um aplicativo móvel, desenvolvido especificamente para este fim, atua como um *gateway*. Ele escuta constantemente as alterações no servidor GATT e, ao receber o sinal de alerta, seleciona o arquivo de áudio correspondente.
3. **Reprodução no Veículo (Smartphone → Carro):** Como o *smartphone* do condutor já se encontra pareado com o sistema de som do veículo para chamadas e mídia, o áudio do alerta gerado pelo aplicativo é automaticamente roteado para os alto-falantes do carro.

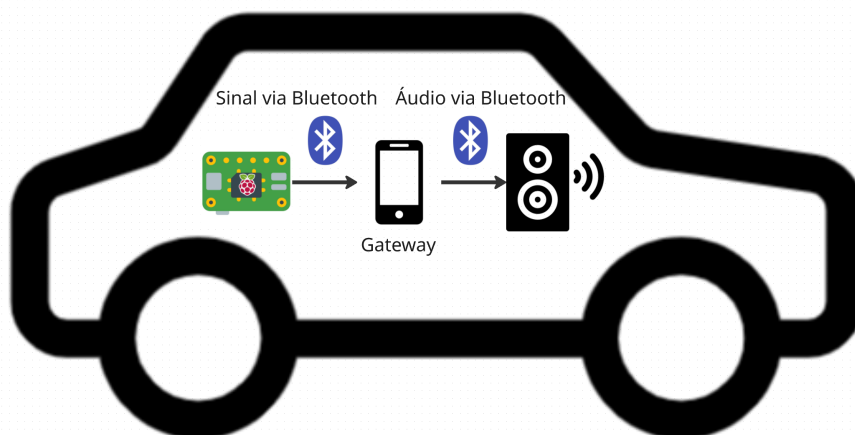


Figura 4.2: Fluxo de dados da arquitetura com smartphone como intermediário.

## 4.1.2 Implementação do Aplicativo Gateway

Para validar esta arquitetura, foi desenvolvido um aplicativo móvel utilizando o *framework* **Flutter** (Linguagem Dart), escolhido pelo acesso nativo facilitado às APIs de Bluetooth em Android. A implementação utilizou a biblioteca `flutter_blue_plus` para o gerenciamento da pilha BLE e a biblioteca `audioplayers` para a reprodução dos alertas. O código incluiu mecanismos de reconexão automática para garantir a persistência do serviço durante a condução.

## 4.1.3 Limitações e Motivação para Evolução

Embora os testes de bancada tenham comprovado a funcionalidade da comunicação, a análise arquitetural revelou que esta abordagem era pouco robusta para um produto final. A dependência mandatória de um dispositivo externo (o celular do motorista) introduziu uma complexidade operacional indesejada, assemelhando-se a uma adaptação provisória (*ad-hoc*) em vez de um sistema embarcado integrado.

Concluiu-se que, para elevar o nível de maturidade do projeto, seria necessário eliminar o celular como intermediário. O objetivo para a próxima fase tornou-se consolidar o processamento e a interface em uma única unidade autônoma, capaz de dialogar diretamente com o veículo.

## 4.2 Fase 2: Arquitetura Embarcada Nativa (Android Auto)

Para eliminar a dependência do celular como intermediário de processamento, o projeto avançou para uma arquitetura em que o Raspberry Pi assume o papel de unidade central veicular completa. O objetivo nesta fase era executar uma instância do *Android Auto*[19] diretamente no Raspberry, permitindo a execução concorrente dos algoritmos de Aprendizado Federado e a gestão de alertas sonoros.

### 4.2.1 Seleção da Plataforma de Software

Embora a compilação do *Android Automotive OS* (AAOS) [19] a partir do código-fonte (AOSP – *Android Open Source Project*) fosse a solução ideal em termos

de pureza técnica, a sua execução prática em hardwares como o Raspberry Pi apresenta uma barreira operacional severa. A construção do sistema exige a configuração de um ambiente de compilação extenso, a resolução de múltiplas dependências no Linux e a aplicação manual de *patches* não oficiais para forçar o funcionamento de periféricos básicos, como adaptadores Wi-Fi e Bluetooth.

Essa complexidade de configuração gera um risco arquitetural elevado para o projeto. Qualquer falha durante o processo de compilação ou na aplicação dessas modificações personalizadas resultaria em um gargalo de desenvolvimento considerável. A resolução desses entraves exigiria um esforço de depuração em nível de administração de sistemas embarcados, o que desviaria os recursos e o tempo do foco real desta pesquisa: a implementação e validação do Aprendizado Federado.

Para mitigar esse risco de desenvolvimento, optou-se pela utilização do **Crankshaft NG**[20], uma distribuição baseada em GNU/Linux que implementa a interface de emulação *OpenAuto*[21] de forma *plug-and-play*. O custo dessa escolha, contudo, é a perda da integração nativa. Enquanto o Android Automotive rodando nativamente permitiria acesso direto à telemetria do carro pelas APIs oficiais do sistema, o Crankshaft atua apenas como um terminal de projeção visual. O problema prático gerado por essa restrição é o isolamento dos processos: os algoritmos de aprendizado de máquina e a coleta de dados da bateria não podem interagir com o painel gráfico do Android Auto. Conseqüentemente, eles ficam restritos a operar como processos independentes do Linux em segundo plano (*background*), exigindo uma solução extra para que os resultados da inteligência artificial possam ser exibidos na tela para o motorista.

## 4.2.2 Desafios de Implementação no Raspberry Pi Zero 2

### W

A primeira iteração utilizou o Raspberry Pi Zero 2 W, mas enfrentou bloqueios críticos:

- **Incompatibilidade de Arquitetura:** A versão mais recente da imagem (*arm7*) causou falhas de inicialização (*boot loop*), exigindo a migração para a versão genérica *armhf* após diagnóstico via console serial.

- **Limitações de Interface (I/O):** A interface gráfica do Crankshaft é projetada para interação via toque (*touch-first*). Embora fosse possível obter saída de vídeo, o funcionamento do recurso de toque (*touch*) do display exigia uma conexão USB-A, indisponível nativamente no Pi Zero 2 W. Tentativas de contorno, conectando um monitor genérico e utilizando mouses via Bluetooth, mostraram-se inviáveis devido à alta latência de resposta e à dificuldade de pareamento via linha de comando sem interface visual, comprometendo a usabilidade do sistema.

### 4.2.3 Migração para Raspberry Pi 4 e Consolidação do Hardware

Para viabilizar a operação autônoma, migrou-se para um **Raspberry Pi 4 Model B**. A disponibilidade de portas Micro HDMI e USB 3.0 permitiu a integração plena de um **Display LCD Touchscreen** de 7 polegadas, consolidando a “central multimídia”.

Além da conectividade, a atualização de hardware trouxe um ganho crítico de memória: o dispositivo conta com 8GB de RAM, em contraste com os 512MB do modelo anterior. Juntamente com o processador Quad-core Cortex-A72, esse recurso ofereceu o desempenho necessário para manter a fluidez da interface do Android Auto concorrentemente com o treinamento do modelo de IA.

### 4.2.4 Configurações Críticas de Sistema e Rede

Para adequar a distribuição Crankshaft, originalmente projetada apenas para espelhamento de tela, às necessidades de um cliente de Aprendizado Federado, foram necessárias modificações estruturais nas configurações do sistema operacional:

1. **Vídeo e Interface (HDMI):** Para garantir a compatibilidade com o display LCD de 7 polegadas, foi necessário editar manualmente os parâmetros de inicialização (`/boot/config.txt`). Como o sistema não inicializava a interface gráfica corretamente antes desses ajustes, este processo tornou-se oneroso, exigindo um ciclo iterativo de remoção física do cartão SD para edição em um

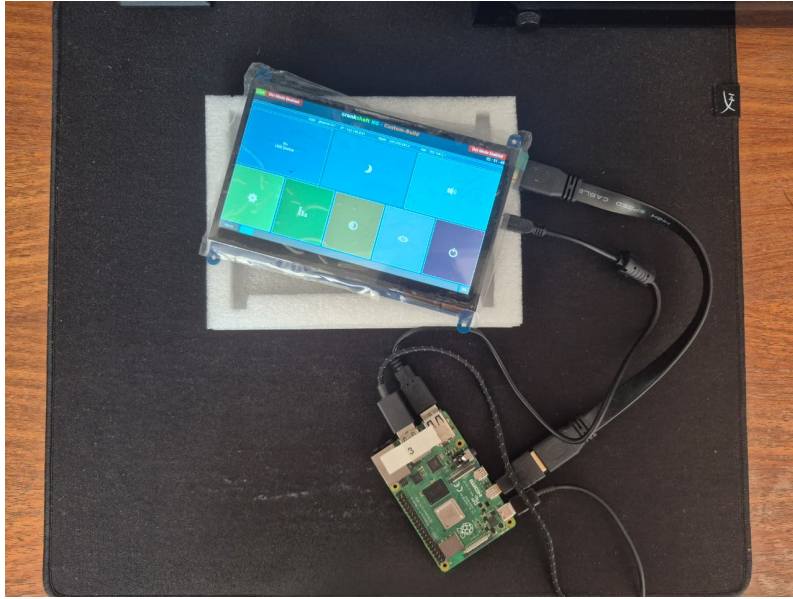


Figura 4.3: Montagem física do dispositivo de borda com display touchscreen

computador externo e reinsertão para teste a cada nova tentativa de configuração.

2. **Acesso ao Modo Desenvolvedor (via SSH):** O sistema vem bloqueado por padrão para o usuário final. Foi necessário ativar o “Modo Desenvolvedor” (*Dev Mode*) para habilitar o servidor SSH. Isso foi fundamental para permitir o acesso remoto ao terminal, viabilizando a transferência de arquivos (*datasets* e *scripts* Python) via protocolo SCP (*Secure Copy Protocol*) e a edição direta de códigos utilizando o editor `nano` ou colagem de texto via terminal.
3. **Inversão da Topologia de Rede (Hotspot vs. Cliente):** Por padrão, o Crankshaft opera em modo *Hotspot*, criando sua própria rede Wi-Fi para que o celular se conecte a ele. No entanto, essa configuração impossibilita o Raspberry Pi de se conectar à Internet de maneira wireless, impedindo a comunicação do cliente com o servidor externo.

Para solucionar isso, alterou-se a configuração de rede para o modo **Cliente Wi-Fi**. O recurso de Hotspot interno foi desativado e o Raspberry Pi foi configurado para se conectar a uma rede externa, fornecida por um roteador Wi-Fi convencional ou pelo compartilhamento de conexão do próprio celular. Essa inversão de topologia garantiu que o dispositivo tivesse acesso simultâneo à Internet, essencial para o Aprendizado Federado, e à comunicação local com

o Android Auto.

## 4.3 Fase 3: O Desafio da Aceleração e a Consolidação do Ambiente

No contexto do projeto AVADiP, o objetivo inicial previa a detecção de Usuários Vulneráveis utilizando aceleradores de IA dedicados, especificamente o módulo **Hailo-8**. A premissa era desonerar a CPU do Raspberry Pi, delegando a inferência custosa de visão computacional para a NPU (*Neural Processing Unit*).

### 4.3.1 Incompatibilidade Arquitetural e Falha na Virtualização

A integração do Hailo-8 exige a instalação de drivers de kernel nativos e bibliotecas de espaço de usuário específicas. Entretanto, o uso da distribuição *Crankshaft NG* criou um conflito de gestão sobre o hardware.

Como o Crankshaft é uma distribuição projetada para assumir o controle total do dispositivo (vídeo, áudio e entrada), a instalação concorrente dos *drivers* do acelerador Hailo revelou-se inviável devido a incompatibilidades de bibliotecas base e conflitos no *kernel*. Buscou-se, então, uma abordagem de virtualização utilizando a ferramenta **QEMU** via linha de comando para tentar isolar o ambiente do Crankshaft dentro do sistema hospedeiro.

Essa abordagem falhou devido à estrita dependência de *hardware* do *Android Auto*. O sistema de projeção exige acesso direto e exclusivo (*passthrough*) à GPU VideoCore e às interfaces USB para garantir a fluidez da imagem. As camadas de abstração impostas pela emulação do QEMU impediram esse acesso direto com o desempenho necessário, resultando em falhas críticas na inicialização da interface gráfica.

### 4.3.2 O Pivô para Predição de SOH (Battery Health)

Diante do impasse técnico entre manter a integração veicular (Crankshaft) ou a aceleração de hardware (Hailo), o projeto priorizou a arquitetura veicular. Sem a NPU, a execução de modelos complexos de detecção de objetos tornou-se inviável

na CPU, comprometendo a latência do sistema.

Decidiu-se alterar o estudo de caso para outra vertente do projeto AVADiP: a **Predição do Estado de Saúde de Baterias (SOH)**. Este novo escopo valida a mesma arquitetura federada proposta, mas utiliza modelos de regressão baseados em séries temporais. Esses modelos são computacionalmente mais leves, permitindo sua execução na CPU do Raspberry Pi 4 simultaneamente ao Android Auto, sem a necessidade de aceleradores externos.

### 4.3.3 Desafios de Software em Ambiente Legado

Com o hardware e o estudo de caso definidos, o projeto enfrentou seu maior desafio de engenharia de software: a obsolescência das bibliotecas do sistema operacional embarcado.

O Crankshaft é baseado no *Debian Buster*, uma versão estável antiga do Linux, que utiliza a biblioteca padrão C (`glibc`) na versão 2.28. Entretanto, as bibliotecas modernas de Aprendizado de Máquina exigidas pelo projeto são compiladas com versões recentes da `glibc` (2.29 ou superior).

#### 4.3.3.1 Falha na Compilação Local

A abordagem inicial e mais direta consistiu em tentar compilar as dependências a partir do código-fonte nativamente no próprio Raspberry Pi. Contudo, a compilação local de bibliotecas científicas (como *NumPy*, *SciPy* e *gRPC*) resultou na exaustão de recursos do dispositivo. O uso intenso da CPU por longos períodos causou superaquecimento (*thermal throttling*), gerando travamentos do sistema antes da conclusão do processo (*build*). Uma alternativa técnica para contornar essa falha seria realizar a compilação cruzada (*cross-compilation*) em um computador hospedeiro com mais desempenho e, posteriormente, transferir os binários para a Raspberry. No entanto, essa técnica introduz uma complexidade considerável na configuração de *toolchains* compatíveis com a arquitetura ARM, exigindo um esforço bem maior de configuração.

#### 4.3.3.2 Definição da Matriz de Compatibilidade de Software

A solução definitiva foi abandonar a compilação local e utilizar binários pré-compilados (*wheels*) fornecidos pelo repositório *piwheels*[22], específicos para a arquitetura ARMv7. Para isso, foi necessário realizar um mapeamento rigoroso de versões, congelando o ambiente de desenvolvimento em uma janela temporal compatível com o sistema legado (aproximadamente 2020-2021).

A tabela abaixo ilustra a defasagem entre as versões utilizadas para garantir a estabilidade do sistema e as versões atuais de mercado (referência 2025), evidenciando o esforço de compatibilidade realizado:

- **Flower (Cliente FL):** Utilizou-se a versão **0.19.0**. Esta versão é significativamente anterior à atual série 1.13+, sendo a última a manter suporte estável para implementações legadas de gRPC compatíveis com o Python 3.7.
- **NumPy:** Fixado na versão **1.19.5**. Versões superiores (como a atual 2.0.0+) exigem instruções de processador e bibliotecas de sistema não suportadas pelo ambiente Debian Buster.
- **SciPy:** Fixado na versão **1.5.4**. Essencial para os cálculos matemáticos do modelo de bateria, esta versão foi a última pré-compilada capaz de operar em conjunto com o NumPy 1.19 sem quebrar dependências binárias.

A combinação dessas bibliotecas permitiu que o cliente de Aprendizado Federado operasse de forma estável dentro das restrições de um sistema operacional automotivo mais legado.

#### 4.3.4 Desacoplamento da Interface de Usuário (HMI Web)

Conforme relatado nas limitações do ecossistema Android Auto, as diretrizes de segurança da Google impõem o uso exclusivo de modelos visuais pré-aprovados (*templates*) para a construção de interfaces nativas. Essa restrição sistêmica impede o desenho livre de telas para garantir que todos os aplicativos sigam um padrão rigoroso de mitigação de distrações ao condutor. Ao optar por hospedar a aplicação visual em um servidor *web* acessado via navegador, o projeto tecnicamente contorna a obrigatoriedade de uso do motor de *templates* oficial. Contudo, é fundamental

ressaltar que essa abordagem não viola os princípios de segurança automotiva. A interface *web* desacoplada foi projetada especificamente para respeitar os pilares do *Design for Driving* da Google, em especial a diretriz de **Legibilidade em Relance** (*Glanceability*). Essa regra estipula que a interface não deve exigir foco sustentado, permitindo que o condutor absorva a informação em desvios de olhar extremamente breves (tipicamente inferiores a dois segundos). O painel desenvolvido atende rigorosamente a esse critério: apresenta tipografia em larga escala, alto contraste e total ausência de elementos interativos complexos ou barras de rolagem. Ao exibir estritamente dados passivos (o percentual numérico de SOH e o status do treinamento), garante-se que a compreensão da informação seja imediata. Dessa forma, a prova de conceito valida o modelo distribuído de maneira flexível, mantendo-se perfeitamente alinhada aos critérios de segurança viária estabelecidos pela indústria.

A solução definitiva foi baseada na implementação de um servidor web local assíncrono, hospedado diretamente no dispositivo Raspberry Pi. A natureza assíncrona, viabilizada através da criação de múltiplas linhas de execução concorrentes (*threads*), foi uma exigência técnica fundamental ilustrado na Figura 4.4. Como o treinamento federado via framework Flower é um processo computacionalmente intenso e bloqueante, a execução comum faria com que o painel web ficasse inacessível e travado durante as rodadas de aprendizado. Com a separação arquitetural, o servidor web Flask processa as requisições da página simultaneamente ao treinamento contínuo do modelo.

Dessa maneira, a visualização da Saúde da Bateria passou a ser agnóstica em relação à tela principal do veículo. Qualquer aparelho conectado à mesma rede do dispositivo de borda pode exibir o painel de telemetria (*Dashboard*) acessando a rede local. Durante os experimentos de bancada, a validação ocorreu com o Raspberry Pi e o computador de desenvolvimento conectados à mesma rede sem fio residencial.

Para a operação em um cenário veicular real, a topologia de rede projetada consiste em utilizar o *smartphone* do condutor para gerar um ponto de acesso de rede local (função de Hotspot). O Raspberry Pi realiza a conexão a essa rede para prover o Android Auto e obter acesso à Internet, enquanto o próprio celular, por ser o hospedeiro da rede local, pode acessar a página do servidor web (Figura 4.5) através

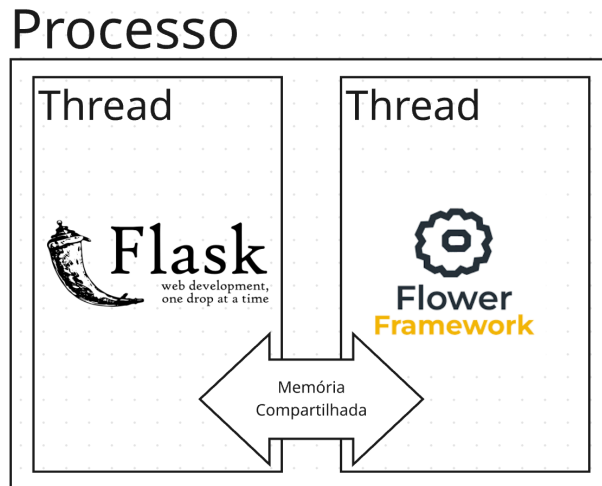


Figura 4.4: Flask + Flower rodando simultaneamente

do navegador, atuando como um visor secundário posicionado em um suporte no painel do carro.

Para a operação em um cenário veicular real, o único requisito externo da arquitetura é a conectividade com a Internet, necessária exclusivamente para a transmissão dos parâmetros durante as rodadas de comunicação com o servidor federado. Em automóveis equipados com conectividade nativa, como modems celulares 4G ou 5G de fábrica, o Raspberry Pi conecta-se diretamente à rede do veículo, operando de maneira integralmente independente.

Caso o veículo não possua conexão própria, a arquitetura permite a utilização de um *smartphone* atuando apenas como ponto de acesso de rede local (função de *hotspot*). Diferente do modelo de projeção tradicional, o celular não realiza nenhum processamento do aprendizado de máquina ou da aplicação veicular; sua função restringe-se ao roteamento de pacotes. Como benefício adicional dessa topologia, qualquer dispositivo conectado a essa rede, incluindo o próprio aparelho roteador, pode acessar a página do servidor web (Figura 4.5) através de um navegador padrão, atuando como um monitor auxiliar no painel do carro.

A Figura 4.5 ilustra a interface final da aplicação veicular apresentada ao condutor. Cabe esclarecer a natureza dessa tela: ela não é uma simples ferramenta de depuração do treinamento, mas sim o painel de diagnóstico do usuário final que reflete os resultados do Aprendizado Federado em tempo real. O valor percentual de SOH exibido (como 92,5%) não é um dado bruto do veículo, mas a inferência



Figura 4.5: Painel de telemetria servido localmente via rede sem fio

mais recente gerada pelo modelo de regressão. Durante o ciclo de comunicação do *framework* Flower, especificamente na etapa de avaliação local (*evaluate*), o modelo utiliza os pesos atualizados para prever a saúde da bateria com base na base de dados de teste local. Esse resultado numérico, juntamente com o status atual do processo ("Treinamento Finalizado"), é colocado no *buffer* de memória compartilhada.

# Capítulo 5

## Desenvolvimento e Implementação

Este capítulo descreve os procedimentos técnicos e as decisões de engenharia de software adotadas para a construção do protótipo funcional. A implementação foi dividida entre a preparação da infraestrutura nos dispositivos de borda, a implementação dos modelos preditivos seguindo os preceitos do Aprendizado Federado e a criação da interface de telemetria em formato web. O objetivo é detalhar como a arquitetura proposta na metodologia foi traduzida em código executável operando sob as restrições de hardware e compatibilidade.

A Figura 5.1 ilustra a arquitetura geral do sistema, detalhando a interação entre os componentes físicos e os módulos de software do protótipo.

Todo o código-fonte desenvolvido para este trabalho, englobando os *scripts* do servidor federado, dos clientes embarcados e as rotinas de preparação dos dados, encontra-se disponível publicamente no GitHub. O repositório pode ser acessado através do link: <https://github.com/vrtaranto-ufrj/Projeto-FInal>.

### 5.1 Configuração do Ambiente e Otimização para Hardware Heterogêneo

A validação do treinamento federado em bancada exigiu a configuração de múltiplos nós clientes para simular um ambiente veicular distribuído. Idealmente, em um cenário real, todos os veículos teriam hardware padronizado. Contudo, devido às limitações físicas dos equipamentos disponíveis para o teste, o ecossistema foi montado de forma heterogênea. Utilizou-se um Raspberry Pi 4 Model B, que serviu

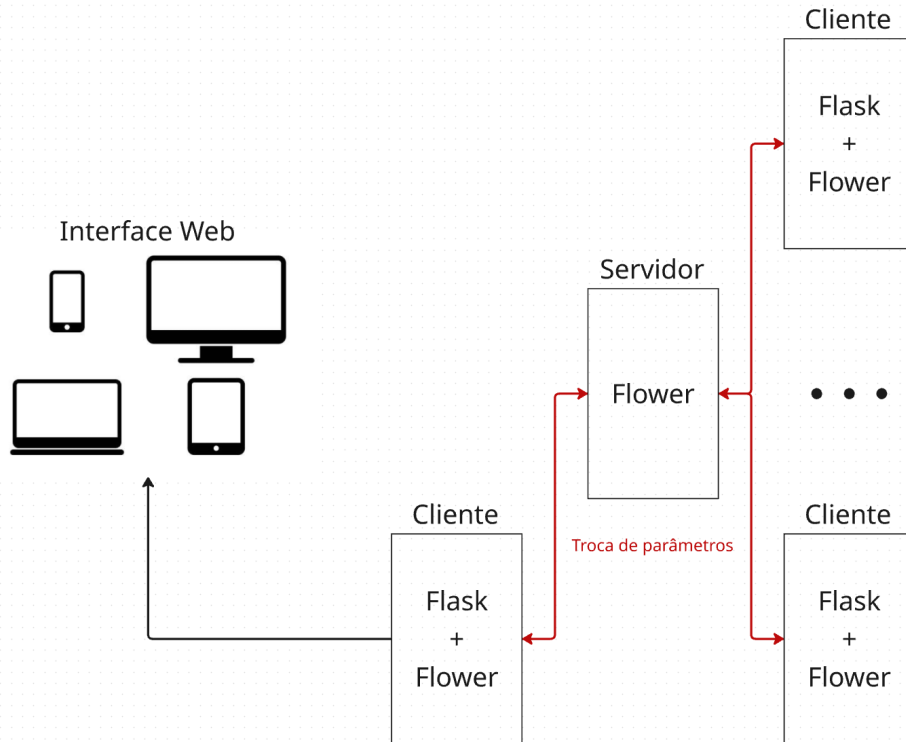


Figura 5.1: Arquitetura de software e hardware do sistema embarcado

como o alvo principal de testes e o hospedeiro da interface veicular, e um Raspberry Pi Zero 2 W. Ambos operaram exclusivamente como nós clientes equivalentes na topologia da rede Flower. O Raspberry Pi Zero 2 W atuou apenas como um dispositivo secundário para viabilizar a agregação federada e, devido à sua extrema limitação de memória, foi configurado para operar estritamente em modo textual (Interface de Linha de Comando), dispensando o servidor gráfico.

O gerenciamento de dependências e a configuração do ambiente Python foram estruturados para garantir compatibilidade total com o sistema operacional base (Debian Buster). Como sistemas embarcados automotivos priorizam a estabilidade e mantêm versões conservadoras de bibliotecas fundamentais, como a `glibc`[23], a adoção de ferramentas modernas de desenvolvimento frequentemente esbarra em limitações de arquitetura. Durante a prototipagem, testou-se inicialmente o uso do `uv`[24], um gerenciador de pacotes e ambientes virtuais avançado. Embora o `uv` ofereça vantagens expressivas de desempenho, como a instalação muito rápida de bibliotecas e a sua facilidade no gerenciamento de múltiplas versões do Python em comparação ao instalador padrão (`pip`), sua execução exige dependências de sistema

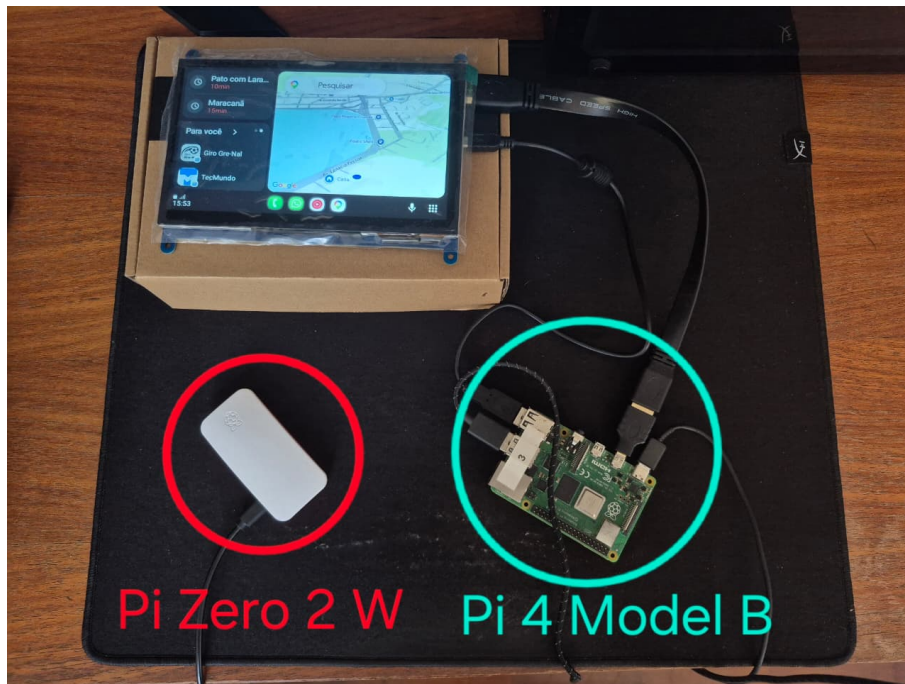


Figura 5.2: Disposição física dos dispositivos de borda e infraestrutura de rede local.

operacional mais recentes, resultando em incompatibilidades de versionamento. Para contornar esse obstáculo e garantir a robustez da aplicação, a infraestrutura foi consolidada utilizando as ferramentas nativas de gerenciamento do Python. Essa decisão técnica assegurou a execução estável de todas as bibliotecas necessárias para o treinamento federado, mantendo a integridade e a segurança do sistema embarcado.

Para garantir o isolamento do ambiente de desenvolvimento dentro das restrições do sistema, adotou-se o módulo nativo `venv` do Python. A estruturação das bibliotecas requereu um mapeamento rigoroso de compatibilidade. Para isso, realizou-se uma etapa de tentativa e erro de instalações de versões diferentes dessas bibliotecas, para encontrar uma combinação em que todas elas conseguissem funcionar simultaneamente.

Durante a configuração do ambiente, observou-se que a instalação padrão via código-fonte apresentava um custo de tempo desproporcional. Testes práticos com a biblioteca `NumPy`, por exemplo, ultrapassaram 20 minutos de processamento local — uma etapa que tipicamente leva menos de um minuto utilizando pacotes prontos. Conforme discutido anteriormente em relação às restrições de *hardware* da placa, essa abordagem tornava o fluxo de desenvolvimento muito lento. Para resolver esse

gargalo, integrou-se o repositório *piwheels* [22], que fornece binários pré-compilados (*wheels*) otimizados para a arquitetura ARM, viabilizando a instalação eficiente da maior parte do ecossistema científico.

No entanto, a exigência por versões específicas e antigas, resultou na ausência de um pacote pronto para a biblioteca *SciPy* precisando, então, fazer a compilação. O processo demandou o uso máximo dos múltiplos núcleos do Raspberry Pi 4 por aproximadamente 15 minutos ininterruptos. Foi exatamente essa sobrecarga contínua que desencadeou os picos de temperatura e o *thermal throttling* relatados nas seções anteriores, evidenciando na prática o limite físico da prototipagem direta na borda.

Para evitar falhas de hardware em tempo de execução e garantir a conclusão do processo, foi necessário aplicar uma solução de resfriamento ativo diretamente na bancada. A intervenção consistiu em manter o contato indireto de um saco plástico impermeável contendo gelo sobre o dissipador de calor do dispositivo. Essa técnica improvisada demandou monitoramento ininterrupto e secagem manual constante da superfície do invólucro, uma vez que o choque térmico provocava a rápida condensação da umidade ambiente na parte externa do plástico, gerando risco crítico de gotejamento e curto-circuito na placa de circuito impresso. Essa ação física atípica foi o fator determinante que permitiu a conclusão da etapa de compilação e a consolidação do ambiente virtual para o cliente federado.

## 5.2 Implementação dos Clientes de Aprendizado Federado

A essência da arquitetura distribuída proposta reside na implementação dos clientes locais utilizando o *framework* Flower. Cada nó da rede, representado pelos dispositivos embarcados, executa uma instância autônoma capaz de treinar um modelo de aprendizado de máquina com seus dados locais e comunicar apenas os parâmetros do modelo atualizados para o servidor central, preservando a privacidade da telemetria bruta do veículo. O código-fonte integral desta implementação encontra-se disponível no Apêndice A.

Para a predição do Estado de Saúde (SOH) das baterias, optou-se pela uti-

lização de um modelo de regressão linear treinado via Gradiente Descendente Estocástico (SGDRegressor). A escolha desse algoritmo justifica-se pela sua extrema eficiência computacional e baixo consumo de memória, características obrigatórias para a execução concorrente no ambiente restrito dos dispositivos Raspberry Pi.



Figura 5.3: Fluxo de processamento do dado: da telemetria local à extração de parâmetros federados.

A escolha do algoritmo `SGDRegressor` da biblioteca Scikit-Learn [25] foi motivada fundamentalmente pela sua eficiência computacional. Diferente de modelos de aprendizado profundo, que exigiriam a execução de *frameworks* pesados como PyTorch ou TensorFlow, o `SGDRegressor` oferece uma abordagem linear leve, plenamente adequada às severas restrições de processamento e memória do Raspberry Pi.

Cabe ressaltar que o escopo desta etapa de implementação não consistiu em desenvolver o modelo preditivo com o maior grau de acurácia possível, tampouco realizar um ajuste fino de hiperparâmetros (*fine-tuning*). O propósito primário foi validar a viabilidade da arquitetura, construindo um protótipo funcional capaz de executar o ciclo completo de treinamento federado no hardware veicular. A adoção de um modelo linear mais simples cumpre perfeitamente o papel de prova de conceito, permitindo focar os esforços de engenharia na comunicação de rede e na execução simultânea do aprendizado federado com a interface do Android Auto.

### 5.2.1 Aquisição e Particionamento dos Dados de Telemetria

Em um cenário veicular ideal, as métricas de saúde da bateria (SOH) seriam extraídas em tempo real diretamente do barramento CAN (*Controller Area Network*). No entanto, na prática automotiva atual, a extração de dados internos do *Battery Management System* (BMS) esbarra em restrições de propriedade intelectual. As montadoras frequentemente ofuscam os identificadores de rede ou

bloqueiam a leitura de métricas avançadas da bateria por sistemas de terceiros, exigindo processos complexos de engenharia reversa para a decodificação dos pacotes.

Diante dessa barreira técnica de interoperabilidade, e considerando que o escopo central deste trabalho é a validação da arquitetura de processamento distribuído (Aprendizado Federado), e não a engenharia reversa de protocolos automotivos proprietários, o protótipo utilizou a ingestão de bases de dados de telemetria padronizadas. Essa abordagem garante a reprodutibilidade do experimento e isola as variáveis de rede, permitindo validar o modelo sem as limitações de *hardware* impostas por fabricantes específicos.

A base de dados original utilizada no projeto foi obtida a partir de um repositório público de pesquisa focado em dados de carregamento de veículos elétricos em operação real [26]. O conjunto original disponibilizado pelos pesquisadores é composto por uma vasta quantidade de arquivos compactados, contendo o histórico de degradação de múltiplas baterias ao longo do tempo.

Para comprovar a viabilidade da execução federada na bancada e aproximar o experimento de um cenário real, foram selecionados dois arquivos distintos desse montante, representando o histórico de telemetria de dois veículos independentes. Essa escolha metodológica é crucial, pois introduz uma característica inerente às redes distribuídas: a heterogeneidade dos dados locais. Como cada veículo está sujeito a padrões de condução, ciclos de recarga e condições térmicas únicos, suas curvas de degradação são naturalmente diferentes. O primeiro arquivo foi alocado integralmente no armazenamento do Raspberry Pi 4, enquanto o segundo foi destinado ao Raspberry Pi Zero 2 W. Essa separação estrutural cumpriu o objetivo de espelhar uma frota autêntica, garantindo que cada nó embarcado possuísse um contexto local exclusivo para realizar o treinamento de seu modelo, preservando a privacidade da base bruta.

Para tornar o código-fonte escalável e evitar a fixação manual de nomes de arquivos dentro do script, a rotina de carregamento dos dados foi construída de forma dinâmica. Utilizando a biblioteca nativa do Python `argparse`, o programa exige que o usuário forneça um parâmetro de identificação no momento da execução via terminal.

Quando o comando de inicialização é executado, o identificador numérico do

cliente é capturado e utilizado para construir dinamicamente o caminho do arquivo de dados local. A título de exemplo, caso um dispositivo embarcado seja iniciado recebendo o identificador '1' via linha de comando, o sistema mapeia e carrega automaticamente o arquivo de telemetria correspondente àquele cliente específico. Essa parametrização simplificou a orquestração dos testes, permitindo implantar exatamente o mesmo código-fonte em múltiplos hardwares, bastando alterar o argumento de execução na inicialização do script.

A construção do cliente, instanciada através da classe `BatteryClient`, exige a definição precisa dos hiperparâmetros do regressor. A Listagem 5.1 ilustra a inicialização do modelo, na qual a taxa de aprendizado e a penalidade foram ajustadas para garantir a convergência gradual das perdas durante o treinamento federado.

Código 5.1: Inicialização do modelo regressor local.

```
1 class BatteryClient(fl.client.NumPyClient):
2     def __init__(self, X_train, y_train, X_test, y_test):
3         self.model = SGDRegressor(
4             penalty="l2",
5             max_iter=1,
6             warm_start=True,
7             learning_rate="constant",
8             eta0=0.001,
9             random_state=42
10        )
11        self.X_train, self.y_train = X_train, y_train
12        self.X_test, self.y_test = X_test, y_test
13        self.model.partial_fit(np.zeros((1, 3)), [0])
```

Durante cada rodada de comunicação, o servidor central envia os pesos globais mais recentes para os dispositivos de borda. O cliente atualiza seu regressor local, realiza o treinamento sobre a base de dados de telemetria particionada e procede à etapa de avaliação.

A etapa de avaliação (`evaluate`) assume um papel duplo nesta arquitetura. Além de calcular o Erro Quadrático Médio (MSE) para o controle da rede federada, ela intercepta a predição local para atualizar a interface veicular. Conforme demons-

trado na Listagem 5.2, o modelo classifica o nível de degradação da bateria e injeta os resultados formatados juntamente com códigos de cores dinâmicos diretamente na estrutura de dados consumida pelo painel web.

Código 5.2: Função de avaliação e injeção de dados na interface.

```
1  def evaluate(self, parameters, config):
2      self.set_parameters(parameters)
3      y_pred = self.model.predict(self.X_test)
4      mse = mean_squared_error(self.y_test, y_pred)
5
6      estimated_soh = np.mean(y_pred)
7      DASHBOARD_DATA["soh"] = f"{estimated_soh:.1f}"
8
9      if estimated_soh >= 90:
10         DASHBOARD_DATA["status"] = "BATERIA SAUDÁVEL"
11         DASHBOARD_DATA["color"] = "#28a745"
12     elif estimated_soh >= 75:
13         DASHBOARD_DATA["status"] = "DEGRADAÇÃO LEVE"
14         DASHBOARD_DATA["color"] = "#fd7e14"
15     else:
16         DASHBOARD_DATA["status"] = "CRÍTICO: MANUTENÇÃO"
17         DASHBOARD_DATA["color"] = "#dc3545"
18
19     return float(mse), len(self.X_test), {"mse":
        float(mse)}
```

Por fim, o desacoplamento entre o treinamento do modelo e a interface com o usuário foi solucionado executando as tarefas de forma concorrente. A orquestração da aplicação divide o sistema em duas frentes de trabalho simultâneas: a via principal do programa, responsável pela comunicação de rede e execução do cliente Flower, e uma *thread* secundária dedicada exclusivamente a manter o servidor web operacional.

O microserviço estruturado via Flask é instanciado nesta segunda *thread* utilizando o modo *daemon*. Para que o painel exiba os dados corretamente, é estruturada uma clara divisão entre as etapas de treinamento e teste no nó cliente.

A base de dados local é previamente particionada, destinando 80% das amostras para o ajuste do modelo e 20% para avaliação. Na *thread* principal, o *framework* *Flower* executa o treinamento na parcela maior e, ao final de cada rodada, realiza a inferência do SOH utilizando o conjunto de testes. É exclusivamente o resultado dessa inferência que é injetado na memória compartilhada. Consequentemente, a disponibilidade do servidor *web* não é afetada pelos ciclos de treinamento, ele continua rodando de forma contínua e assíncrona em segundo plano, apenas consumindo o último valor inferido para que o condutor consulte o SOH atualizado a qualquer momento. A característica *daemon* atua estritamente como um mecanismo de segurança de ciclo de vida: caso a aplicação embarcada seja finalizada pelo sistema operacional ou interrompida manualmente, a *thread* secundária é abortada automaticamente. Isso evita a ocorrência de processos órfãos que manteriam portas de rede travadas indefinidamente no Raspberry Pi.

A arquitetura do código foi propositalmente desenhada para ser simples e autocontida. Toda a estrutura visual da interface web (o código HTML e a estilização CSS) foi embutida como uma única variável de texto puro dentro do próprio script Python. Essa abordagem eliminou a necessidade de criar diretórios externos para gerenciamento de *templates* ou arquivos estáticos, consolidando todo o cliente embarcado em um único arquivo executável de fácil distribuição.

Para viabilizar a comunicação interna entre a rotina de aprendizado e a página web, adotou-se uma estrutura de dados global compartilhada (um dicionário Python) que atua como um *buffer* em memória. A mecânica de atualização reflete a natureza iterativa do *framework*: o painel não exhibe apenas o *status* do treinamento, mas atua como um monitor de inferência em tempo real. A cada nova rodada de comunicação, especificamente durante a etapa de avaliação local (*evaluate*), o modelo recém-atualizado pelo servidor realiza a predição do SOH sobre os dados de teste do veículo. A *thread* do Flower então registra esse valor inferido e a respectiva cor de alerta no dicionário. Simultaneamente, a *thread* do Flask consome essa mesma estrutura para renderizar o diagnóstico dinâmico na tela.

Para criar a percepção visual de atualização em tempo real, avaliou-se inicialmente o uso de *WebSockets*, um protocolo de comunicação bidirecional projetado para manter uma conexão contínua e persistente entre o cliente e o servidor. Embora

seja o padrão da indústria para painéis dinâmicos, a adoção dessa tecnologia adicionaria uma complexidade arquitetural e um custo de gerenciamento de conexões desnecessários para o escopo deste projeto. Como a interface gráfica desenvolvida é extremamente leve, consistindo apenas na renderização de dados textuais numéricos, optou-se por uma abordagem de *polling* passivo. A solução integrou uma instrução nativa de metadados no cabeçalho do HTML (*meta refresh*) que instrui o navegador a recarregar a página inteira de forma automática a cada três segundos. Essa escolha simplificou drasticamente a implementação e garantiu que o *smartphone* ou a central multimídia buscasse constantemente os valores mais recentes no *buffer* compartilhado, sem onerar o servidor e proporcionando ao condutor a visualização atualizada dos diagnósticos sem qualquer interação manual.

A Listagem 5.3 apresenta o bloco principal de inicialização, demonstrando a simplicidade da orquestração em que a *thread* secundária do Flask é iniciada instantes antes da conexão bloqueante do cliente federado.

Código 5.3: Orquestração concorrente das tarefas locais.

```
1 def main():
2     X_train, X_test, y_train, y_test = load_data()
3
4     # Inicializa o painel web de forma desacoplada via
5     # daemon thread
6     site_thread = threading.Thread(target=run_flask,
7     daemon=True)
8     site_thread.start()
9
10    # Inicia a conexão bloqueante com o servidor federado na
11    # thread principal
12    try:
13        fl.client.start_numpy_client(
14            server_address=args.server,
15            client=BatteryClient(X_train, y_train, X_test,
16                                y_test)
17        )
18    except Exception as e:
```

## 5.3 Desenvolvimento da Interface de Telemetria

Na implementação prática da interface web, um detalhe técnico relevante para a controle do sistema foi a definição dinâmica das portas de rede. Para garantir flexibilidade durante o desenvolvimento, a porta de acesso ao servidor `Flask` é calculada em tempo de execução, somando o valor base 5000 ao número de identificação do cliente fornecido via linha de comando.

Embora na configuração de *hardware* final cada Raspberry Pi hospede apenas um cliente exclusivo, essa estratégia de alocação de portas foi fundamental nas fases iniciais de validação. O mapeamento dinâmico permitiu a execução simultânea de múltiplos clientes simulados em uma única máquina física, possibilitando testes de estresse e sincronização na rede federada sem gerar conflitos de alocação de portas no sistema operacional.

A interface com o usuário foi projetada com foco absoluto na usabilidade veicular e na segurança viária. Como o público-alvo final é um motorista em condução, o painel exige interação cognitiva nula, atuando estritamente como um mostrador passivo. A tela apresenta o nível de Saúde da Bateria (SOH) através de numeração em grande escala e utiliza um sistema de semáforo visual para indicar o estado de degradação. A lógica de categorização foi programada diretamente no script cliente: baterias com saúde igual ou superior a 90 por cento recebem a cor verde; valores entre 75 e 89 por cento acionam a cor laranja; e predições inferiores a 75 por cento alteram a interface para a cor vermelha.

Para manter o sincronismo entre os cálculos do modelo e a interface visual sem demandar qualquer ação manual, a página HTML foi configurada com uma diretiva nativa de metadados. Essa instrução obriga o navegador do cliente a recarregar o conteúdo automaticamente a cada três segundos, garantindo que o condutor visualize a progressão do treinamento passivamente.

Do ponto de vista da infraestrutura de software, a estrutura completa da página e as regras de estilização CSS foram embutidas no próprio script Python em formato de texto puro. Além disso, a logomarca do projeto AVADiP não é ar-

mazenada localmente, sendo requisitada dinamicamente através de um link externo hospedado no site institucional ([gta.ufrj.br](http://gta.ufrj.br)). Essa decisão de projeto dispensou totalmente a necessidade de instalar e configurar servidores web robustos dedicados a servir arquivos estáticos, como Apache ou Nginx. O resultado foi a consolidação do cliente embarcado como uma aplicação monolítica extremamente leve, otimizada para operar sem sobrecarregar a memória do hardware.

## 5.4 Orquestração da Comunicação e Estratégia de Agregação

O gerenciamento da arquitetura federada é conduzido por um servidor central operando sob o *framework* Flower. No escopo da implementação, este elemento coordenador foi configurado para utilizar a estratégia de agregação *Federated Averaging* (FedAvg), executada nativamente pela biblioteca através do módulo estratégico padrão.

A codificação do servidor foi estruturada para garantir a estabilidade da regressão linear desde o primeiro instante da conexão. Como os clientes embarcado iniciam suas execuções com pesos aleatórios locais, o servidor foi programado para inicializar e injetar parâmetros globais padronizados na rede antes da primeira rodada de treinamento. A Listagem 5.4 apresenta a rotina responsável por instanciar um modelo de regressão idêntico ao dos clientes, extrair seus coeficientes iniciais e transmiti-los, garantindo que todos os nós partam exatamente do mesmo estado inicial.

A implementação do servidor foi estruturada para garantir a estabilidade da regressão linear desde o primeiro instante da conexão. Como os clientes iniciam com pesos aleatórios, o servidor foi programado para injetar parâmetros iniciais padronizados na rede. A Listagem 5.4 apresenta a função responsável por criar um modelo de regressão idêntico ao dos clientes, garantindo que todos partam do mesmo estado matemático.

É importante destacar que a camada de orquestração do Flower opera de forma totalmente agnóstica ao modelo preditivo utilizado. Para o servidor central, os dados trafegados resumem-se a matrizes numéricas representando pesos e ter-

mos de viés. O orquestrador não possui conhecimento sobre a natureza física dos dados ou sobre a finalidade específica da regressão linear, limitando-se a executar operações aritméticas de média ponderada. Esse isolamento funcional reforça a robustez da arquitetura, garantindo que o servidor atue apenas como um mediador matemático sem acesso à semântica da telemetria veicular. O código completo do servidor encontra-se no Apêndice B.

Código 5.4: Definição de parâmetros iniciais no servidor.

```
1 def get_initial_parameters():
2     model = SGDRegressor(
3         penalty="l2",
4         max_iter=1,
5         warm_start=True,
6         learning_rate="constant",
7         eta0=0.001,
8         random_state=42,
9     )
10    model.partial_fit(np.zeros((1, 3)), [0])
11    return fl.common.weights_to_parameters([model.coef_,
        model.intercept_])
```

A configuração da estratégia do servidor flower é definida no bloco principal do script. Conforme ilustrado na Listagem 5.5, os parâmetros `min_fit_clients` e `min_available_clients` foram configurados com o valor 2. Essa restrição é vital para a integridade do experimento: o servidor é instruído a aguardar obrigatoriamente a conexão dos dois dispositivos Raspberry Pi antes de iniciar qualquer rodada de treinamento. Sem esse quórum mínimo, a média federada não teria amostras suficientes para representar a diversidade da rede.

Código 5.5: Configuração da estratégia FedAvg e inicialização do servidor.

```
1 if __name__ == "__main__":
2     strategy = fl.server.strategy.FedAvg(
3         min_fit_clients=2,
4         min_available_clients=2,
5         min_eval_clients=2,
```

```
6     initial_parameters=get_initial_parameters(),
7 )
8
9 fl.server.start_server(
10     server_address="0.0.0.0:8080",
11     config={"num_rounds": 100},
12     strategy=strategy,
13 )
```

A execução ocorre através do protocolo gRPC, no qual o servidor escuta todas as interfaces de rede na porta 8080. O parâmetro `num_rounds` foi definido como 100, permitindo que o sistema realize sucessivas trocas de coeficientes matemáticos. A cada rodada, o servidor refina a reta de regressão global, permitindo que um veículo se beneficie da experiência de degradação observada pelo outro, sem que nenhum bit de informação privada sobre a rotina de uso das baterias tenha sido transmitido pela rede.

# Capítulo 6

## Resultados e Análise Experimental

Este capítulo apresenta a avaliação quantitativa e qualitativa do protótipo desenvolvido. Os experimentos buscam validar a precisão do modelo de regressão linear em um ambiente de Aprendizado Federado e a eficácia da interface de telemetria operando em tempo real nos dispositivos de borda.

### 6.1 Cenário de Testes e Infraestrutura

A validação experimental foi conduzida em uma rede local dedicada, simulando a infraestrutura de comunicação sem fio de um pátio veicular ou estação de recarga. O servidor de orquestração foi executado em um computador de mesa pessoal (PC), atuando como o nó central de agregação. Os clientes foram implantados em dois hardwares distintos: um Raspberry Pi 4 Model B (Cliente 1) e um Raspberry Pi Zero 2 W (Cliente 2). Essa configuração evidencia a capacidade do sistema de operar de forma estável em um ecossistema de hardware heterogêneo.

A comunicação entre os nós utilizou o framework Flower sobre o protocolo gRPC, garantindo a integridade da troca de parâmetros matemáticos entre a borda e o servidor central. Para a coleta de resultados, o sistema foi configurado para executar 100 rodadas de comunicação. Embora a convergência estatística ocorra em um número reduzido de iterações (12 rodadas), o tempo estendido foi mantido para validar a persistência da interface de usuário e a estabilidade dos processos síncronos nos dispositivos embarcados.

## 6.2 Validação Funcional da Arquitetura de Rede

Antes de analisar a evolução do aprendizado, é fundamental recapitular a modelagem preditiva em execução. O modelo de regressão linear embarcado nos nós clientes recebe como entrada (*features*) matrizes contendo dados locais de telemetria do veículo, englobando variáveis como tensão, corrente, temperatura e tempo de recarga. A partir dessas múltiplas variáveis independentes, o algoritmo tem o objetivo de estimar uma única variável de saída (*target*): o Estado de Saúde da bateria (SOH), representado como um valor percentual contínuo.

Os resultados obtidos demonstram uma convergência consistente dessa estimativa. Conforme ilustrado na Figura 6.1, ambos os dispositivos apresentaram uma queda acentuada no Erro Quadrático Médio (MSE) logo nas primeiras rodadas.

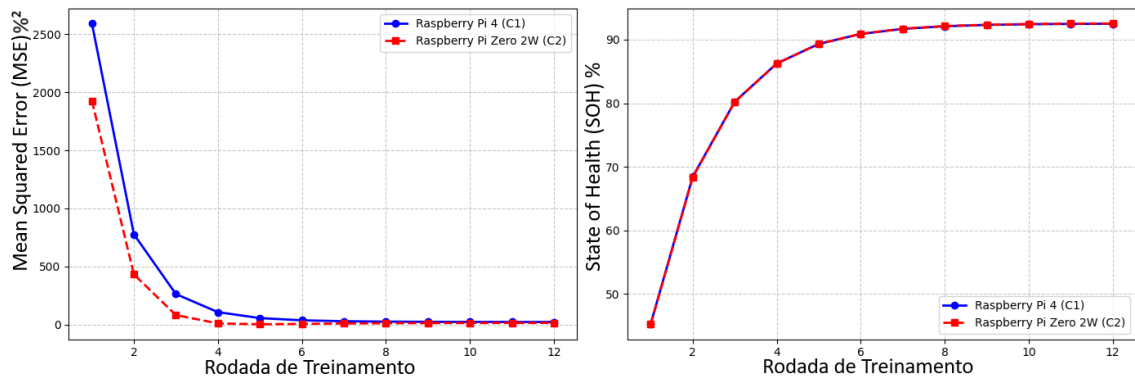


Figura 6.1: Evolução do Erro Quadrático Médio (MSE) e convergência da predição de Saúde da Bateria (SOH) para ambos os clientes

Na execução da prova de conceito, observou-se que o Cliente 1 iniciou com um MSE de 2592,43, enquanto o Cliente 2 partiu de 1925,89. Essa variação inicial na magnitude do erro é esperada em rodadas preliminares, visto que os nós partem de conjuntos de dados locais distintos. Uma observação técnica pertinente diz respeito ao comportamento inicial da predição do SOH, que parte de valores consideravelmente baixos nas primeiras iterações. Esse fenômeno é uma consequência direta da inicialização padrão dos parâmetros do modelo regressor (pesos e viés), que iniciam zerados. Conforme a otimização matemática ajusta esses parâmetros a cada rodada de comunicação, a inferência "escala" rapidamente. Após aproximadamente 12 rodadas de treinamento federado, o comportamento assintótico da curva revela que ambos os clientes estabilizaram o erro de forma sincronizada, com a predição

de SOH convergindo para a faixa de 92,5%.

Como estabelecido na metodologia, o escopo desta avaliação empírica não visa comparar a acurácia final com modelos centralizados de referência (*baselines*), mas sim atestar a viabilidade operacional do fluxo distribuído. A convergência demonstrada no gráfico comprova que a infraestrutura desenvolvida consegue gerenciar a troca de parâmetros via rede local e processar o treinamento no *hardware* restrito do Raspberry Pi. A estabilização do erro evidencia que, para a complexidade específica deste problema de regressão de saúde de bateria, a utilização de algoritmos lineares leves atende plenamente aos requisitos da prova de conceito, validando a execução do ciclo federado na borda.

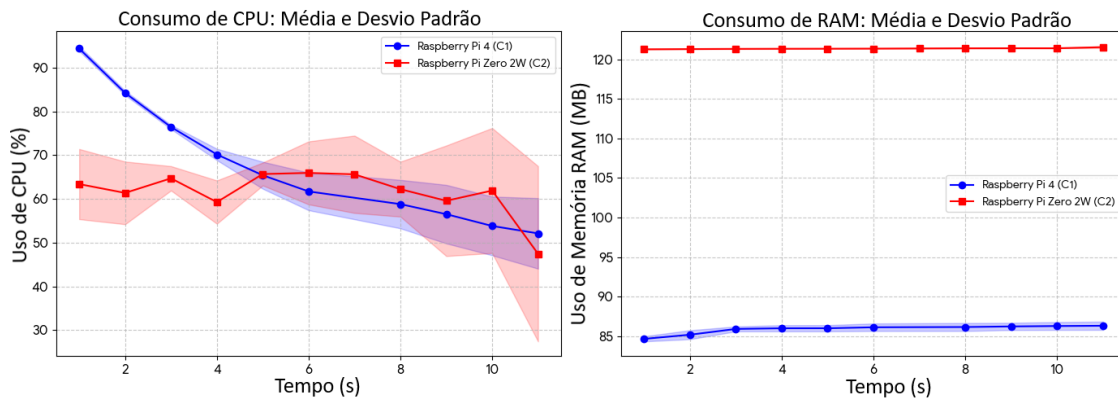


Figura 6.2: Média e desvio padrão do consumo de CPU e RAM durante 5 execuções do ciclo de treinamento nos dispositivos de borda

Para mitigar a variabilidade estatística e garantir o rigor da validação empírica, os testes de monitoramento de *hardware* foram executados em cinco rodadas independentes. A Figura 6.2 ilustra as médias de consumo computacional, com as áreas sombreadas representando o desvio padrão ao longo do tempo. A análise de alocação dinâmica de memória (*tracemalloc*) revelou que as operações durante o treinamento exigiram picos máximos pequenos: uma média de apenas 0,66 MB no Cliente 1 (Raspberry Pi 4) e 0,42 MB no Cliente 2 (Pi Zero 2 W). O consumo total de memória RAM estabilizou-se de forma estática com desvio padrão na faixa de 86,2 MB no Pi 4 e 121,3 MB no Pi Zero. Esses dados confirmam que o modelo adotado opera de forma segura dentro das margens restritas dos hardwares usados.

Sob a ótica do processamento, o monitoramento corroborou a estabilidade ao longo do treinamento. No Cliente 1, que estava renderizando também a in-

terface web, o uso de CPU apresentou um pico inicial em torno de 94,5% provocado pela *startup* da conexão do *Flower*, mas demonstrou um decaimento contínuo, estabilizando-se confortavelmente abaixo de 40% nas etapas finais. O Cliente 2, operando de forma restrita e sem interface, sustentou um fluxo ruidoso porém controlado, oscilando entre 50% e 75% de utilização e finalizando o treinamento em cerca de 11 segundos, sem evidências de *thermal throttling*. A constância observada nessas cinco repetições comprova a robustez arquitetural e a resiliência do sistema embarcado frente ao aprendizado federado.

### 6.3 Validação da Interface Web em Tempo Real

A validação qualitativa do sistema consistiu em verificar se a interface gráfica refletia as mudanças de estado processadas pelos modelos locais. O registro de logs apresentado na Figura 6.3 confirma a execução simultânea e a atualização dos parâmetros em tempo real.

```

pi@RASPMBRASHF-NC: ~$ cat
DEBUG flower 2026-01-12 04:24:31,088 | connection.py:39 | ChannelC
onnectivity.IDLE
DEBUG flower 2026-01-12 04:24:31,094 | connection.py:39 | ChannelC
onnectivity.CONNECTING
DEBUG flower 2026-01-12 04:24:31,120 | connection.py:39 | ChannelC
onnectivity.READY
[Cliente 1] MSE: 2592.4304 | SOH: 45.33%
[Cliente 1] MSE: 777.2285 | SOH: 68.45%
[Cliente 1] MSE: 262.8706 | SOH: 80.25%
[Cliente 1] MSE: 197.8528 | SOH: 86.26%
[Cliente 1] MSE: 56.1687 | SOH: 89.33%
[Cliente 1] MSE: 37.1989 | SOH: 90.90%
[Cliente 1] MSE: 29.4017 | SOH: 91.70%
[Cliente 1] MSE: 25.9096 | SOH: 92.11%
[Cliente 1] MSE: 24.2520 | SOH: 92.32%
[Cliente 1] MSE: 23.4368 | SOH: 92.43%
[Cliente 1] MSE: 23.0279 | SOH: 92.48%
[Cliente 1] MSE: 22.8204 | SOH: 92.51%
[Cliente 1] MSE: 22.7144 | SOH: 92.52%
[Cliente 1] MSE: 22.6601 | SOH: 92.53%
[Cliente 1] MSE: 22.6322 | SOH: 92.54%
[Cliente 1] MSE: 22.6178 | SOH: 92.54%
[Cliente 1] MSE: 22.6104 | SOH: 92.54%
[Cliente 1] MSE: 22.6065 | SOH: 92.54%
[Cliente 1] MSE: 22.6046 | SOH: 92.54%
[Cliente 1] MSE: 22.6035 | SOH: 92.54%
DEBUG flower 2026-01-12 04:24:40,511 | connection.py:121 | gRPC ch
annel closed
INFO flower 2026-01-12 04:24:40,511 | app.py:101 | Disconnect and
shut down
--- TREINAMENTO FINALIZADO ---
O programa continuará rodando para manter o site ativo.

vitor@victorp2: ~$ cat
* Running on http://127.0.0.1:5002
* Running on http://192.168.0.96:5002
Press CTRL+C to quit
DEBUG flower 2026-03-15 19:15:57,072 | connection.py:39 | Cha
nnelConnectivity.READY
[Cliente 2] MSE: 1925.8961 | SOH: 45.25%
[Cliente 2] MSE: 432.2129 | SOH: 68.38%
[Cliente 2] MSE: 81.9110 | SOH: 80.20%
[Cliente 2] MSE: 10.8532 | SOH: 86.25%
[Cliente 2] MSE: 2.7891 | SOH: 89.34%
[Cliente 2] MSE: 6.0729 | SOH: 90.92%
[Cliente 2] MSE: 9.6986 | SOH: 91.73%
[Cliente 2] MSE: 12.0669 | SOH: 92.14%
[Cliente 2] MSE: 13.4155 | SOH: 92.35%
[Cliente 2] MSE: 14.1428 | SOH: 92.46%
[Cliente 2] MSE: 14.5264 | SOH: 92.52%
[Cliente 2] MSE: 14.7245 | SOH: 92.55%
[Cliente 2] MSE: 14.8275 | SOH: 92.56%
[Cliente 2] MSE: 14.8807 | SOH: 92.57%
[Cliente 2] MSE: 14.9081 | SOH: 92.57%
[Cliente 2] MSE: 14.9222 | SOH: 92.57%
[Cliente 2] MSE: 14.9295 | SOH: 92.58%
[Cliente 2] MSE: 14.9333 | SOH: 92.58%
[Cliente 2] MSE: 14.9353 | SOH: 92.58%
[Cliente 2] MSE: 14.9363 | SOH: 92.58%
DEBUG flower 2026-03-15 19:15:59,700 | connection.py:121 | gR
PC channel closed
INFO flower 2026-03-15 19:15:59,701 | app.py:101 | Disconnect
and shut down
--- TREINAMENTO FINALIZADO ---
O programa continuará rodando para manter o site ativo.

```

Figura 6.3: Logs de execução simultânea: convergência do MSE e estabilização do SOH nos dois clientes (Raspberry Pi 4 e Pi Zero)

Para comprovar a eficácia do sistema de cores e do recarregamento automático, a interface foi monitorada desde o estado inicial até a convergência final. A Figura 6.4 apresenta os estados visuais capturados durante a execução. No estado (a), o painel aguarda o início do treinamento; no estado (b), a cor azul indica que o modelo está

sendo ajustado localmente; e no estado (c), a cor verde sinaliza o status saudável após a estabilização do SOH em 92,5%.



Figura 6.4: Estados da interface web durante o ciclo de vida do treinamento federado

O mecanismo de atualização passiva na interface demonstrou ser adequado para a dinâmica da aplicação. É importante ressaltar que, em um cenário de produto final (para um usuário comum), não seria desejável exibir previsões instáveis enquanto o modelo não atinge a convergência. Contudo, essa exposição contínua e intermediária dos dados foi uma escolha metodológica intencional para esta prova de conceito. O objetivo prático de exibir os valores sendo atualizados em tempo real a cada rodada é comprovar, visualmente, o sucesso do desacoplamento arquitetural: a interface *web* consegue ler o *buffer* de memória de forma contínua enquanto a lógica pesada do *framework* federado treina o modelo paralelamente em segundo plano, sem que um processo bloqueie o outro. A renderização do painel e a transição dos estados visuais ocorreram de forma contínua durante todo o ciclo de comunicação, atestando a robustez da engenharia de *software* aplicada na borda.

# Capítulo 7

## Conclusão

### 7.1 Considerações Finais

Este trabalho apresentou o desenvolvimento e a implementação de uma arquitetura de Aprendizado Federado voltada para o monitoramento da saúde de baterias em veículos elétricos. A pesquisa partiu da premissa de que a privacidade do condutor e a soberania sobre os dados de telemetria são requisitos indispensáveis para a próxima geração de sistemas veiculares inteligentes.

A principal contribuição deste estudo transcende o aspecto teórico do aprendizado de máquina, consolidando-se na complexa engenharia de software e hardware necessária para materializar um protótipo funcional. A evolução do projeto, que migrou de uma arquitetura dependente de dispositivos móveis para uma solução nativa em Raspberry Pi operando em conjunto com o Android Auto, exigiu a superação de severos desafios práticos, desde a compatibilidade de bibliotecas legadas até limitações de processamento em borda. A demonstração em bancada provou que é possível executar o treinamento orquestrado sem comprometer a fluidez do sistema multimídia.

Além da robustez da rede federada, a integração de uma interface web dinâmica validou a aplicabilidade prática da solução. O sistema provou ser capaz de fornecer feedback visual imediato e contínuo ao motorista, traduzindo parâmetros de degradação em indicadores simples e passivos. Em suma, a prova de conceito desenvolvida cumpre os objetivos propostos: enquanto o painel garante a usabilidade para o condutor, a arquitetura de Aprendizado Federado consolida-se como uma

alternativa viável e segura aos modelos de processamento centralizado em nuvem.

## 7.2 Trabalhos Futuros

Apesar dos resultados promissores obtidos em ambiente de bancada, a transição desta arquitetura para um produto veicular de produção demanda investigações adicionais. Como desdobramentos naturais desta pesquisa, propõe-se as seguintes frentes de trabalho:

- **Compilação Nativa do Sistema Operacional:** Substituir o uso de distribuições pré-compiladas e legadas, como o Crankshaft, pela compilação do código-fonte original do Android Automotive. Essa abordagem garantirá controle total sobre o kernel do sistema operacional, eliminando restrições de compatibilidade.
- **Integração de Aceleração de Hardware para Visão Computacional:** Com um sistema operacional nativo e atualizado, torna-se viável a integração do acelerador neural Hailo diretamente na mesma placa de processamento da multimídia. Isso permitirá o treinamento e a inferência simultâneos de modelos computacionalmente custosos, como a detecção de Usuários Vulneráveis da Via (VRU), sem sobrecarregar a unidade central de processamento.
- **Aquisição de Dados em Tempo Real:** Embora o ecossistema fechado de determinados veículos elétricos comerciais (como o modelo cedido para testes no âmbito do projeto AVADIP) imponha barreiras à leitura direta do *Battery Management System* (BMS), a arquitetura federada proposta é agnóstica em relação à fonte de dados. Um trabalho futuro essencial é a validação deste sistema em plataformas veiculares com telemetria aberta ou através de parcerias com montadoras para acesso aos protocolos nativos do barramento CAN. Isso permitirá que o cliente embarcado abandone o uso de bases estáticas e realize o treinamento contínuo com as leituras reais e instantâneas dos sensores.
- **Testes de Resiliência em Redes Móveis:** Conduzir ensaios de campo com o veículo em movimento para avaliar o comportamento do protocolo de orquestração federada diante das instabilidades inerentes às redes celulares (4G

e 5G). O objetivo é validar a tolerância a falhas do sistema durante trocas de células de transmissão, variações drásticas de latência e desconexões temporárias em áreas de sombra ou túneis.

# Referências Bibliográficas

- [1] GeeksforGeeks, “Collaborative Learning and Federated Learning”, <https://www.geeksforgeeks.org/machine-learning/collaborative-learning-federated-learning/>, 2024, Acessado em: 15 de março de 2026.
- [2] Grupo de Teleinformática e Automação - GTA, “Tutorial para Gerar Rótulos de VRUs com a Ferramenta CVAT”, [https://www.youtube.com/watch?v=w-v1n0\\_T9Lg&list=PL6207h-p1DtG2XYodBuW4bvJTA52\\_1dVA](https://www.youtube.com/watch?v=w-v1n0_T9Lg&list=PL6207h-p1DtG2XYodBuW4bvJTA52_1dVA), 2025, Acessado em: 15 de março de 2026.
- [3] International Energy Agency, *Global EV Outlook 2025*, Report, IEA, Paris, 2025. Acessado em: 15 de março de 2026.
- [4] HARAZ, A., OTHERS, “State-of-Health and State-of-Charge Estimation in Electric Vehicles Batteries: A Survey on Machine Learning Approaches”, *IEEE Access*, v. 12, pp. 158110–158139, 2024.
- [5] LELLI, E., MUSA, A., BATISTA, E., *et al.*, “On-Road Experimental Campaign for Machine Learning Based State of Health Estimation of High-Voltage Batteries in Electric Vehicles”, *Energies*, v. 16, n. 12, pp. 4639, 2023.
- [6] LIU, J., OTHERS, “Privacy and Security in Internet of Vehicles: Attacks, Countermeasures, and Future Directions”, *IEEE Internet of Things Journal*, v. 10, n. 15, pp. 13504–13525, 2023.
- [7] MCMAHAN, B., MOORE, E., RAMAGE, D., *et al.*, “Communication-Efficient Learning of Deep Networks from Decentralized Data”. In: *Artificial Intelligence and Statistics*, pp. 1273–1282, PMLR, 2017.

- [8] BOCHIE, K., SAMMARCO, M., CAMPISTA, M. E. M., “3FL: Seleção de Clientes Mais Rápidos para Aumento de Desempenho no Aprendizado Federado Cross-Device”. In: *Anais do XLII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2024)*, SBC, 2024.
- [9] WONG, K. L., TSE, R., TANG, S. K., *et al.*, “Decentralized Deep-Learning Approach for Lithium-Ion Batteries State of Health Forecasting Using Federated Learning”, *IEEE Transactions on Transportation Electrification*, , 2024.
- [10] LV, X., CHENG, Y., MA, S., *et al.*, “State of health estimation method based on real data of electric vehicles using federated learning”, *International Journal of Electrochemical Science*, v. 19, n. 8, pp. 100591, 2024.
- [11] SILVA, V., OTHERS, “Privacy-Preserving State Of Health Prediction for Lithium-ion Batteries in electric vehicles using Federated Learning”, *TechReport - Grupo de Teleinformática e Automação (GTA/UFRJ)*, , 2024.
- [12] VILLAS, L. A., OTHERS, *Mobility-aware federated learning in vehicle edge computing*. Ph.D. dissertation, Universidade Estadual de Campinas, 2025.
- [13] TAIK, A., NOUR, B., CHERKAOUI, S., “Edge computing and machine learning for smart cities: A comprehensive survey”, *IEEE Communications Surveys & Tutorials*, , 2020.
- [14] SHI, W., CAO, J., ZHANG, Q., *et al.*, “Edge computing: Vision and challenges”, *IEEE internet of things journal*, v. 3, n. 5, pp. 637–646, 2016.
- [15] KONEČNÝ, J., MCMAHAN, H. B., YU, F. X., *et al.*, “Federated learning: Strategies for improving communication efficiency”, <https://arxiv.org/abs/1610.05492>, 2016, Acessado em: 15 de março de 2026.
- [16] Flower Labs, “Flower: A Friendly Federated Learning Framework”, <https://flower.ai/>, 2026, Acessado em: 15 de março de 2026.
- [17] CASALS, L. C., RODRÍGUEZ, M., CORCHERO, C., *et al.*, “Evaluation of the End-of-Life of Electric Vehicle Batteries According to the State-of-Health”, *World Electric Vehicle Journal*, v. 10, n. 4, pp. 63, 2019.

- [18] GOOGLE, “Android for Cars - App quality guidelines and Template restrictions”, <https://developer.android.com/docs/quality-guidelines/car-app-quality>, 2026, Acessado em: 15 de março de 2026.
- [19] ANDROID, “O que é o Android Automotive?”, [https://source.android.com/docs/automotive/start/what\\_automotive?hl=pt-br](https://source.android.com/docs/automotive/start/what_automotive?hl=pt-br), 2025, Acessado em: 15 de março de 2026.
- [20] CRANKSHAFT, “Crankshaft”, <https://getcrankshaft.com/>, 2026, Acessado em: 15 de março de 2026.
- [21] SZWAJ, M., “Meet OpenAuto, an Android Auto emulator for Raspberry Pi”, <https://opensource.com/article/18/3/openauto-emulator-Raspberry-Pi>, 2018, Acessado em: 15 de março de 2026.
- [22] piwheels, <https://www.piwheels.org/>, 2026, Acessado em: 15 de março de 2026.
- [23] Free Software Foundation, “The GNU C Library (glibc)”, <https://sourceware.org/glibc/>, 2026, Acessado em: 17 de março de 2026.
- [24] Astral Software, “uv: An extremely fast Python package and project manager”, <https://docs.astral.sh/uv/>, 2026, Acessado em: 17 de março de 2026.
- [25] PEDREGOSA, F., OTHERS, “Scikit-learn: Machine Learning in Python”, *Journal of Machine Learning Research*, v. 12, pp. 2825–2830, 2011.
- [26] BATICM, “Battery Charging Data of On-Road Electric Vehicles”, <https://github.com/Baticm/battery-charging-data-of-on-road-electric-vehicles>, 2024, Acessado em: 15 de março de 2026.

# Apêndice A

## Código Fonte do Cliente Federado

A listagem a seguir apresenta a implementação integral do cliente de Aprendizado Federado desenvolvida em linguagem Python, contemplando a configuração do modelo de regressão linear, as regras de avaliação local e o servidor assíncrono para a geração da interface de telemetria veicular.

Código A.1: Script completo do nó de processamento na borda

```
1 import argparse
2 import warnings
3 import flwr as fl
4 import numpy as np
5 import pandas as pd
6 import threading
7 import time
8 import sys
9 from flask import Flask, render_template_string
10 from sklearn.linear_model import SGDRegressor
11 from sklearn.preprocessing import StandardScaler
12 from sklearn.model_selection import train_test_split
13 from sklearn.metrics import mean_squared_error
14
15 warnings.filterwarnings("ignore")
16
17 parser = argparse.ArgumentParser()
18 parser.add_argument('--id', type=int, required=True)
```

```

19 parser.add_argument('--server', type=str,
    default="192.168.0.50:8080")
20 args = parser.parse_args()
21
22 CLIENT_ID = args.id
23 FLASK_PORT = 5000 + CLIENT_ID
24 DATASET_PATH = f"dataset_cliente{CLIENT_ID}.csv"
25
26 app = Flask(__name__)
27
28 DASHBOARD_DATA = {
29     "id": f"CLIENTE #{CLIENT_ID}",
30     "soh": "---",
31     "status": "Aguardando Início...",
32     "color": "#9e9e9e"
33 }
34
35 HTML_TEMPLATE = """
36 <!DOCTYPE html>
37 <html lang="pt-br">
38 <head>
39     <meta charset="UTF-8">
40     <meta name="viewport" content="width=device-width,
    initial-scale=1.0">
41     <meta http-equiv="refresh" content="3">
42     <title>AVADiP Cliente {{ id }}</title>
43     <style>
44         body { background-color: #f4f6f9; font-family:
    'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    color: #333333; display: flex; flex-direction:
    column; align-items: center; justify-content:
    center; height: 100vh; margin: 0; }
45         .container { background: #ffffff; padding: 40px;
    border-radius: 20px; box-shadow: 0 10px 25px

```

```

        rgba(0,0,0,0.1); width: 85%; max-width: 450px;
        text-align: center; border-top: 5px solid
        #009688; }
46 .logo-img { max-width: 180px; margin-bottom: 25px; }
47 .client-id { font-size: 13px; color: #666;
        margin-bottom: 30px; border: 1px solid #ddd;
        display: inline-block; padding: 5px 12px;
        border-radius: 15px; background-color: #f8f9fa;
        font-weight: 600; }
48 .label { font-size: 16px; text-transform: uppercase;
        color: #888; margin-bottom: 5px; letter-spacing:
        1px; font-weight: 600; }
49 .value { font-size: 90px; font-weight: bold; margin:
        10px 0; line-height: 1; color: {{ color }}; }
50 .unit { font-size: 30px; color: #999;
        vertical-align: super; }
51 .status-box { margin-top: 30px; padding: 15px;
        background: #f1f3f5; border-radius: 10px;
        font-size: 16px; font-weight: bold; color: {{
        color }}; text-transform: uppercase;
        letter-spacing: 1px; border: 1px solid #e9ecef; }
52 </style>
53 </head>
54 <body>
55     <div class="container">
56         
59         <br>
60         <div class="client-id">{{ id }}</div>
61         <div class="label">Saúde da Bateria (SOH)</div>
        <div class="value">{{ soh }}<span
            class="unit">%</span></div>
        <div class="status-box">{{ status }}</div>

```

```

62     </div>
63 </body>
64 </html>
65 """
66
67 @app.route('/')
68 def index():
69     return render_template_string(HTML_TEMPLATE,
70                                  **DASHBOARD_DATA)
71
72 def run_flask():
73     app.run(host='0.0.0.0', port=FLASK_PORT, debug=False,
74            use_reloader=False)
75
76 def load_data():
77     try:
78         df = pd.read_csv(DATASET_PATH)
79     except FileNotFoundError:
80         sys.exit(1)
81
82     X = df[['Voltage', 'Current', 'Temperature']].values
83     y = df['SOH'].values
84
85     scaler = StandardScaler()
86     X = scaler.fit_transform(X)
87
88     return train_test_split(X, y, test_size=0.2,
89                             random_state=42)
90
91 class BatteryClient(fl.client.NumPyClient):
92     def __init__(self, X_train, y_train, X_test, y_test):
93         self.model = SGDRegressor(
94             penalty="l2", max_iter=1, warm_start=True,
95             learning_rate="constant", eta0=0.001,

```

```

        random_state=42
92     )
93     self.X_train, self.y_train = X_train, y_train
94     self.X_test, self.y_test = X_test, y_test
95     self.model.partial_fit(np.zeros((1, 3)), [0])
96
97     def get_parameters(self, config):
98         return [self.model.coef_, self.model.intercept_]
99
100    def set_parameters(self, parameters):
101        self.model.coef_ = parameters[0]
102        self.model.intercept_ = parameters[1]
103
104    def fit(self, parameters, config):
105        DASHBOARD_DATA["status"] = "Treinando Modelo..."
106        DASHBOARD_DATA["color"] = "#007bff"
107
108        self.set_parameters(parameters)
109        self.model.partial_fit(self.X_train, self.y_train)
110        return self.get_parameters(config={}),
111           len(self.X_train), {}
112
113    def evaluate(self, parameters, config):
114        self.set_parameters(parameters)
115        y_pred = self.model.predict(self.X_test)
116        mse = mean_squared_error(self.y_test, y_pred)
117
118        estimated_soh = np.mean(y_pred)
119
120        DASHBOARD_DATA["soh"] = f"{estimated_soh:.1f}"
121
122        if estimated_soh >= 90:
123            DASHBOARD_DATA["status"] = "BATERIA SAUDÁVEL"
124            DASHBOARD_DATA["color"] = "#28a745"

```

```

124     elif estimated_soh >= 75:
125         DASHBOARD_DATA["status"] = "DEGRADAÇÃO LEVE"
126         DASHBOARD_DATA["color"] = "#fd7e14"
127     else:
128         DASHBOARD_DATA["status"] = "CRÍTICO: MANUTENÇÃO"
129         DASHBOARD_DATA["color"] = "#dc3545"
130
131     return float(mse), len(self.X_test), {"mse":
132         float(mse)}
133
134 def main():
135     X_train, X_test, y_train, y_test = load_data()
136
137     site_thread = threading.Thread(target=run_flask,
138         daemon=True)
139     site_thread.start()
140
141     try:
142         fl.client.start_numpy_client(
143             server_address=args.server,
144             client=BatteryClient(X_train, y_train, X_test,
145                 y_test)
146         )
147     except Exception as e:
148         pass
149
150     DASHBOARD_DATA["status"] = "Treinamento Finalizado"
151     DASHBOARD_DATA["color"] = "#28a745"
152
153     try:
154         while True:
155             time.sleep(10)
156     except KeyboardInterrupt:
157         pass

```

155

156 `if __name__ == "__main__":`

157  `main()`

# Apêndice B

## Código Fonte do Servidor de Orquestração

A listagem a seguir apresenta o código integral do servidor responsável pela orquestração do Aprendizado Federado, utilizando a estratégia FedAvg para a agregação dos modelos de regressão linear.

Código B.1: Script completo do servidor de agregação Flower

```
1 import flwr as fl
2 import numpy as np
3 from sklearn.linear_model import SGDRegressor
4
5 def get_initial_parameters():
6     model = SGDRegressor(
7         penalty="l2",
8         max_iter=1,
9         warm_start=True,
10        learning_rate="constant",
11        eta0=0.001,
12        random_state=42,
13    )
14    model.partial_fit(np.zeros((1, 3)), [0])
15    return fl.common.weights_to_parameters([model.coef_,
16        model.intercept_])
```

```
17 if __name__ == "__main__":
18     strategy = fl.server.strategy.FedAvg(
19         min_fit_clients=2,
20         min_available_clients=2,
21         min_eval_clients=2,
22         initial_parameters=get_initial_parameters(),
23     )
24
25     fl.server.start_server(
26         server_address="0.0.0.0:8080",
27         config={"num_rounds": 100},
28         strategy=strategy,
29     )
```