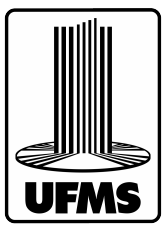
André Luís Violin

Criação de uma abordagem de reuso de artefatos de software baseado em Micro-Frontends para Internet das Coisas na Pecuária Digital



UNIVERSIDADE FEDERAL
DE MATO GROSSO DO SUL

Campo Grande – MS

Maio de 2025

André Luís Violin

Criação de uma abordagem de reuso de artefatos de software baseado em Micro-Frontends para Internet das Coisas na Pecuária Digital

Dissertação de mestrado apresentada ao Programa de Mestrado Stricto Sensu em Computação Aplicada, da Faculdade de Computação, mantido pela Fundação Universidade Federal de Mato Grosso do Sul, como parte dos requisitos para a obtenção do título de Mestre em Computação Aplicada (Área de Concentração: Aplicações Distribuídas).

Universidade Federal de Mato Grosso do Sul – UFMS

Faculdade de Computação – FACOM

Programa de Pós-Graduação em Computação Aplicada

Orientador: Profa. Dra. Hana Karina Salles Rubinsztejn

Coorientador: Dr. Camilo Carromeu

Campo Grande – MS

Maio de 2025

André Luís Violin
Criação de uma abordagem de reuso de artefatos de software baseado em Micro-Frontends para Internet das Coisas na Pecuária Digital/ André Luís Violin. – Campo Grande – MS, Maio de 2025-
75p. : il. (algumas color.) ; 30 cm.

Orientador: Profa. Dra. Hana Karina Salles Rubinsztejn
Coorientador: Dr. Camilo Carromeu

Dissertação (Mestrado) – Universidade Federal de Mato Grosso do Sul – UFMS
Faculdade de Computação – FACOM
Programa de Pós-Graduação em Computação Aplicada, Maio de 2025.

1. Pecuária de Precisão. 2. Micro frontend. I. Profa. Dra. Hana Karina Salles Rubinsztejn. II. Universidade Federal de Mato Grosso do Sul. III. Faculdade de Computação. IV. Criação de uma abordagem de reuso de artefatos de software baseado em Micro-Frontends para Internet das Coisas na Pecuária Digital.

André Luís Violin

Criação de uma abordagem de reuso de artefatos de software baseado em Micro-Frontends para Internet das Coisas na Pecuária Digital

Dissertação de mestrado apresentada ao Programa de Mestrado Stricto Sensu em Computação Aplicada, da Faculdade de Computação, mantido pela Fundação Universidade Federal de Mato Grosso do Sul, como parte dos requisitos para a obtenção do título de Mestre em Computação Aplicada (Área de Concentração: Aplicações Distribuídas).

Profa. Dra. Hana Karina Salles Rubinsztejn
Orientador

Dr. Camilo Carromeu
Coorientador

Prof. Dr. Evandro Mazina Martins
Membro Interno

Prof. Dr. Márcio Aparecido Inácio da Silva
Membro Externo

Campo Grande – MS
Maio de 2025

Agradecimentos

Ao longo desta jornada acadêmica, encontrei apoio, incentivo e orientação de pessoas essenciais, às quais expresso minha profunda gratidão.

À minha amada esposa Gabriela, por seu amor incondicional, paciência, dedicação e apoio em cada etapa deste desafio. Sua presença e incentivo tornou a jornada mais leve e as conquistas ainda mais significativas. Sem você, este caminho não seria finalizado.

À minha filha, Maria Elisa, que chegou no meio deste processo e foi fonte de inspiração, alegria e força. Que este trabalho sirva como exemplo de persistência para você, da mesma forma como seu amor e sua presença foram essenciais para finalizar essa jornada.

À toda minha família, pelo amor incondicional e por acreditar em mim em todos os momentos, oferecendo suporte e inspiração para seguir em frente.

À minha orientadora, Dra. Hana Karina Salles Rubinsztejn, expresso minha profunda gratidão por sua orientação, paciência e incentivo ao longo desta jornada acadêmica. Sua experiência e comprometimento foram fundamentais para o desenvolvimento deste trabalho.

Ao meu coorientador, Dr. Camilo Carromeu, pelos direcionamentos e apontamentos técnicos que permitiram que este trabalho fosse concluído com um propósito e pudesse contribuir com plataforma e-Cattle.

Obrigado a ambos por acreditarem em meu potencial e por contribuir de maneira tão significativa para esta conquista.

E, por fim, a todos àqueles que, direta ou indiretamente, contribuíram para que esta dissertação se tornasse realidade. Minha sincera gratidão!

Resumo

Ao longo dos anos, a pecuária de corte brasileira vem se destacando no mercado mundial de carne bovina. Dada a importância do setor para a economia nacional, criar e aperfeiçoar os recursos adotados na cadeia de produção de gado de corte é fundamental para ampliar seu potencial produtivo, diminuir os custos e garantir o progresso contínuo imposto pelo mercado. Neste domínio, ter acesso aos dados coletados na propriedade produtora de carne bovina contribui para o monitoramento, gestão e tomada de decisões. A parceria entre a FACOM/UFMS e a Embrapa Gado de Corte gera diversas pesquisas e projetos alinhados à pecuária de precisão, como a plataforma e-Cattle. Este trabalho teve por objetivo a produção de uma ferramenta , via CLI, para automatizar a criação de aplicações *frontend* para a plataforma e-Cattle. Dessa forma, foi realizado um estudo de caso pautado em uma arquitetura de micro-frontends integrada com os recursos de aplicação web progressiva (PWA). Deste estudo, foram criados modelos padronizados baseados em uma abordagem de reuso de artefatos de software para serem utilizados pela CLI criada.

Palavras-chave: Pecuária de Precisão. Micro-Frontend. Internet das Coisas. Progressive Web App.

Abstract

Over the years, Brazilian beef cattle farming has stood out in the global beef market. Given the sector's importance to the national economy, creating and improving the resources adopted in the beef cattle production chain is essential to expand its productive potential, reduce costs, and ensure the continuous progress imposed by the market. In this domain, having access to data collected from the beef-producing property contributes to monitoring, management, and decision-making. The partnership between FACOM/UFMS and Embrapa Gado de Corte generates various research initiatives and projects aligned with precision livestock farming, such as the e-Cattle platform. This work aimed to produce a CLI to automate the creation of frontend applications for the e-Cattle platform. To achieve this, a case study was conducted based on a micro-frontends architecture integrated with progressive web application (PWA) resources. From this study, standardized models were developed based on an artifact reuse approach to be used by the created CLI.

Keywords: Precision Livestock. Micro-Frontend. Internet of Things. Progressive Web App.

Lista de ilustrações

Lista de tabelas

Lista de Abreviaturas e Siglas

ABIEC	Associação Brasileira das Indústrias Exportadoras de Carne
API	*Application Programming Interface* - Interface de Programação de Aplicações
CLI	*Command-Line Interface*
DT	Digital Twin
Embrapa	Empresa Brasileira de Pesquisa Agropecuária
ES	*EcmaScript*
ESM	*EcmaScript Module*
HMR	*Hot Module Replacement*
IA	Inteligência Artificial
IoT	Internet das Coisas
JSON	*JavaScript Object Notation*
JWT	JSON Web Token
PIB	Produto Interno Bruto
PWA	Aplicativos Web Progressivos
SEO	Otimização para Mecanismos de Busca
SOA	Arquitetura Orientada a Serviços
SPA	*Single Page Application*
TEC	Toneladas Equivalente Carcaça
TI	Tecnologia da Informação
TIC	Tecnologias da Informação e Comunicação
UFMS	Universidade Federal de Mato Grosso do Sul

Sumário

1 Introdução

Este capítulo pretende fornecer uma visão geral do documento, o contexto atual deste trabalho e explicar o problema em que esta dissertação se baseia. Posteriormente serão apresentados os objetivos, abordagem e processo de desenvolvimento definido.

1.1 Contexto

A Plataforma e-Cattle (CARROMEU, 2019) propõe a implementação de uma infraestrutura de Internet das Coisas (IoT) de baixo custo, destinada ao monitoramento e suporte à automação dos processos de produção em propriedades de criação de bovinos. O projeto visa integrar diversas tecnologias aplicadas à pecuária para resolver problemas relacionados ao uso de sensores para a coleta autônoma de dados no campo, processamento desses dados em softwares de alto nível para o monitoramento e automação no manejo pecuário baseado nas informações analisadas.

Dessa forma, o projeto e-Cattle surge da demanda por um ecossistema digital que apoie a pecuária de corte, promovendo a transformação digital no setor agropecuário e simplificando a complexidade computacional envolvida. Para isso, foi desenvolvida uma arquitetura baseada em barramentos de serviços, destinada a facilitar a comunicação entre os diferentes componentes tecnológicos da plataforma de automação rural. Esse desenho arquitetônico permite que a plataforma seja evolutiva e integrada, possibilitando a implementação de técnicas de pecuária de precisão na produção de carne bovina brasileira.

Por isso, a arquitetura proposta pelo e-Cattle é multicamada e utiliza tecnologias modernas de comunicação de dados, garantindo a independência entre as camadas. Um componente importante desta infraestrutura é o *middleware* BigBoxx, responsável pela implementação dessas camadas e viabilidade da implantação do sistema e-Cattle em propriedades rurais envolvidas na cadeia produtiva da pecuária de corte (CARROMEU, 2019).

Para justificar o desenvolvimento de tecnologias e plataformas como a e-Cattle, a seguir serão apresentados alguns números e valores gerados pela pecuária de corte. Conforme os dados mais recentes do relatório anual realizado pela Associação Brasileira das Indústrias Exportadoras de Carne (ABIEC), o sistema agroindustrial da carne bovina foi responsável por $8,2\%$ do Produto Interno Bruto (PIB), isto corresponde a R$ 895 bilhões do total de toda riqueza gerada pelo Brasil, mensurada em R$ $10,9$ trilhões em 2023. Nesse mesmo ano, chegou-se à marca histórica da ordem de US$ $10,55$ bilhões de faturamento em exportações, reforçando a posição de líder global e confirmando a

relevância, consistência e solidez deste setor (ABIEC, 2024).

Além disso, a pecuária de corte movimenta toda uma cadeia de produção para além da venda de bovino, fazendo com que os valores movimentados sejam muito mais vultosos. Nutrição, sanidade animal e investimentos em genética são alguns dos insumos e serviços utilizados pela pecuária e estão entre os negócios que compõem essa cadeia produtiva (ABIEC, 2024).

Com o segundo maior rebanho de bovinos do mundo, o Brasil possui cerca de 197,2 milhões de cabeças, representando 11, 9% do rebanho mundial. A produção de carne bovina fica atrás somente da produção norte-americana, de 12, 2 milhões de Toneladas Equivalente Carcaça (TEC), enquanto o Brasil, no mesmo período, obteve uma produção de 10.6 milhões de TEC. Entretanto, o maior destaque da pecuária de corte nacional é nas exportações, com vultosos 28, 53% das exportações mundiais em 2023, sendo o maior fornecedor de carne bovina. Examinando o cenário nacional, o estado de Mato Grosso do Sul (MS) possui o terceiro maior rebanho, com 11, 28% do total produzido no Brasil (ABIEC, 2024).

A inovação aliada ao uso de tecnologias é fundamental para manter o crescimento econômico e produtivo. As empresas que desejam se manter relevantes e competitivas no mercado utilizam a capacidade de inovar como principal ferramenta. O setor de agronegócio também vem adotando esta postura em seus meios de produção e nas propriedades. Neste cenário, a transformação digital exerce um papel fundamental na forma como o agronegócio conduz seu processo de criação, administração e negócios.

Isto posto, este trabalho gerou uma *Command-Line Interface* (CLI) e alguns módulos padronizados para automatizar a criação de aplicações utilizando arquitetura de micro-frontends para produzir Aplicativos Web Progressivos (PWA) para a plataforma e-Cattle. Estimulando, assim, a formação de uma comunidade de desenvolvedores que irão criar novos artefatos que poderão ser reutilizados e ressignificados para serem disponibilizados no repositório da plataforma e-Cattle. Este projeto provê uma solução para a camada de Aplicação da plataforma, tendo como público programadores e *software houses*. Todas as camadas serão explicadas no decorrer desta dissertação.

1.2 Descrição do Problema

Atualmente, as empresas enfrentam algo novo ao adotarem tecnologias disruptivas e acessíveis, tais como: Internet das Coisas (*Internet das Coisas* - IoT), Inteligência Artificial (IA), nanotecnologia, Computação Móvel, *Machine Learning* e Nuvem. Este processo de digitalização inevitável ao qual as empresas deverão se submeter para se manterem relevantes no mercado (BUCCI; BENTIVOGLIO; FINCO, 2018).

A adoção de tecnologias modernas, como robótica, sistemas embarcados, sensores e análise de dados, pode ser utilizada pelo produtor como apoio para agilizar a tomada de decisões. Além disso, pode modernizar a propriedade rural e, nas áreas relacionadas ao processo de produção, poderá proporcionar economia e agregar valor à produção (SHAMSHIRI et al., 2018). Tais tecnologias são responsáveis pelo crescimento da produção e da receita, inclusive em muitas empresas que não pertencem ao setor de Tecnologias da Informação e Comunicação (TIC) (MILLER; ATKINSON, 2014).

A arquitetura da plataforma e-Cattle é escalável e conhecer os padrões comuns entre as camadas que a compõem permite definir interfaces genéricas para incluir novos sensores (CARROMEU, 2019). Essas interfaces facilitam a inclusão de novos produtos à medida que são desenvolvidos. A arquitetura multicamada proposta é estruturada para otimizar o processo de coleta, processamento e utilização de dados na pecuária, começando pela Camada de Sensoriamento. Esta camada fundamental se encarrega da obtenção dos dados brutos por meio de sensores, estabelecendo os padrões de comunicação iniciais baseados na caracterização dos sensores utilizados.

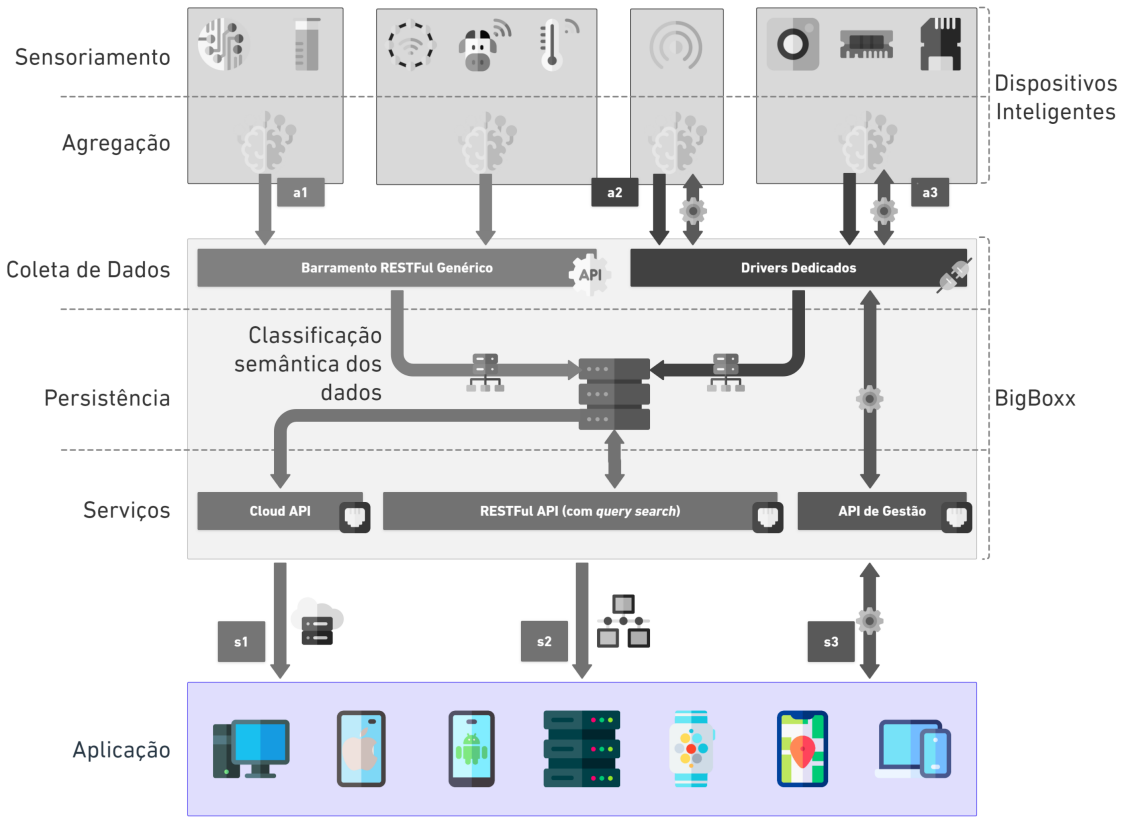
A identificação de padrões nas camadas subsequentes é um reflexo do portfólio existente de projetos independentes que utilizam sensores para abordar problemas específicos na pecuária. Ou seja, a definição desses padrões baseia-se nas similaridades e requisitos comuns identificados durante o desenvolvimento de diversos artefatos de software, contribuindo para uma melhor integração e interoperabilidade entre eles.

A arquitetura projetada para a plataforma e-Cattle é composta de seis camadas distintas, como pode ser visto na Figura 1, cada uma com seus componentes específicos, seja de hardware ou software. Estas camadas são de:

1. **Sensoriamento**: responsável pela coleta inicial dos dados brutos.
2. **Agregação**: onde os dados de diferentes sensores são combinados.
3. **Coleta de Dados**: encarregada da organização dos dados agregados.
4. **Persistência**: onde os dados são armazenados para uso futuro.
5. **Serviços**: oferece serviços baseados nos dados coletados.
6. **Aplicação**: onde as funcionalidades finais são apresentadas ao usuário.

Essa estrutura busca simplificar o desenvolvimento e manutenção da plataforma, aumentar sua flexibilidade para atualizações futuras e melhorar a interoperabilidade com novas tecnologias (CARROMEU, 2019). O foco principal na camada de Aplicação. Esta camada contém os *softwares* de alto nível (aplicativos) que recorrem aos dados de sensores. Tais aplicações podem combinar informações de outras fontes, como, por exemplo: valor

Figura 1 – Arquitetura projetada para a plataforma e-Cattle.



Fonte: Adaptado de (CARROMEU, 2019).

de arroba do boi obtido por meio de *crawlers*; barramentos de dados de terceiros; outros bancos de dados da propriedade ou mesmo informações inseridas pelo usuário; entre outras. Ou seja, esta camada poderá auxiliar o produtor na geração de indicadores, na tomada de decisão, na emissão de alerta, avisos de manejo, dentre muitas outras ações relacionadas à inteligência do negócio (CARROMEU, 2019).

Portanto, a evolução tecnológica em propriedades rurais e em sua cadeia produtiva gera muitos dados brutos que devem ser transformados em informações claras e objetivas para os produtores rurais. É necessário que elas sejam disponibilizadas em um ambiente de fácil visualização e com total acessibilidade. Para isso, as aplicações *web* e suas tecnologias oferecem excelentes recursos para exibição, filtragem e acesso à informação.

No decorrer dos anos, a evolução do ecossistema *frontend* permitiu a inserção de diferentes arquiteturas como *Single-Page Applications* (Aplicativos de página única), *Isomorphic Applications* (Aplicações Isomórficas), *Static-Page Websites* (sites de páginas estáticas) e Jamstack que resolveram problemas diversos. Entretanto, é necessária uma solução que possibilite o dimensionamento de projetos entre dezenas ou centenas de desenvolvedores trabalhando (MEZZALIRA, 2021).

1.3 Objetivo

Construção de uma abordagem de reuso de artefatos de *software* para plataforma e-Cattle baseado em Micro-Frontends e PWA para aplicações de Internet das Coisas (IoT).

1.4 Objetivos Específicos

- Implementação e configuração de um estudo de caso utilizando arquitetura de micro-frontends e PWA;
- Criação de modelos *host* e *remote* reutilizáveis adequados a dados sensoriais no contexto da agropecuária, baseados no estudo de caso desenvolvido;
- Desenvolvimento de uma ferramenta CLI para uso dos modelos desenvolvidos;
- Tornar a ferramenta acessível e disponibilizar, por meio de repositório, os módulos e aplicações.

1.4.1 Organização do Texto

No Capítulo 1, foi introduzido o tema desta dissertação, bem como o problema descrição do problema e os objetivos deste trabalho. O capítulo seguinte, Capítulo 2, o contexto é detalhado, incluindo a descrição das tecnologias utilizadas no desenvolvimento do projeto. Já o Capítulo 3 apresenta a revisão literária acerca das soluções existentes no mercado, os principais trabalhos também são apresentados e comparados com as tecnologias que poderiam ser adotadas para atingir os objetivos em questão. Enquanto o Capítulo 4, a plataforma e-Cattle é detalhada. São explicadas suas características e aplicabilidade, bem como as camadas que se correlacionam com este projeto são discutidas. O Capítulo 5 analisa os detalhes relacionados à implementação do estudo de caso, os modelos criados, a solução final e sua disponibilidade para outros desenvolvedores. E, por fim, no Capítulo 6, são apresentadas as considerações finais e sugestões de evolução sobre este projeto.

2 Fundamentação Teórica

Antigamente, o termo designado para aplicações *web* que os distinguia de sites corporativos tradicionais mais estáticos era *rich internet applications* (RIAs - aplicativos ricos para internet). Hoje, existem muitos RIAs, ou aplicações *web*, na rede mundial de computadores. Os serviços on-line se proliferaram, o que possibilita imprimir cartões de visita sob demanda, assistir filmes ou eventos ao vivo, pedir pizza, gerenciar contas bancárias e muitas outras atividades que auxiliam nas tarefas diárias (MEZZALIRA, 2021).

Ao iniciar um novo projeto, pode-se optar por um aplicativo de página única ou um isomórfico, em que o código pode ser executado no servidor e no cliente, ou até mesmo criar várias páginas estáticas para serem executadas em uma infraestrutura de nuvem ou local. Embora haja uma gama muito ampla de opções, nem todas são adequadas a todos os trabalhos (MEZZALIRA, 2021). Para tomar a decisão correta de qual tipo de aplicação escolher para um projeto, é necessário conhecer melhor cada uma delas.

Como mencionado anteriormente, este capítulo irá apresentar uma descrição do contexto tecnológico, motivando a escolha da arquitetura utilizada durante o desenvolvimento do projeto, bem como a pilha tecnológica presente no *frontend*.

A seguir serão apresentados, sem maiores detalhes, os principais tipos e arquiteturas de aplicações *frontend*, demonstrando a evolução gradual que acorreu ao longo dos últimos anos. Além disso, serão mostradas as vantagens e desvantagens acerca das arquiteturas e a motivação da escolha de micro-frontends para este projeto. Vale ressaltar que cada uma tem sua relevância e deve ser escolhida após as devidas análises técnicas e organizacionais acerca do projeto que será construído.

2.1 Principais Tipos de Aplicações *Frontend*

Esta seção traz os principais tipos de aplicação *frontend*, tais como as de página única, as isomórficas, as estáticas e as Jamstacks. Como será visto, cada uma possui suas peculiaridades e, portanto, caberá aos desenvolvedores ou equipe entender as necessidades específicas do projeto que está implementando. Considerando as especificidades do projeto, como a localização onde a plataforma será utilizada, e as recomendações do setor de Tecnologia da Informação (TI) da Empresa Brasileira de Pesquisa Agropecuária (Embrapa), optou-se por criar PWAs.

2.1.1 Aplicações de Página Única

As *Single Page Application*s (SPAs) são compostas de um único ou alguns arquivos JavaScript que encapsulam toda a aplicação *frontend*, normalmente baixados antecipadamente. Quando os servidores *web* ou a rede de entrega de conteúdo (CDN - *content delivery network*) recebem o pedido de uma de índice HTML, o SPA carrega o JavaScript, CSS e quaisquer arquivos adicionais necessários para mostrar todas as partes da aplicação.

De acordo com Mezzalira (2021), normalmente as SPAs se comunicam com *Application Programming Interface* - Interface de Programação de Aplicações (API), trocando dados com a camada de persistência do servidor. Este tipo de aplicação também evita várias requisições ao servidor para carregar lógica adicional e renderiza todas as visualizações instantaneamente durante o ciclo de vida da aplicação. Tais recursos melhoram a experiência do usuário e simulam o que normalmente se tem ao interagir com um aplicativo nativo para dispositivos móveis ou *desktop*, em que é possível ir de uma parte a outra da aplicação sem precisar aguardar muito. Além disso, uma SPA gerencia o roteamento no lado do cliente e também permite decidir a divisão da lógica da aplicação entre o servidor e o cliente. Outro benefício é que o cliente baixa o código do aplicativo apenas uma vez, no início do seu ciclo de vida, e toda a lógica do aplicativo fica disponível para toda a sessão do usuário.

Entretanto, há algumas desvantagens ao adotar essa arquitetura para determinados tipos de aplicativos. O tempo de carregamento inicial pode ser maior que em outras arquiteturas, uma vez que o aplicativo inteiro está sendo baixado ao invés de apenas o que o usuário quer ver. Caso a aplicação não seja bem projetada, o tempo de *download* pode comprometer a aplicação, especialmente se for carregada em conexão instável ou não confiável por meio de dispositivos móveis. Este tipo de aplicação também leva desvantagem quanto à Otimização para Mecanismos de Busca (SEO), pois como a aplicação é normalmente montada no cliente via JavaScript, o *crawler* tem dificuldade de indexar o conteúdo. Em grandes aplicações, baixar toda a lógica da aplicação também pode ser uma desvantagem, pois pode haver vazamento de memória quando o usuário pular de uma visualização para outra se o código não for bem implementado. A última desvantagem é no lado organizacional, pois se for uma aplicação grande gerenciada por equipes distribuídas trabalhando na mesma base de código, áreas diferentes da aplicação podem misturar as abordagens e decisões, o que pode confundir os membros da equipe (MEZZALIRA, 2021).

2.1.2 Aplicações Isomórficas

Também conhecidas como universais, são aplicações *web* em que o código é compartilhado entre servidor e cliente e pode rodar em ambos os contextos. Em virtude da possibilidade de gerar a página no lado servidor, isto se torna benéfico para o tempo de

interação, teste A/B[1] e SEO, uma vez que a responsabilidade em otimizar o código para as principais características do projeto é da equipe de desenvolvimento. Por compartilharem código entre o servidor e o cliente, permite que o servidor, por exemplo, renderize a página solicitada pelo navegador, recupere os dados a serem exibidos do banco de dados ou de um microsserviço, agregue-os e, em seguida, pré-renderize-os com o sistema de modelo usado para gerar a visualização, a fim de fornecer ao cliente uma página que não precisa de viagens de ida e volta adicionais para solicitar dados a serem exibidos (MEZZALIRA, 2021).

Uma vez que a página solicitada é pré-renderizada no lado do servidor e é parcial ou totalmente interpretada no *backend*, o tempo de interação é aprimorado. Evitando muitas requisições do *frontend*, então não será necessário carregar recursos adicionais (fornecedores, código do aplicativo, etc.), e o navegador pode interpretar uma página estática com quase tudo dentro. Mezzalira (2021) também cita os prós e contras de aplicações isomórficas.

Entre as vantagens, estão:

- Otimização para Mecanismos de Busca (SEO) melhorada: graças à renderização no lado do servidor, sem a necessidade de solicitações adicionais do servidor;
- Compartilhamento de código entre contextos: permite usar essa técnica em uma abordagem híbrida, podendo renderizar parte da página no lado do servidor para melhorar o tempo de interação e, em seguida, carregar arquivos JavaScript adicionais por meio de *lazy-load*[2];
- Roteamento no lado do servidor: o reteamento entre conteúdos pode ser gerenciado totalmente no lado do servidor, servindo apenas uma página estática sempre que o usuário interagir com um link no cliente;
- Renderização mista: a renderização da primeira visualização carregando um SPA pode ser feita do lado servidor com o roteamento global para outras SPA, em cada uma tem seu próprio sistema de roteamento para navegar entre as visualizações;
- Integração com teste A/B: integrar plataformas de teste A/B não exige muito esforço.

Já as desvantagens listadas pelo autor, tem-se:

- Escalabilidade: esse tipo de aplicação pode ter problemas se houver grande demanda por seu conteúdo;
- Organização: possui os problemas similares aos de uma SPA cuja base de código é única e mantida por várias equipes.

[1] É o ato de executar um experimento simultâneo entre duas ou mais variantes de uma página para ver qual delas tem o melhor desempenho (MEZZALIRA, 2021).

[2] É uma técnica onde componentes ou módulos apenas são carregados quando realmente são necessários.

2.1.3 Sites de Páginas Estáticas

Talvez a opção mais antiga e ainda utilizada, onde sempre que um usuário clica em um link, uma nova página é carregada. Um site de página estática é útil para sites rápidos que não devem ficar on-line por um longo período, como aqueles que anunciam um produto ou serviço específico, também conhecidos como *hotsites*, que deve destacar-se sem usar o site corporativo ou que devem ser simples e fáceis de construir e manter pelo usuário final.

Nos últimos anos, esse tipo de site se transformou em uma única página que se expande verticalmente em vez de carregar páginas diferentes. Alguns desses sites também carregam o conteúdo lentamente, usando a técnica de *lazy-loading*, esperando até que o usuário role para uma posição específica para carregar o conteúdo. A mesma técnica é usada com *hiperlinks*, onde todos os *links* são ancorados dentro da mesma página e o usuário navega rapidamente entre *bits* de informação disponíveis no site. Esses tipos de projetos são geralmente criados por pequenas equipes ou colaboradores individuais. O investimento no lado técnico é bastante baixo, e é um bom *playground* para desenvolvedores experimentarem novas tecnologias ou novas práticas, ou ainda consolidarem as existentes (MEZZALIRA, 2021).

2.1.4 Jamstack (JavaScript, APIs e Markup)

Essa arquitetura surgiu nos últimos anos com ótimos resultados. O Jamstack pretende ser uma arquitetura moderna para auxiliar na criação de sites rápidos e seguros e aplicativos dinâmicos com JavaScript/API e marcação pré-renderizada, servidos sem servidores web. Seu resultado é um artefato estático composto de HTML, CSS e JavaScript. O artefato pode ser fornecido diretamente por um CDN, já que o aplicativo não requer nenhuma tecnologia do lado do servidor para funcionar. Uma das maneiras mais simples de servir um aplicativo Jamstack é usar páginas do GitHub para hospedar o resultado final. Nesta categoria, encontram-se soluções populares como os *frameworks*: Gatsby.js, Next.js ou Nuxt.js (MEZZALIRA, 2021). Tendo como vantagens:

- Melhor desempenho e infraestrutura e manutenção mais baratas, já que podem ser atendidas diretamente por uma CDN;
- Grande escalabilidade porque atendem arquivos estáticos;
- Maior segurança devido à diminuição da superfície de ataque; e
- Fácil integração com um sistema de gerenciamento de conteúdo (CMS) *headless*[3].

[3] É um sistema de gerenciamento de conteúdo da *web* somente *backend* que atua principalmente como um repositório de conteúdo.

Dado seu desacoplamento, uma vez que o desenvolvedor *frontend* precisa se concentrar apenas na criação e depuração do *frontend*, essa arquitetura geralmente proporciona uma abordagem mais focada no resultado.

2.1.5 Aplicativos Web Progressivos

Para que uma aplicação *web* seja considerada progressiva, ela deve ser instalável e confiável para todos os usuários, independente do tamanho da tela e da entrada. A seguir serão listados os principais requisitos, propostas pela Google (2021), que definem uma PWA:

- Desempenho: focar na otimização de métricas de desempenho centradas no usuário é fundamental para o sucesso de qualquer experiência on-line, uma vez que sites com alto desempenho engajam e retêm melhor os usuários;
- *Cross-browser*: PWAs devem funcionar em todos os navegadores, mesmo que a experiência não seja idêntica neles todos, podendo, inclusive, haver recursos que não sejam compatíveis com um navegador, mas com um substituto que garanta uma boa experiência;
- Responsivo: deve ser acessível em qualquer tamanho de tela, e todo o conteúdo deve estar disponível em qualquer tamanho de janela de visualização;
- Página personalizada: se o usuário estiver off-line, mantê-lo na PWA promove uma experiência mais integrada e semelhante a um aplicativo do que ser encaminhado para uma página off-line padrão do navegador;
- Instalável: permitir que o usuário instale ou adicione a aplicação à tela inicial faz com que haja mais interação do usuário e aproveitamento dos recursos, gerando melhor experiência de uso.

Estes são os requisitos fundamentais que uma PWA deve possuir. Entretanto, para uma aplicação ótima, que tenha a aparência de um aplicativo de ponta, é necessário incorporar outras características além das principais. A lista de verificação ideal de uma PWA consiste em tornar a aplicação mais eficiente e confiável. Google (2021) propõe as seguintes características ideais para uma PWA:

- Conexão: identificar os recursos que não exigem conectividade e permitir que ao menos algumas funcionalidades da aplicação sejam usadas off-line cria uma experiência autêntica semelhante a um aplicativo nativo;

- Acessibilidade: essa característica garante que a aplicação possa ser usada por todos, para isso é preciso verificar se o conteúdo e interações da aplicação são entendidas por leitores de tela, utilizáveis somente com o teclado, que o foco está indicado e que o contraste de cores é forte;

- Recursos avançados: sempre que disponível, usar ferramentas para criar PWAs altamente capazes e totalmente integradas, como mensagens *push*, WebAssembly e WebGL, acesso a sistemas de arquivos, seletores de contato e integração com *app store*, permite criar uma experiência de usuário completa, antes restritas apenas a aplicativos de plataforma;

- Buscável: pesquisas orgânicas respondem por mais da metade de todo tráfego, por isso é fundamental que a PWA possa ser encontrada e garantir que as URLs canônicos existam para o conteúdo e que o site possa ser indexado pelos mecanismos de pesquisa;

- Entradas: usuários não devem ter problemas de uso da aplicação ao alternar entre os tipos de entrada, os métodos de entrada devem ser independentes do tamanho de tela;

- Permissões: solicitações de permissões, tais como notificações, geolocalização e credenciais, devem ser acionadas somente após o fornecimento de justificativa do contexto, para melhorar as chances de o usuário aceitar as solicitações;

- Código: garantir que seu aplicativo esteja sempre atualizado e que o código-fonte mantenha sua integridade é essencial para a implementação eficiente de novos recursos, alinhando-se assim com os outros objetivos delineados nesta lista de verificação.

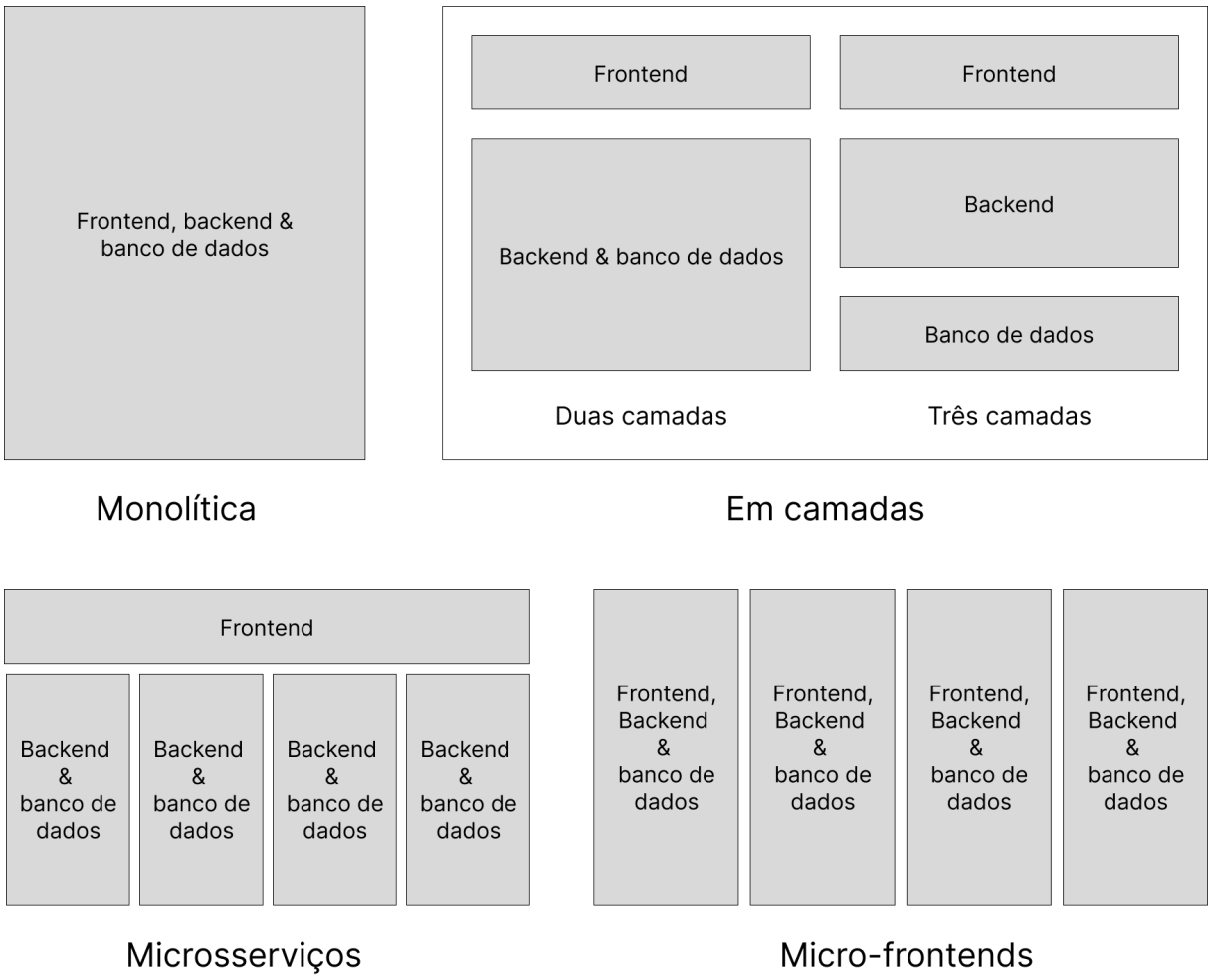
PWAs fornecem um conjunto de recursos baseados em *service workers*. Um *service worker* é um *script* executado no navegador em segundo plano, separado de uma página web, para prover funcionalidades como experiência off-line ou notificações *push* (MEZZALIRA, 2021).

Service works permitem criar nossa estratégia de *cache* para um aplicativo da web, com API nativas disponíveis dentro dos navegadores. Esse padrão é chamado de off-line *first*, ou *cache first*, e é a estratégia mais popular para servir conteúdo ao usuário. Se um recurso estiver em *cache* e disponível off-line, retorne-o primeiro antes de tentar baixá-lo do servidor. Se ainda não estiver no *cache*, baixe-o e armazene-o em cache para uso futuro. É tão simples quanto isso, mas muito poderoso para aprimorar a experiência do usuário em um aplicativo da web, especialmente em dispositivos móveis.

2.2 Principais Arquiteturas de Aplicações *Frontend*

Segundo Richards e Ford (2020), um estilo de arquitetura pode ser definido como "a estrutura abrangente de como a interface do usuário e o código-fonte do backend são organizados (como em camadas de uma implantação monolítica ou serviços implantados separadamente) e como esse código-fonte interage com um armazenamento de dados". Essa definição fundamenta a abordagem desta pesquisa, na qual estilos de arquitetura, de agora em diante referidos simplesmente como arquiteturas, são compreendidos como maneiras distintas de estruturar as unidades implantáveis de uma aplicação. Neste estudo, serão examinadas quatro principais arquiteturas: monolítica, em camadas, microsserviços e micro-frontends. A Figura 2 ilustra a estrutura fundamental desses distintos estilos arquiteturais, enquanto a Tabela 1 apresenta um panorama histórico de sua adoção, bem como uma descrição de suas características e das motivações que impulsionaram sua introdução.

Figura 2 – Estilos de arquiteturas *web*.



Fonte: Adaptado de Geers (2019).

Tabela 1 – Resumo dos estilos de arquitetura *web*

Arquitetura	**Adoção**	**Descrição**	**Motivação**
Monolítica	1990s (DOYLE; LOPES, 2008)	Todas as funcionalidades estão em uma única unidade de implantação (WOLFF, 2016).	Requer o mínimo de infraestrutura e experiência para implantação e manutenção (WOLFF, 2016).
Em camadas	1990s (GALLAUGHER; RAMANATHAN, 1996)	A funcionalidade é dividida entre 2-n níveis, geralmente 2-3 (GALLAUGHER; RAMANATHAN, 1996).	Dividir a aplicação em partes mais gerenciáveis, por exemplo, separando o código de apresentação da lógica. Facilidade de desenvolvimento e melhor escalabilidade (GALLAUGHER; RAMANATHAN, 1996).
Microsserviços	2010s (MAZZARA; MEYER et al., 2017)	A funcionalidade de *backend* é dividida entre vários serviços (MAZZARA; MEYER et al., 2017).	Dividir o aplicativo em partes mais gerenciáveis. Permitir que diferentes equipes trabalhem em diferentes serviços de forma independente. Permitir que os serviços escalem de forma independente (MAZZARA; MEYER et al., 2017).
Micro-frontend	2020s (PAVLENKO et al., 2020)	Toda a funcionalidade é dividida entre vários serviços (PAVLENKO et al., 2020).	Dividir o *frontend* em partes mais gerenciáveis e permitir que diferentes equipes trabalhem nos *frontends* de forma independente (PAVLENKO et al., 2020).

2.2.1 Monolítica

A arquitetura monolítica agrupa todos os componentes da aplicação em um único pacote, implantado em um servidor web. O crescimento da aplicação aumenta a complexidade da base de código, tornando a manutenção e escalabilidade mais desafiadoras e dispendiosas. Alterações pequenas exigem testes e reimplantação de toda a aplicação, aumentando o tempo e recursos necessários. Para lidar com cargas maiores, é necessário escalar todo o sistema, mesmo que somente uma parte precise de aumento de capacidade (ABDULLAH; IQBAL; ERRADI, 2019).

Esta arquiteturas unifica componentes como a interface do usuário, a lógica computacional e o acesso a dados em um único artefato executável. Essa abordagem, embora prática inicialmente, pode se tornar complexa e custosa à medida que a aplicação cresce, dificultando sua manutenção e escalabilidade.

Em suma, uma aplicação estruturada segundo o estilo de arquitetura monolítica, caracteriza-se por uma única unidade de implantação que abrange todo o código-fonte (RICHARDS; FORD, 2020). Embora um monólito possa ser composto internamente por

diversos módulos, sua implantação ocorre integralmente, sem possibilidade de separação (WOLFF, 2016). Em algumas definições, monólitos são descritos como aplicações que possuem um único executável (VELEPUCHA; FLORES, 2023); no entanto, esta pesquisa adotará a concepção segundo a qual a característica definidora do monólito é sua única implantação. Esse aspecto implica que os módulos internos não podem ser executados de maneira independente, exigindo que a aplicação opere na totalidade coeso.

2.2.2 Em Camadas

Uma aplicação que adota a arquitetura em camadas é caracterizada pela divisão de suas unidades funcionais em componentes distintos, que operam de forma independente e podem ser implantados separadamente (RICHARDS; FORD, 2020). Geralmente, a comunicação entre essas camadas ocorre exclusivamente entre camadas adjacentes (RICHARDS; FORD, 2020). As configurações mais frequentemente utilizadas incluem arquiteturas de duas e três camadas, embora, em teoria, não exista um limite para a quantidade de camadas que uma aplicação pode possuir.

No contexto das aplicações web, a arquitetura de duas camadas é comumente estruturada em um *frontend* e um servidor de banco de dados (GALLAUGHER; RAMANATHAN, 1996; SHAN; HUA, 2006), de modo que a interação entre ambos ocorre de maneira direta (GALLAUGHER; RAMANATHAN, 1996).

Por outro lado, a arquitetura de três camadas introduz um elemento adicional—o backend—posicionado entre o *frontend* e o banco de dados (GALLAUGHER; RAMANATHAN, 1996; RICHARDS; FORD, 2020). Esse modelo promove uma divisão mais clara de responsabilidades, atribuindo ao *backend* a lógica de processamento de dados, ao *frontend* a interface de apresentação e ao banco de dados a função de armazenamento (GALLAUGHER; RAMANATHAN, 1996). Como consequência, o *frontend* não mantém comunicação direta com o banco de dados, mas sim por intermédio do *backend*.

Já na arquitetura de quatro camadas, a estruturação inclui uma camada suplementar, que pode desempenhar funções como *firewall* ou um servidor web adicional (IBM, January 2025). Além disso, há exemplos de arquiteturas com seis camadas (SHAN; HUA, 2006), em que a lógica do sistema é distribuída de maneira ainda mais fragmentada, permitindo maior controle sobre os diferentes aspectos do processamento e da comunicação entre componentes.

2.2.3 Microsserviços

Um microsserviço é definido como uma unidade autônoma e coesa que pode ser implantada de maneira independente, interagindo com outras unidades por meio de mensagens (MAZZARA; MEYER et al., 2017). Uma aplicação é considerada aderente

à arquitetura de microsserviços quando sua composição é inteiramente baseada em tais unidades independentes (MAZZARA; MEYER et al., 2017). No entanto, a implementação tradicional desse estilo arquitetônico geralmente se limita à segmentação da funcionalidade do lado do servidor, mantendo o *frontend* estruturado como uma única unidade monolítica (PAVLENKO et al., 2020). Quando o *frontend* também é particionado em múltiplas unidades autônomas, a arquitetura resultante é denominada arquitetura de micro-frontends (PAVLENKO et al., 2020).

2.2.4 Micro-frontend

Trata-se de uma arquitetura emergente que se inspira na arquitetura de *backend* de microsserviços. Micro-frontends tem como conceito principal a divisão de uma base monolítica em partes menores, permitindo a distribuição dos trabalhos entre equipes autônomas, sejam elas co-localizadas ou distribuídas, sem reduzir a velocidade de entrega. Utilizar microsserviços simplifica a lógica de negócios, entretanto é necessário lidar com a complexidade intrínseca em níveis distintos, tais como: rede, camada de persistência, protocolos de comunicação, segurança e muitas outras. Caso haja uma redução drástica da lógica de negócios e na complexidade do código, a sobrecarga na automação, governança, observação e comunicação precisarão ser consideradas.

Da mesma forma que outras arquiteturas, micro-frontends podem não ser adequados para todos os projetos e arquiteturas já existentes, como renderização do lado servidor ou Jamstack, continuam sendo opções válidas. Todavia, micro-frontends podem fornecer uma maneira de estruturar aplicações *frontend* em escala, resolvendo alguns dos principais desafios de escalabilidade encontrados por empresas no passado, tanto de uma perspectiva técnica quanto organizacional.

Micro-frontends, combinados com microsserviços e uma forte cultura de engenharia onde todos são responsáveis por seu próprio domínio, podem ajudar a alcançar agilidade organizacional e melhor tempo de comercialização. Esta arquitetura pode ser usada em combinação com outra arquitetura de *backend*, como um *backend* monolítico ou Arquitetura Orientada a Serviços (SOA). No entanto, micro-frontends são adequados quando se pode ter uma arquitetura de microsserviços, permitindo definir fatias de um aplicativo que estão evoluindo juntas (MEZZALIRA, 2021). A arquitetura escolhida para este estudo foi de micro-frontend, motivada, principalmente, pela autonomia das unidades desenvolvidas.

2.3 *Tecnologias utilizadas*

A aplicação desenvolvida para estudo de caso seguiu o padrão de desenvolvimento adotado pela Embrapa Gado de Corte. Dessa forma, foram utilizadas as *frameworks* e tecnologias já utilizadas e validadas em outros projetos, mas com uma arquitetura

de micro-frontends integrada com recursos de PWA. A seguir serão apresentadas estas ferramentas.

2.3.1 Federação de Módulos

A Federação de Módulos é a principal tecnologia usada para implementar a arquitetura de micro-frontends. Essa é uma tecnologia de ponta que revoluciona o compartilhamento de código e o gerenciamento de recursos no desenvolvimento *web*. Com ela é possível integrar perfeitamente vários aplicativos JavaScript, ou micro-frontends, compartilhando código e recursos dinamicamente em tempo de execução. Esse conceito de módulos desacoplados e independentes aprimora a modularidade, a escalabilidade e a manutenibilidade em projetos *web* complexos.

A arquitetura de um projeto que implementa o conceito de Federação de Módulos normalmente gira em torno de um aplicativo principal, servindo como aplicativo *host* e contêiner, usado para agregar um conjunto de micro-aplicativos remotos independentes com base em requisitos. Cada aplicativo remoto pode compartilhar vários módulos que podem incluir componentes, elementos personalizados, componentes de biblioteca ou funções de API. O conceito de Federação de Módulos foi introduzido com o empacotador Webpack e, recentemente, também a ferramenta de construção Vite começou a oferecer a possibilidade de incorporar esse conceito arquitetônico.

2.3.2 Vite

As informações apresentadas neste tópico foram retiradas da documentação do Vite (2024). Antes da introdução dos módulos *EcmaScript* (ES) em navegadores, os desenvolvedores não tinham um método nativo para escrever JavaScript em um formato modular. Consequentemente, o conceito de "agrupamento"(*"bundling"*) surgiu, necessitando de ferramentas para rastrear, processar e concatenar módulos de origem em arquivos executáveis no navegador. Com o tempo, avanços como Webpack, Rollup e Parcel melhoraram significativamente a experiência de desenvolvimento de *frontend*.

No entanto, à medida que os aplicativos se tornaram mais complexos, o grande volume de JavaScript envolvido aumentou drasticamente. Projetos de grande escala geralmente consistem em milhares de módulos, levando a problemas de desempenho com ferramentas baseadas em JavaScript, onde iniciar um servidor de desenvolvimento pode levar um tempo impraticável, às vezes minutos. Mesmo com a *Hot Module Replacement* (HMR), as edições de arquivo podem levar segundos para refletir no navegador, prejudicando a produtividade e a satisfação no trabalho dos desenvolvedores.

O Vite aborda esses desafios utilizando avanços recentes do ecossistema, como o surgimento de módulos ES nativos em navegadores e ferramentas JavaScript desenvolvidas

em linguagens de compilação para nativas.

Em configurações tradicionais de *build* baseadas em *bundler*, iniciar um servidor de desenvolvimento exige um rastreamento rápido e a construção de todo o aplicativo. O Vite aprimora os tempos de inicialização do servidor, categorizando módulos do aplicativo em dependências e código-fonte.

Dependências, geralmente grandes e raramente alteradas, são pré-agrupadas usando *esbuild*, que, sendo escrito em linguagem de programação Go, pré-agrupa dependências significativamente mais rápido do que *bundlers* baseados em JavaScript. O código-fonte, normalmente sujeito a edições e transformações frequentes, é servido por *EcmaScript Module* (ESM) nativo. Isso permite que o navegador manipule parte do processo de empacotamento, permitindo que o Vite transforme e sirva o código-fonte sob demanda.

Esse método garante que o código por trás das importações dinâmicas condicionais seja processado somente se necessário pela exibição atual.

Em configurações baseadas em *bundler*, editar um arquivo exige a reconstrução de todo o pacote, fazendo com que as velocidades de atualização diminuam conforme os aplicativos crescem.

Alguns empacotadores tentam mitigar isso executando o empacotamento na memória, invalidando apenas partes do gráfico do módulo em alterações, mas eles ainda precisam reconstruir o pacote inteiro e recarregar a página, interrompendo o estado do aplicativo. Embora o HMR ofereça algum alívio ao permitir que os módulos sejam atualizados sem afetar o restante da página, sua velocidade também se deteriora com aplicativos maiores.

O Vite executa o HMR sobre o ESM nativo, invalidando precisamente a cadeia entre o módulo editado e seu limite HMR mais próximo, garantindo atualizações consistentemente rápidas.

O Vite também emprega cabeçalhos HTTP para agilizar recarregamentos de página inteira, tornando as solicitações do módulo de código-fonte condicionais e armazenando em *cache* as solicitações do módulo de dependência, minimizando os acessos ao servidor uma vez armazenados em *cache*. Experimentar a velocidade do Vite pode fazer com que os desenvolvedores relutem em retornar ao desenvolvimento empacotado tradicional.

Apesar do amplo suporte ao ESM nativo, o envio do ESM desagregado na produção continua ineficiente devido a viagens de ida e volta de rede adicionais de importações aninhadas, mesmo com HTTP/2. Para desempenho de carregamento de produção ideal, o agrupamento com *tree-shaking*, *lazy-loading* e divisão de pedaços comuns continua sendo preferível.

Garantir a consistência entre o comportamento do servidor de desenvolvimento e as compilações de produção é desafiador; portanto, o Vite oferece um comando de compilação

pré-configurado com otimizações de desempenho integradas.

2.3.3 Vue.js

Segundo a documentação oficial, adotada com fonte deste tópico, o Vue.js (2024) é um *framework* JavaScript projetado para criar interfaces de usuário. Ele estende HTML, CSS e JavaScript padrão, oferecendo um paradigma de programação declarativo e centrado em componentes que facilitam o desenvolvimento eficiente de interfaces de usuário, independentemente de sua complexidade.

Os atributos fundamentais do Vue abrangem:

- Renderização declarativa: aprimora o HTML convencional ao incorporar uma sintaxe de modelo, permitindo a especificação declarativa da saída HTML dependente do estado do JavaScript;
- Reatividade: monitora intrinsecamente as modificações dentro do estado do JavaScript e atualiza o DOM com eficiência em resposta a essas alterações.

Vue é uma estrutura e ecossistema que compreende a maioria das funcionalidades comuns imperativas ao desenvolvimento *front-end*. No entanto, a *web* é imensamente heterogênea - os projetos desenvolvidos para *web* podem variar significativamente em formato e escala.

À vista disso, essa *framework* foi projetada para ser flexível e ser adotada de forma incremental. Dependendo do caso de uso, Vue pode ser empregado de maneiras distintas:

- Otimizar HTML estático sem a etapa de compilação;
- Incorporar como Componentes *web* em qualquer página;
- Criar SPA;
- Renderizar no lado servidor (SSR);
- Gerar aplicações estáticas (SSG);
- Desenvolver aplicações nativas para *desktop* móvel, WebGL e para terminal.

O desenvolvimento de componentes em Vue utiliza um formato de arquivo similar ao HTML, denominado Componente de Arquivo Único, reunindo a lógica do componente (JavaScript), o modelo de marcação (HTML) e os estilos (CSS) em um único arquivo com extensão *.vue. Além disso, os componentes podem ser criados utilizando dois estilos distintos, API de Opções e API de Composição.

A API de Opções está fundamentada no conceito de uma "instância de componente", o que geralmente se alinha mais adequadamente com um modelo mental baseado em classes para usuários acostumados com linguagens de programação orientadas a objetos. Essa abordagem também é mais acessível para iniciantes, pois abstrai muitos detalhes da reatividade e organiza o código por meio de grupos de opções.

Por outro lado, a API de Composição foca na declaração direta de variáveis de estado reativo no escopo da função, permitindo a composição do estado a partir de múltiplas funções para gerenciar a complexidade. Esta abordagem é mais flexível e requer um entendimento sólido de como a reatividade opera no Vue para ser utilizada de forma eficaz. Em contrapartida, sua flexibilidade oferece padrões mais robustos para a organização e reutilização da lógica.

Ambos os estilos de API são totalmente aptos a atender casos de uso comuns. Trata-se de interfaces distintas suportadas pelo mesmo sistema subjacente. A API de Opções é construída sobre a API de Composição e os conceitos fundamentais e o conhecimento sobre o Vue são compartilhados por ambos os estilos. Este projeto adota o estilo de API de Composição como padrão de desenvolvimento.

2.3.4 Vuetify

Desde o lançamento em 2014, o Vue.js se destacou e figura entre os mais populares *frameworks* JavaScript adotados mundialmente. Em parte, essa popularidade se dá devido ao amplo uso de componentes que possibilitam o desenvolvimento de módulos pequenos que podem ser usados e reutilizados em uma aplicação.

Segundo a documentação, fonte de informações para este tópico, o Vuetify (2024) é uma coleção de componentes pré-fabricados, integrados com recursos robustos, como temas dinâmicos, padrões globais, layouts de aplicativos e muitos outros. Seu objetivo principal é equipar os desenvolvedores com ferramentas abrangentes essenciais para construir experiências de usuário ricas e cativantes.

Vuetify é um robusto *framework* de componentes Vue, desenvolvido do zero para ser de fácil aprendizagem. Com uma coleção de componentes de interface do usuário que mantém um estilo uniforme para toda a aplicação, com opções de personalização para atender a diversos cenários de uso.

O Vuetify é um projeto de código aberto disponível gratuitamente sob a licença do MIT. O código-fonte está disponível no GitHub, possibilitando que desenvolvedores modifiquem e contribuam para seu desenvolvimento.

Os componentes em Vuetify são desenvolvidos seguindo a especificação do Material Design do Google, oferecendo uma vasta gama de opções de personalização que se ajustam a qualquer estilo ou design, mesmo que não sigam o padrão Material. Além disso, possui

um extenso ecossistema de ferramentas auxiliares que aprimoram a experiência de desenvolvimento, abrangendo desde a criação de projetos até o design de conjuntos de interface do usuário.

2.4 *Plugins* relevantes

Esta seção destaca os principais *plugins* empregados neste projeto. Embora módulos externos tenham sido utilizados durante a implementação do caso de uso e dos modelos padronizados, estes foram fundamentais para criar a federação de módulos e para a concepção de PWA.

2.4.1 Vite PWA

As informações trazidas neste tópico são parte da documentação do PWA (2024), a qual apresenta que se trata de um *plugin* (`vite-plugin-pwa`) utilizado para converter aplicações *web* convencionais em PWAs com pouca configuração. Ele vem predefinido com configurações padrão integradas para casos de uso comuns.

Este *plugin* pode:

- Gerar o manifesto da aplicação *web* (*web App Manifest*) e adicioná-lo ao seu ponto de entrada (normalmente `index.html`);
- Gerar o *service worker*;
- Gerar um `script` para registrar o *service worker* no navegador.

Maiores detalhes sobre o *plugin*, inclusive relacionados à instalação, configurações e forma de uso, podem ser consultados em sua documentação oficial.

2.4.2 Vite Plugin Federation

Embora a Federação de Módulos não seja suportada nativamente no Vite, há *plugins* de terceiros disponíveis que permitem seu uso. Em particular, o `vite-plugin-federation` da OriginJs permite que projetos Vite utilizem o Federação de Módulos, permitindo o carregamento dinâmico de módulos remotos e o compartilhamento de dependências entre aplicativos implantados de forma independente (ORIGIN.JS, 2022).

A configuração do *plugin* é semelhante ao Webpack e é realizada no arquivo `vite.config.js`. Neste caso, é necessário instalar manualmente o *plugin* dentro de cada aplicativo do projeto, usando um gerenciador de pacotes como o `npm`, `yarn` ou outras ferramentas; então ele pode ser importado como `@originjs/vite-plugin-federation` (ORIGIN.JS, 2022).

2.5 GraphQL

Graph Query Language (GraphQL) é uma linguagem robusta de acesso a dados baseada na *web*, desenvolvida em 2012 e lançada como *software* de código aberto pela Meta/Facebook em 2015. Ela resolve algumas limitações das APIs RESTful, como a habilidade de especificar exatamente quais dados obter de um grafo de dados interligados com uma única consulta. Diversas empresas, incluindo Facebook, GitHub, Atlassian e Coursera, começaram a oferecer APIs na *web* utilizando GraphQL. GraphQL atua como uma linguagem de consulta em tempo de execução no lado do servidor para APIs. Ao lidar com um conjunto de dados estruturados em um gráfico de nós conectados, GraphQL permite consultas a esse gráfico, especificando para cada nó os campos e conexões a serem recuperados (e de forma recursiva em cada nó conectado recuperado) (BELHADI; ZHANG; ARCURI, 2024).

Um serviço *web* GraphQL geralmente está operando em um soquete TCP, aguardando consultas GraphQL como parte de solicitações HTTP em um *endpoint* específico (geralmente /graphql). Contudo, GraphQL não é limitado ao protocolo HTTP, e as consultas podem ser enviadas por outros métodos de comunicação. Uma vantagem significativa é que um usuário pode requisitar todos os dados necessários (e apenas esses) em uma única chamada HTTP.

Um conceito fundamental para linguagem é o esquema, que consiste em uma coleção de tipos e as relações entre eles. Ele descreve quais tipos e campos nesses tipos um cliente pode solicitar. GraphQL é fortemente tipada e possui uma linguagem própria para a definição do esquema. Ao responder a uma consulta, uma API GraphQL devolve um objeto *JavaScript Object Notation* (JSON) composto por dois campos: dados, que contém o resultado da consulta; e erros, caso haja algum erro na consulta (nesse cenário, o campo dados não estará presente). Ressalta-se que o GraphQL não diferencia entre erros do usuário (como uma consulta mal formatada ou uma entrada que não atende a uma restrição lógica de negócios) e erros do servidor (como uma falha interna na lógica de negócios da API, por exemplo, uma exceção de ponteiro nulo). Contudo, pode haver suporte para isso no futuro (BELHADI; ZHANG; ARCURI, 2024).

Ademais, Belhadi, Zhang e Arcuri (2024) afirmam que o GraphQL opera de forma independente do HTTP, uma consulta com erros ainda pode resultar em uma resposta HTTP 200 (indicando que está tudo 'OK'). Este é um comportamento comum em diversas implementações de estruturas GraphQL. Outra limitação do GraphQL é a ausência, até o momento, de um método padronizado para expressar restrições nos campos do gráfico (como um inteiro em um intervalo numérico específico ou uma *string* que precisa satisfazer uma expressão regular específica). Tais restrições poderiam ser incorporadas através de *"directives"*, que são decoradores empregados para ampliar a semântica do esquema. No entanto, essas seriam customizadas e exclusivas para cada API implementada de maneira

distinta.

2.5.1 Apollo Client

É uma biblioteca abrangente de gerenciamento de estado para JavaScript. Ela permite o gerenciamento de dados locais e remotos com GraphQL. É usada para buscar, armazenar em cache e modificar dados de aplicativos, tudo isso enquanto atualiza automaticamente a interface de usuário (APOLLO, 2024).

A documentação do Apollo (2024) diz que essa biblioteca auxilia a estruturar o código de forma econômica, previsível e declarativa, consistente com as práticas de desenvolvimento modernas. Alguns dos principais recursos incluem:

- Busca de dados declarativa: escrever uma consulta e receber dados sem rastrear manualmente os estados de carregamento;
- Cache de solicitação e resposta normalizado: Aumenta o desempenho respondendo quase imediatamente a consultas com dados armazenados em cache;
- Adotável incrementalmente: é possível inserir em qualquer aplicativo JavaScript e incorporar recurso por recurso;
- Universalmente compatível: usar com qualquer configuração de compilação e qualquer API GraphQL.

2.5.2 Considerações Finais

Este capítulo apresentou os principais conceitos acerca das tecnologias, ferramentas e bibliotecas utilizados no desenvolvimento do projeto. Parte das informações trazidas aqui são referências à documentação oficial.

Algumas das tecnologias (Vue.js, Vuetify e PWA) conceituadas neste capítulo são adotadas por padrão pelo departamento de TI da Embrapa Gado de Corte. Para a elaboração do estudo de caso, utilizou-se as *frameworks* Vue.js e Vuetify e a adoção dos recursos de PWA para permitir melhor interação em dispositivos móveis, viabilizado pelo Vite PWA.

A arquitetura de micro-frontends é adequada à proposta de desenvolvimento de aplicações modulares (PAVLENKO et al., 2020). Para isso, houve a necessidade da instalação e uso do Vite Plugin Federation. Já o uso do GraphQL e do Apollo Client foi em virtude da arquitetura da plataforma, pois ela utiliza esse meio de consumo de dados.

3 Revisão da Literatura

Este capítulo apresenta uma revisão buscando na literatura trabalhos que utilizem tecnologia de federação de módulos por meio da arquitetura de micro-frontends, Aplicativos Web Progressivos (PWA), Internet das Coisas (IoT) no setor de agropecuária de precisão para fundamentar o enfoque implementado neste trabalho.

A seção 3.1 apresenta as principais características de trabalhos que se relacionam com esta pesquisa. Na seção 3.2 é feito um comparativo entre os artigos analisados e essa proposta. Por último, na seção 3.3 são apresentadas as considerações finais acerca do capítulo.

3.1 Trabalhos Relacionados

Foi realizada uma pesquisa por trabalhos que utilizaram arquitetura de micro-frontends (MFes) aplicados a PWA, com coleta de dados por meio de IoT empregadas no setor de agropecuária de precisão que fundamentassem o enfoque deste trabalho. A Tabela 2 mostra o resultado das pesquisas realizadas.

Mena et al. (2019) propuseram o uso de uma arquitetura de *software* baseada em microsserviços e micro-frontend para auxiliar na aquisição amigável e contínua de dados geoespaciais e informações utilizando IoT. Esta solução orquestra os microsserviços eu uma PWA baseada em componentes. O microsserviço principal lida com a criação de configurações de componentes usando um gráfico de seleção que consiste em *tags* de componentes e outras propriedades descritivas e também informações contextuais sobre o usuário do aplicativo. Os dados são exibidos em tempo real em dispositivos em IoT com interface formada por componentes. Para alcançar mais usuários, foi utilizado PWAs, o que permitiu romper a barreira tecnológica de compatibilidade de aplicação *frontend* com dispositivos móveis, somente ajustando o aplicativo *web* e tornando-o híbrido. A abordagem de micro-frontend permitiu construir a interface do usuário dinamicamente e desenvolver componentes visuais de forma independente, encontrando diferentes maneiras de mostrar os dados do usuário.

Por outro lado, Simões et al. (2024), propuseram a criação da conectividade perfeita entre Digital Twins (DTs)[1] e o desenvolvimento de camadas de apresentação personalizadas, com a decomposição de sistemas complexos em Digital Twins (DTs) modulares baseados na *web*. Ademais, a estrutura aborda os desafios impostos por elementos

[1] Digital Twins (DTs) fornecem réplicas virtuais de plantas industriais do mundo real, oferecendo recursos para simulação, controle e análise.

Tabela 2 – Trabalhos relacionados que utilizam micro-frontends

Artigo	Área de Estudo	MFes	PWA	IoT
Progressive Web Application Based on Micro-services Combining Geospatial Data and the Internet of Things (MENA et al., 2019)	Geoespacial	✓	✓	✓
Implementing Digital Twins via micro-frontends, micro-services, and web 3D (SIMÕES et al., 2024)	Plantas Industriais	✓	✓	✓
(KAUSHIK; KUMAR; RAJ, 2024)	*Framework*	✓	-	-
Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review (PELTONEN; MEZZALIRA; TAIBI, 2021)	Revisão de literatura	✓	-	-
Micro-Frontends: Principles, Implementations, and Pitfalls (TAIBI; MEZZALIRA, 2022)	*Streaming* de esportes	✓	-	-
Towards a Modular Architecture for Industrial HMIs (SHAKIL; ZOITL, 2020)	Interface homem-máquina	✓	-	-
A Novel Application of Educational Management Information System based on Micro Frontends (WANG et al., 2020)	Gestão educacional	✓	-	-
Microservice-based architecture for engineering tools enabling a collaborative multi-user configuration of robot-based automation solutions (SCHäFFER et al., 2019)	Automação	✓	-	-

de vários fornecedores dentro de DTs e facilita a orquestração de serviços e ambientes virtuais para integração e operação perfeitas, permitindo a troca de dados em tempo real entre modelos virtuais e máquinas físicas. O intuito do trabalho é revolucionar a criação e o gerenciamento de DT, introduzindo uma estrutura escalável e adaptável projetada para habilitar o *web* 3D para a indústria e a manufatura. Ao integrar tecnologias de micro-frontend de ponta, essa abordagem buscou melhorar a manutenibilidade, eficiência e produtividade. Com a incorporação de micro-frontends na arquitetura de *frontend*, é possível obter maior escalabilidade nos processos de desenvolvimento e a integração perfeita de sistemas complexos, revolucionando a fragmentação de aplicativos corporativos. No entanto, potenciais limitações, como duplicação de código e preocupações com carga útil de aplicativo, foram reconhecidas. O uso de PWA atenuou algumas desvantagens associadas às SPAs e micro-frontends, como os tempos de carregamento mais longos em virtude do carregamento de todo o aplicativo no navegador. No caso de micro-frontends, as PWAs também reduzem o uso de código redundante. Aproveitar a IoT e os serviços de dados, garantiu uma representação de DT atualizada, reduzindo a duplicação de serviços e esforços, ao mesmo tempo, em que facilitamos uma abordagem mais eficiente e flexível ao desenvolvimento de DT por meio da criação de um portfólio de recursos reutilizáveis. No

geral, a estrutura proposta oferece soluções promissoras para muitos desafios em ambientes industriais, abrindo caminho para uma implantação e gerenciamento de DT mais eficientes e eficazes.

Já em (KAUSHIK; KUMAR; RAJ, 2024), um *framework* para desenvolvimento de aplicações *web* usando micro-frontends na interface e microsserviços no *backend*, visando melhorar o desempenho foi desenvolvido. Um estudo empírico compara esse *framework* com abordagens tradicionais, mostrando melhorias no tempo de resposta e na taxa de transferência. Adicionalmente, o trabalho apresenta um modelo de aprendizado de máquina (*Machine Learning*) para prever o tempo de resposta das aplicações baseadas em microsserviços. Os resultados indicam que a combinação de micro-frontends e microsserviços oferece vantagens significativas em termos de desempenho. Ameaças à validade do estudo, como o tamanho limitado da aplicação de caso, são discutidas.

Em contrapartida, (PELTONEN; MEZZALIRA; TAIBI, 2021), analisam a adoção de micro-frontends em desenvolvimento web, examinando as motivações por trás dessa escolha, os benefícios percebidos e os desafios encontrados. Por meio de uma revisão da literatura acadêmica e não acadêmica, o estudo identifica a complexidade crescente de monolíticos e a necessidade de escalabilidade como principais impulsionadores da adoção de micro-frontends. Os benefícios incluem maior independência de equipes, suporte a diferentes tecnologias e implantações autônomas. Porém, o estudo também destaca problemas como aumento do tamanho do *payload*, duplicação de código e complexidade na manutenção e monitoramento. A pesquisa conclui que, apesar dos benefícios, micro-frontends exigem planejamento cuidadoso e não são a solução ideal para todos os cenários.

Em (TAIBI; MEZZALIRA, 2022) a arquitetura de micro-frontends é debatida sob uma abordagem para decompor aplicações *web* em micro-aplicações semi-independentes. É apresentado os princípios, implementações (como *application shell*, *iFrames*, *web Components*, *module federation*, *Edge Side Includes* e composição do lado do servidor) e desvantagens dessa arquitetura, baseando-se na experiência dos autores com o DAZN. O artigo também identifica áreas de pesquisa futuras e oferece recomendações para o desenvolvimento bem-sucedido de micro-frontends, incluindo a importância de automação e governança. Finalmente, ele destaca lições aprendidas com a implementação desta arquitetura.

Sob outro enfoque, Shakil e Zoitl (2020) exploram o desenvolvimento de uma arquitetura modular para interfaces homem-máquina (IHMs) para sistemas industriais, utilizando uma abordagem de micro-frontends. A arquitetura proposta visa facilitar a integração ágil e modular de componentes distribuídos em sistemas de produção modulares e adaptáveis, permitindo a atualização dinâmica da IHM com o acréscimo de novos módulos. O trabalho descreve os requisitos para uma IHM modular, detalha uma implementação baseada em micro-frontends com Vue.js, e discute desafios e pontos a serem investigados

em trabalhos futuros, como a descoberta automática de módulos e a comunicação flexível entre eles. A pesquisa se baseia em conceitos de microsserviços e arquiteturas orientadas a serviços, buscando maior flexibilidade e escalabilidade em sistemas ciber-físicos de produção.

De outra forma, em (WANG et al., 2020) é descrito o desenvolvimento de um sistema de informação de gestão educacional para a Universidade Normal do Leste da China, baseado na arquitetura de micro-frontends. A abordagem visa solucionar os problemas de sistemas tradicionais, como alta complexidade e acoplamento entre módulos, por meio da decomposição em aplicações independentes e facilmente escaláveis. A metodologia utiliza o Design Orientado a Domínio (DDD) para dividir o sistema em contextos delimitados, e tecnologias como *web Components* e *iFrames* para a implementação. O resultado é um sistema mais ágil, com desenvolvimento incremental e melhor experiência do usuário. Finalmente, o estudo compara sua solução com abordagens existentes e destaca suas contribuições.

E por fim, Schäffer et al. (2019) descrevem o desenvolvimento de um configurador baseado em microsserviços para soluções de automação robótica, permitindo a configuração colaborativa multiusuário. A arquitetura de microsserviços, combinada com micro-frontends, melhora a escalabilidade, manutenibilidade e a colaboração em tempo real entre equipes de desenvolvimento. O protótipo implementado utiliza tecnologias modernas como Docker, Node.js e Socket.IO para alcançar essa funcionalidade multiusuário. O artigo destaca as vantagens dessa abordagem em comparação com ferramentas monolíticas tradicionais, focando na redução de custos e aumento da eficiência no desenvolvimento de software para automação industrial. Finalmente, são apresentadas propostas para futuras otimizações e extensões do sistema.

3.2 Comparação com os trabalhos relacionados

O artigo escrito por Mena et al. (2019), é o que mais se assemelha, em termos de tecnologias utilizadas, com a proposta deste trabalho. Em ambos, o foco é no uso da arquitetura de micro-frontend para montar uma PWA e usa informações coletadas por IoT, tendo como principal diferença a área de atuação. Enquanto o artigo é voltado para dados geoespaciais, esta dissertação é voltada para a pecuária de precisão, com informações sobre os animais e o ambiente em que estão inseridos.

Da mesma forma, o artigo de Simões et al. (2024) contém similaridades no uso de tecnologias. Mais uma vez, os dois utilizam a arquitetura de micro-frontend para criar uma PWA e coletam dados usando sensores usando IoT, entretanto, a área de atuação também é distinta, uma vez que o artigo propõe o desenvolvimento de projeto para uma DT.

Nos demais trabalhos (KAUSHIK; KUMAR; RAJ, 2024; PELTONEN; MEZZA-

LIRA; TAIBI, 2021; TAIBI; MEZZALIRA, 2022; SHAKIL; ZOITL, 2020; WANG et al., 2020; SCHäFFER et al., 2019), a única semelhança com este trabalho é o uso da arquitetura de micro-frontend. Pois, um anuncia um *framework* para desenvolvimento de aplicações *web* usando micro-frontends na interface e microsserviços no *backend*; o outro é uma revisão da literatura acadêmica e não acadêmica que analisa a adoção de micro-frontends em desenvolvimento web; no próximo é discutido a arquitetura de micro-frontends, sob uma abordagem para decompor aplicações *web* em micro-aplicações semi-independentes, abordando os princípios, implementações, desvantagens da arquitetura de micro-frontend com base na experiência dos autores com o desenvolvimento da DAZN; posteriormente, mostra uma abordagem de micro-frontend no desenvolvimento de uma arquitetura modular para interfaces homem-máquina (IHMs) para sistemas industriais; é subsequentemente descrito o desenvolvimento de um sistema de gestão educacional utilizando a arquitetura de micro-frontends; e, por último, é descrito o desenvolvimento de um configurador baseado em microsserviços para soluções de automação robótica, permitindo a configuração colaborativa multiusuário, combinada com micro-frontends.

3.3 Considerações Finais

Embora tenham sido encontrados dois trabalhos que utilizam tecnologias similares às aplicadas no desenvolvimento desse projeto, como a combinação da arquitetura de micro-frontends com PWA e coleta de dados via IoTs, as áreas de aplicação são distintas.

O restante dos artigos analisados se assemelha somente pelo foco no uso da arquitetura de micro-frontends, diferenciando-se no restante das tecnologias e também na área de atuação.

Não foram encontrados artigos que aplicaram a arquitetura de micro-frontends na área de pecuária ou agricultura. Assim como não foram encontradas propostas que combinassem o uso de *module federation*, PWA e o empacotador Vite.

O diferencial desse trabalho é justamente o uso da associação de uma arquitetura que utiliza tenologias atuais para a criação de módulos que deem suporte visual para uma plataforma voltada para pecuária de precisão que utiliza, entre outras fontes, sensores para coletar dados.

Portanto, a proposta é significativa para a área em que esta dissertação está inserida, principalmente por detalhar as principais configurações utilizadas que permitiram utilizar as tecnologias mais recentes em um contexto pouco explorado e com pouco conteúdo disponível.

4 Plataforma e-Cattle

Neste capítulo é apresentada a plataforma e-Cattle, tese de doutorado de Carromeu (2019), sua arquitetura e seus componentes. Na seção 4.1, é descrita de maneira geral a arquitetura em camadas da plataforma e suas principais definições. A seção 4.2 contextualiza sucintamente o *middleware* BigBoxx. A seguir, na seção 4.3, é explicado como os dados dos sensores foram padronizados e classificados antes de serem armazenados no banco de dados e o funcionamento das aplicações Kernel e Totem UI. Na seção 4.4, é mostrado como se faz para credenciar uma aplicação utilizando a aplicação Totem UI, para que ela possa acessar o barramento GraphQl. A seção 4.5 apresenta resumidamente o componente Query. E, por fim, a seção 4.6 discorre acerca das considerações finais sobre este capítulo.

4.1 Arquitetura

A infraestrutura da Plataforma e-Cattle é fundamentada em IoT, para integrar, monitorar e automatizar serviços para o setor de pecuária de precisão. Com foco em estabelecer diversos projetos desenvolvidos pela parceria entre Embrapa e Universidade Federal de Mato Grosso do Sul (UFMS), em uma única plataforma de baixo custo, com interface de usuário simples e funcional e com implantação em qualquer nível de propriedade rural, de hostis e desconectadas da internet até as mais tecnológicas, com o mínimo esforço (CARROMEU, 2019).

Carromeu (2019) descreve que a arquitetura projetada para a plataforma e-Cattle é composta de seis camadas distintas, como pode ser visto na Figura 3, cada uma com seus componentes específicos, seja de hardware ou software. Essa estrutura busca simplificar o desenvolvimento e manutenção da plataforma, aumentar sua flexibilidade para atualizações futuras e melhorar a interoperabilidade com novas tecnologias (CARROMEU, 2019).

A primeira camada é a de Sensoriamento, nela estão os artefatos de hardware e software embarcado responsáveis pela coleta de dados a campo. Os dados são coletados de maneira mais bruta e em formatos variados. Além disso, novos sensores podem ser incluídos à plataforma desde que o desenvolvedor siga o padrão de autenticação e comunicação definidos.

Na camada de Agregação estão presentes os coordenadores, que são dispositivos auxiliares encarregados pelo envio do dado coletado para a camada seguinte. Dessa forma, ela está conectada a alguma rede ou interface de recebimento de dados da Camada de Coleta de Dados.

As três camadas a seguir são parte do BigBoxx, componente central por implantar

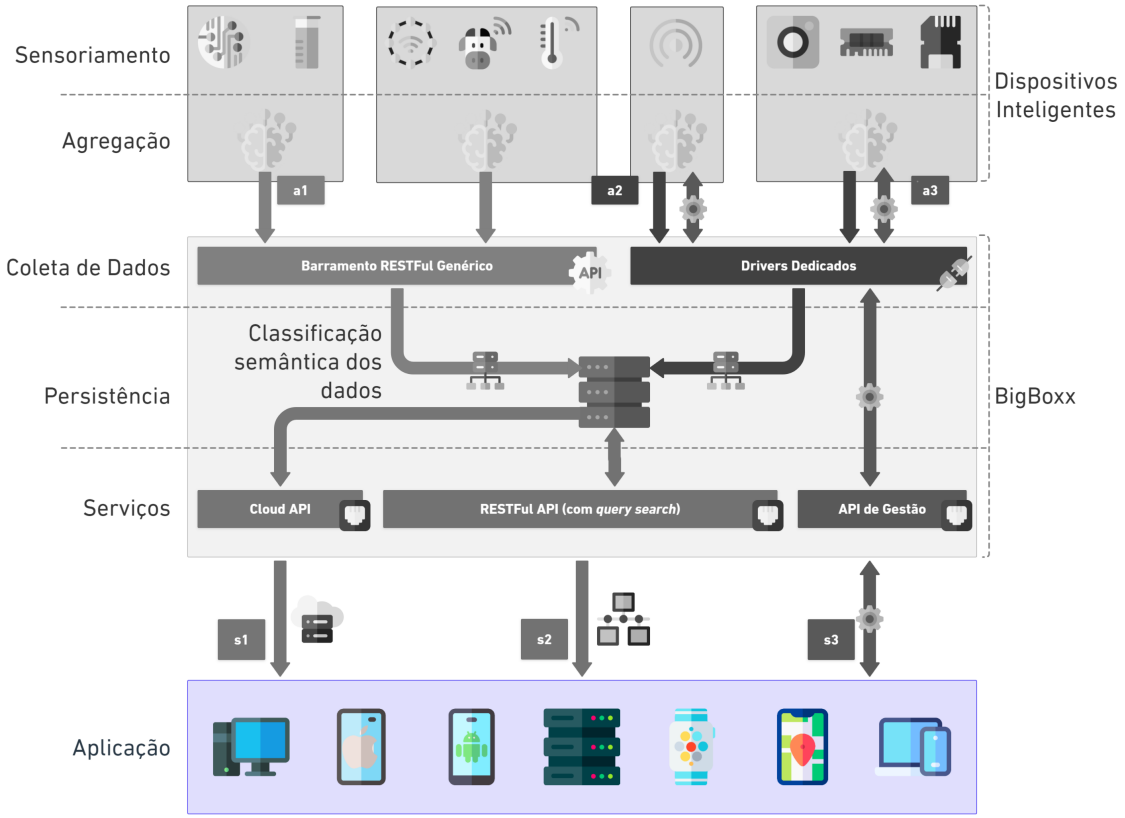


Figura 3 – Arquitetura projetada para a plataforma e-Cattle.

a arquitetura da plataforma e-Cattle. Trata-se de um *middleware* com três componentes de software (Kernel, Service Bus e Totem UI), para mais detalhes é recomendada a leitura de (CARROMEU, 2019).

A camada de Coleta de Dados implementa os barramentos para a interface RESTful genérica e outra interface para os *drivers* de comunicação dedicados. Esta camada está implementada no dispositivo de hardware autônomo (BigBoxx), que está conectado à rede de dados da propriedade rural.

Já a camada de Persistência armazena os dados sensoriais obtidos pela camada anterior em um banco de dados do tipo NoSQL. Esta camada também faz o tratamento e validação dos dados do sensor conforme seus valores semânticos.

Por outro lado, a camada de Serviços é responsável pela comunicação do *middleware* com as aplicações externas. Para isso, são ofertados três serviços do tipo *pull/push* implementados utilizando o protocolo HTTP (padrão RESTful). O primeiro serviço (*s1*) envia dados para a aplicação em nuvem do e-Cattle; o segundo (*s2*) é um barramento de dados com consulta utilizado pelos aplicativos externos para consumir os dados sensoriais; e o terceiro (*s3*) é a interface para gerenciamento dos dispositivos.

Por fim, a camada de Aplicação é onde estão os *softwares* de alto nível. Ela utiliza os dados dos sensores para auxiliar os produtores rurais na geração de indicadores, tomada

de decisão e outras ações relacionadas à inteligência de gestão. Dados de outras fontes podem ser utilizados em associação, tais como o valor da arroba do boi, obtidos por meio de *crawlers*, barramentos de dados de outros, bancos de dados da propriedade ou mesmo informações incluídas pelo usuário.

Um exemplo prático de como a plataforma funciona é o projeto BalPass, balança de passagem, que recebe os dados de pesagem coletados pelas Camadas de Sensoriamento e Agregação. Os dados são sincronizados, classificados semanticamente e armazenados nas Camadas de Coleta de Dados e de Persistência no *middleware* BigBoxx. As aplicações de alto nível, criadas na Camada de Aplicação, recorrem a assinaturas de publicação e subscrição (`Subscription Pub/Sub`) de pesagem para monitorar as informações utilizando a API GraphQL na camada de Serviços e contrastam com a cotação da arroba do dia. A aplicação faz a análise das informações e decide quais animais estão qualificados para a venda.

A plataforma foi delineada para ser robusta e conforme o paradigma de *software* livre, para isso emprega tecnologias atuais de desenvolvimento de *software* embarcado consolidadas no mercado (CARROMEU, 2019).

4.2 BigBoxx

Este *middleware* é responsável pela implementação das camadas de Coleta de Dados, Persistência e Serviços dentro da arquitetura proposta. O Raspberry Pi[1] foi selecionado como o *hardware* para a implantação, devido ao seu baixo custo, tamanho compacto e baixo consumo energético.

O dispositivo foi modificado para incluir um suporte que possibilita a fixação do *display* ao *hardware*, conforme ilustrado na Figura 4. A intenção é que tal dispositivo permaneça na fazenda, operando através da rede local sem exigir conexão com a Internet.

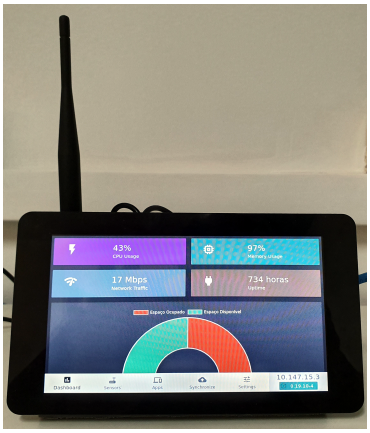


Figura 4 – *Hardware* do *middleware* BigBoxx.

[1] https://www.raspberrypi.org

Para o desenvolvimento das aplicações das camadas pertencentes ao *middleware* BigBoxx a linguagem de programação definida foi JavaScript, por ser compatível com plataformas *desktop*, móvel e *web*. Na Tabela 3 estão presentes as principais tecnologias aplicadas nas camadas da plataforma e-Cattle.

Tabela 3 – Tecnologias usadas nas camadas que compõem o BigBoxx

Camadas	**Tecnologias**
Coleta de Dados	Express JWT
Persistência	Mongoose MongoDB
Serviços	GraphQL

Na camada de Coleta de Dados, as tecnologias utilizadas em sua implementação foram o *framework* Express e o padrão JSON Web Token (JWT). A Camada de Persistência foi desenvolvida utilizando banco de dados não relacional (NoSQL) MongoDB para o armazenamento das informações coletadas e a biblioteca Mongoose. Já a Camada de Serviços recorreu à linguagem de consulta GraphQL em sua construção. A seguir, serão descritas a implementação e o funcionamento desses componentes.

4.3 Dados Sensoriais

Devido à abundância de sensores e diversidade de dados relacionados à pecuária de precisão, foi realizada a padronização das informações. Silva (2018) realizou uma revisão sistemática da literatura relacionando os tipos de dados sensoriais e seus padrões para gerar um modelo de dados para a Camada de Persistência e, assim, criar a modelagem para a classificação semântica dos dados.

Os dados foram categorizados em **comportamental**, referentes ao comportamento do animal, tais como a posição geográfica do animal, a trajetória do animal na área de pastejo, a frequência sob sol ou sombra, entre outros; **microclima**, informações ambientais de um local ou área específica, como umidade e temperatura do ar, radiação solar, temperatura do bulbo úmido, dentre outros; **fisiológicos**, dados fisiológicos individuais de cada animal, entre eles o peso, a altura, a temperatura corporal, a frequência cardíaca; e **contextuais**, informações de sensores que conferem alterações gerais da propriedade, como a condição da porteira (fechada/aberta).

A classificação semântica dos dados foi feita com o banco de dados não relacional (NoSQL) MongoDB, em associação com a biblioteca Mongoose, para modelagem de dados fundamentada em *Schemas*. A representação dos dados é feita por `entidades` com diferentes atributos, por meio de documentos do tipo chave-valor. Para cada sensor, há

uma coleção (*Collection*) de persistência de documentos que o MongoDB usa para mapear um tipo de dado.

Finalizada a estruturação do modelo de dados, Santos (2018) implementou a Camada de Coleta de Dados por meio da aplicação Kernel. Foram desenvolvidos serviços de Barramento RESTful para obter e validar dados, conforme sua classificação semântica, e armazená-los no banco de dados, conforme mostrado na Figura 3.

Para que os dados sensoriais coletados sejam integrados ao e-Cattle, é necessário primeiramente o credenciamento do dispositivo responsável. A Figura 5 mostra o credenciamento de um dispositivo, feito por meio de requisição `HTTP POST` para o endereço `http://<ip-bigboxx>:3000/device`, com os dados do nome, marca, modelo, local, descrição, endereço MAC e um vetor com os sensores que o integram, em formato JSON.

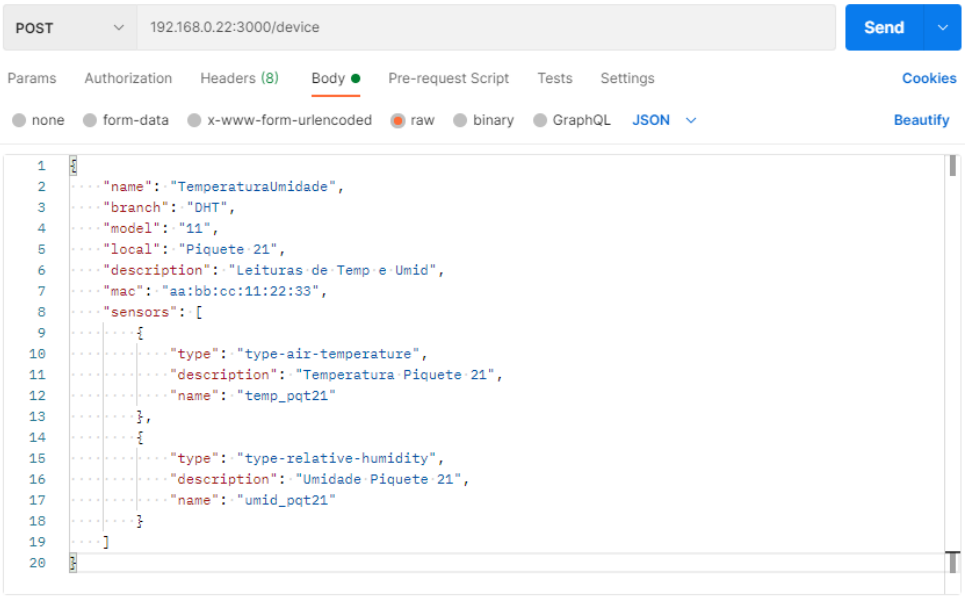


Figura 5 – Credenciamento do dispositivo.

Se os dados da requisição estiverem corretos, um `token` é retornado, e este deverá ser usado como parâmetro no cabeçalho de requisição ao enviar os dados sensoriais, como apresentado na Figura 6.

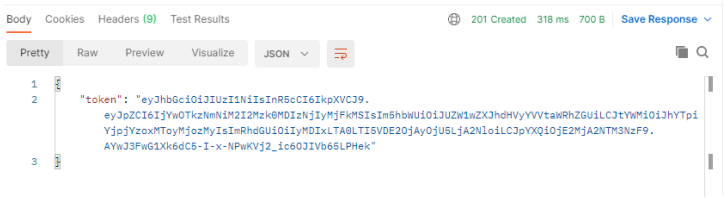
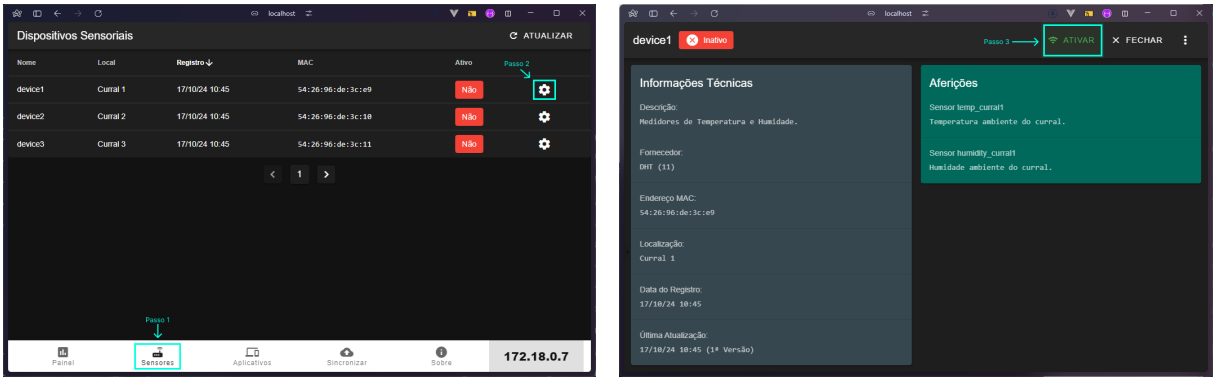


Figura 6 – Resposta do credenciamento do dispositivo.

Em seguida, o usuário deve acessar o Totem UI, que é uma interface gráfica para o gerenciamento do *middleware* Bigboxx, clicar em Sensores (passo 1), selecionar o dispositivo

cadastrado (passo 2) e ativá-lo (passo 3) para que ele possa enviar dados coletados pelos sensores informados, como mostram as Figuras 7a e 7b.



(a) Dispositivos cadastrados. (b) Ativação de dispositivo.

Figura 7 – Ativação de sensor.

Após credenciar o dispositivo, os dados dos sensores podem ser transmitidos por meio de requisições `HTTP POST` usando a URL `http://<ip-do-BigBoxx>:3000/measure` com o endereço MAC do dispositivo, o `token` e os dados dos sensores, como mostrado na Figura 8.

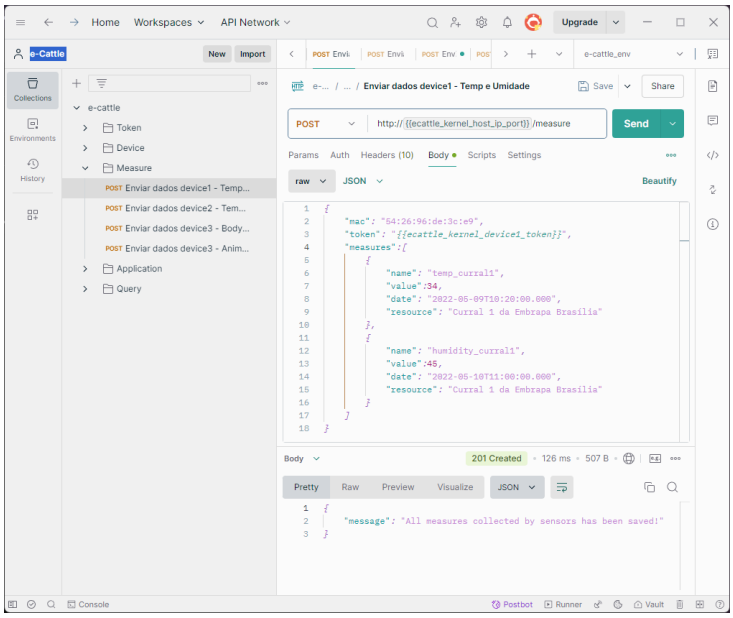


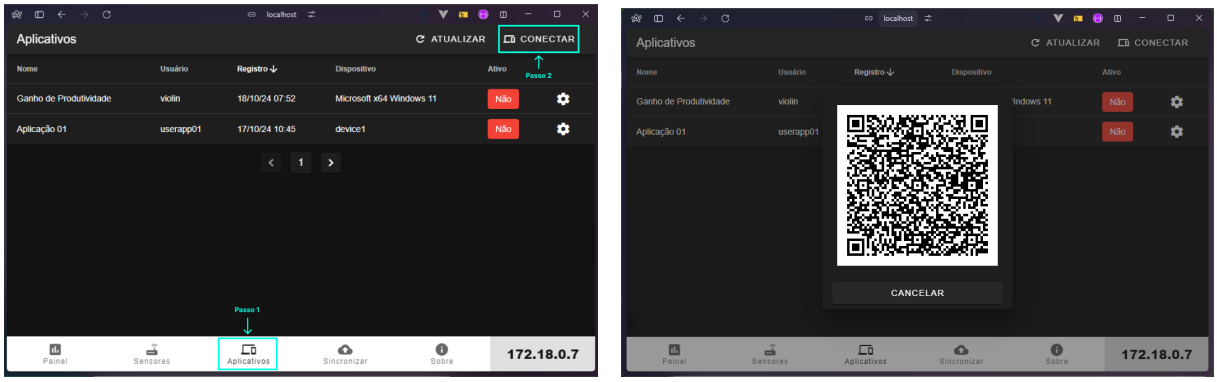
Figura 8 – Envio de dados sensoriais.

4.4 Credenciamento de Aplicações

Para utilizar as informações armazenadas no e-Cattle, as aplicações de alto nível devem ser credenciadas ao barramento GraphQL do BigBoxx. Isso é feito por meio da aplicação Totem UI, acessando `Aplicativos` na barra inferior (Passo 1) e em seguida

no botão `CONECTAR` no canto superior direito da tela (Passo 2), como demonstrado na Figura 9a.

A seguir, um QrCode aparece na tela, como na Figura 9b. Nele constam o endereço IP e um `token JWT` temporário usado para o envio das informações de cadastro da aplicação, quais sejam: código, nome, descrição, usuário, e-mail, imagem e dispositivo. O `token JWT` é válido por um tempo pré-definido, se o usuário exceder esse tempo, deverá refazer o procedimento, senão receberá um `token JWT` definitivo que será usado na obtenção e/ou monitoramento dos dados.



(a) Credenciamento de aplicações. (b) QrCode para credenciamento de aplicação.

Figura 9 – Registro de aplicação no BigBoxx.

Com o Totem UI, o usuário pode executar operações administrativas para habilitar, desabilitar, remover e visualizar os dispositivos que enviarão dados dos sensores e das aplicações que vão usar e/ou monitorar os dados.

A aplicação também possibilita visualizar dados de endereço IP, da interface de rede e das versões atuais do Kernel e do Totem. É possível, ainda, reiniciar e desligar o BigBoxx.

4.5 Consultas e Monitoramentos

Com o componente Query é possível realizar consultas (`Queries`), alterações (`Mutations`) e monitoramentos (`Subscriptions`) dos dados. A implementação do barramento, realizada por Cáceres (2020), usou os *models* da padronização de dados sensoriais desenvolvidos por Silva (2018), utilizando estruturas de *collections* com os campos de cada sensor.

A Tabela 4 apresenta como foram organizadas as categorias que serviram como base para a estruturação dos *models*, a criação dos `System Types` e os `Resolvers`.

A ferramenta *playground* do GraphQL, Figura 10, auxilia o desenvolvedor na elaboração de operações de `queries`, `mutations` e `subscriptions` na API. Essa interface

Tabela 4 – Organização do sistema de tipos de sensores.

Categorias	**Sensores**
Microclima	AirTemperature BlackGlobeTemperature CH4 CO2 DewPointTemperature DryBulbTemperature PH Precipitation RelativeHumidity SoilTemperature SoilMoisture SoilWaterPotencial SoilNitrogen SolarRadiation WaterTemperature WetBulbTemperature WindSpeed
Fisiológico	AnimalWeight BodyTemperature HeartRate RespiratoryFrequency RetalTemperature
Comportamental	Accelerometer AnimalSpeed Gyroscope Magnetometer
Contextual	GateOpened GeographicCoordinate

permite elaborar e simular consultas usando filtros e selecionando os campos que serão retornados na busca.

Permite, ainda, a inclusão de cabeçalho de autenticação por `token JWT`, para o barramento controlar o acesso sobre a aplicação. Cáceres (2020) mostra em detalhes a construção das consultas, Figura 10, que foram implementadas por meio das funções `query`, `value` e `period`, que permitem a filtragem dos campos de cada sensor armazenado no BigBox.

4.6 Considerações Finais

Este capítulo teve como objetivo apresentar o contexto da arquitetura do e-Cattle, sua estrutura em camadas e a implementação de seus componentes por meio do *middleware* BigBoxx. Estes componentes são aplicações desacopladas para simplificar a manutenção e

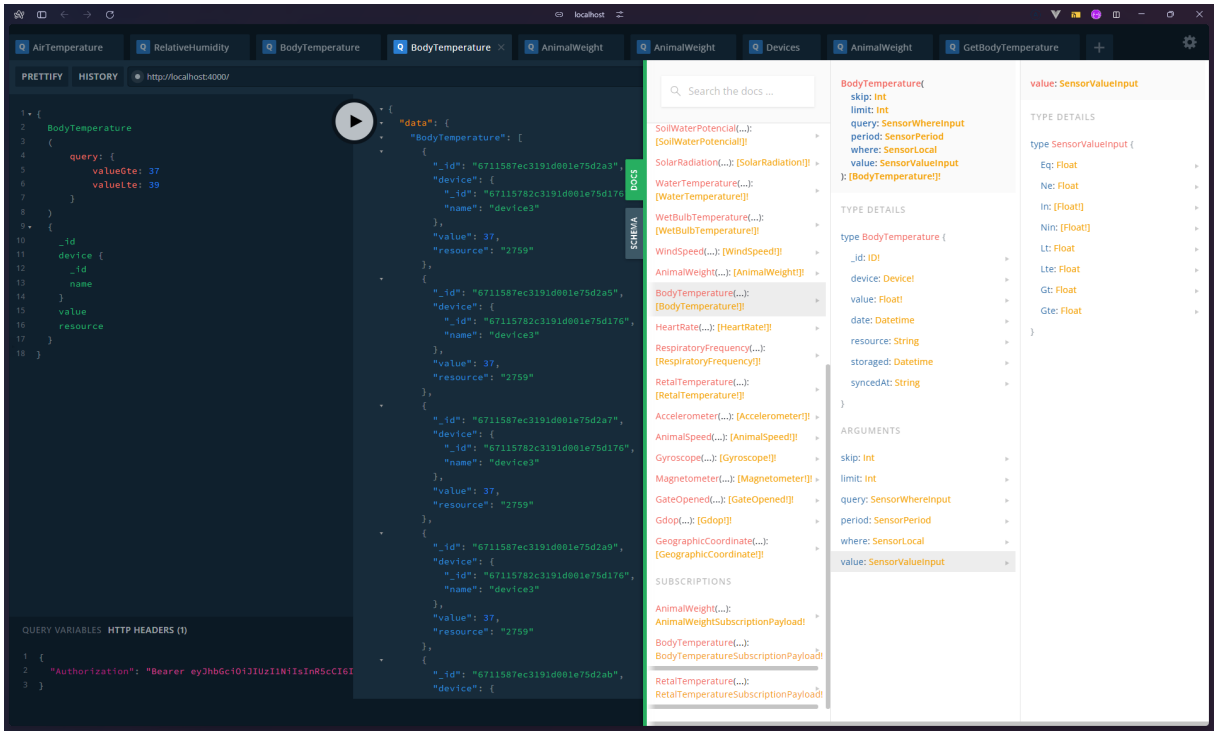


Figura 10 – Ferramenta *playground*.

para poder evoluir com autonomia. Ademais, o componente Kernel foi apresentado em detalhes, mostrando que é formado por um barramento de dados genérico que recebe e valida os dados de entrada, conforme sua classificação semântica, e a persistência no banco de dados. Ressaltando que trata-se de um projeto de código aberto e pode ser encontrado em repositório público[2].

Este capítulo é importante para indicar de onde vêm as informações apresentadas na Camada de Apresentação, objeto deste estudo. No capítulo que segue, serão apresentados os detalhes de uma proposta para implementação da Camada de Aplicação usando a arquitetura de federação de módulos juntamente com a aplicação *web* progressiva.

[2] https://github.com/e-cattle

5 Desenvolvimento do Projeto

Como descrito anteriormente, a plataforma e-Cattle é um sistema que usa IoT para coleta de dados sensoriais na pecuária de precisão, permitindo integrar diversos aplicativos e sensores para monitorar e automatizar o manejo de gado de corte. A plataforma possibilita integrar dados sensoriais de diferentes fontes, padronizar a comunicação entre dispositivos e reduzir a complexidade para os usuários, principalmente pequenos produtores. A arquitetura é multicamadas e utiliza tecnologias como NodeJS, MongoDB e GraphQL. Entre as camadas propostas está a Camada de Aplicação, seção 5.1, objeto deste trabalho. Este capítulo descreve os artefatos desenvolvidos no decorrer deste trabalho.

O restante do capítulo foi organizado da seguinte forma: na seção 5.2, é exposta à arquitetura geral do projeto; já a seção 5.3 traz a estrutura e organização do estudo de caso; seção 5.4, é apresentado o processo de desenvolvimento deste projeto; as principais configurações que possibilitam a integração de micro-frontend com PWA são descritas na subseção 5.4.4; e, por fim, na seção 5.5 são apresentadas as considerações finais discutidas no capítulo.

5.1 Escopo

A camada de Aplicação do e-Cattle é a camada mais externa da arquitetura da plataforma, responsável por interagir com o usuário final e fornecer as informações necessárias para gerenciar sua(s) propriedade(s) rural. Ela é composta por *softwares* de alto nível que utilizam dados dos sensores para fornecer informações e funcionalidades aos usuários. Além disso, é possível combinar dados de diversas origens, como:

- Valor da arroba do boi obtida por meio de *crawlers*;
- Barramento de dados de terceiros;
- Outros bancos de dados da propriedade;
- Informações inseridas pelo usuário.

O escopo da camada de Aplicação é muito amplo e não há uma definição rígida estabelecida pela plataforma e-Cattle. Isso significa que os desenvolvedores têm flexibilidade para criar uma variedade de aplicações que atendam às necessidades específicas de cada produtor. Alguns exemplos de funcionalidades que podem ser implementadas na Camada de Aplicação incluem:

- **Geração de indicadores e relatórios:** Os aplicativos podem processar os dados dos sensores para gerar indicadores sobre o desempenho do rebanho, ganho de peso, consumo de água e de seu comportamento. Esses indicadores podem ser apresentados em painéis (*dashboards*) e relatórios, fornecendo percepções valiosas para o produtor.
- **Emissão de alerta e notificações:** As aplicações podem ser configuradas para monitorar os dados dos sensores e emitir alerta em caso de anomalias, como mudanças bruscas de temperatura, queda no consumo de água ou animais fora de áreas delimitadas. Com essas notificações, o produtor poderá tomar medidas corretivas rapidamente.
- **Automação de processos e tomada de decisão:** Com base nos dados dos sensores e em regras predefinidas, as aplicações podem automatizar atividades, como o controle de portões, o acionamento de sistemas de irrigação e a separação de animais em currais eletrônicos.
- **Integração com outras plataformas e serviços:** As aplicações podem ser conectadas a outras plataformas e serviços, como sistemas de gerenciamento de fazendas, bancos de dados meteorológicos e APIs de mercado, para fornecer informações e funcionalidades adicionais.

O e-Cattle incentiva o desenvolvimento de aplicações de código aberto, permitindo que a comunidade de usuários contribua para a evolução da plataforma. Destaca-se que a camada de Aplicação se beneficia da organização semântica dos dados sensoriais fornecida pelo *middleware* BigBoxx. Dessa forma, as aplicações acessam os dados de forma estruturada e sem dificuldade no entendimento, sem a necessidade de lidar com a complexidade da coleta e do processamento dos dados brutos. Além disso, camada de Aplicação se comunica com a camada de Serviços do e-Cattle por meio de barramento de serviços. O barramento de serviços fornece uma interface padronizada para os aplicativos acessarem os dados dos sensores e outros serviços da plataforma.

5.2 Arquitetura

Conforme relatado na seção anterior, os dados dos sensores são fornecidos pela Camada de Serviços por intermédio do barramento de serviços que fornece uma interface padronizada. A Camada de Aplicação transforma esses dados e, caso necessário, também dados de outras origens, em informações relevantes para a gestão da propriedade. O foco desse trabalho é a Camada de Aplicação, mas é fundamental apresentar a arquitetura dos componentes que integram o BigBoxx para que, em seguida, seja mostrada a estrutura do estudo de caso implementada neste projeto. A Figura 11 mostra os fluxos do funcionamento dos componentes do BigBoxx.

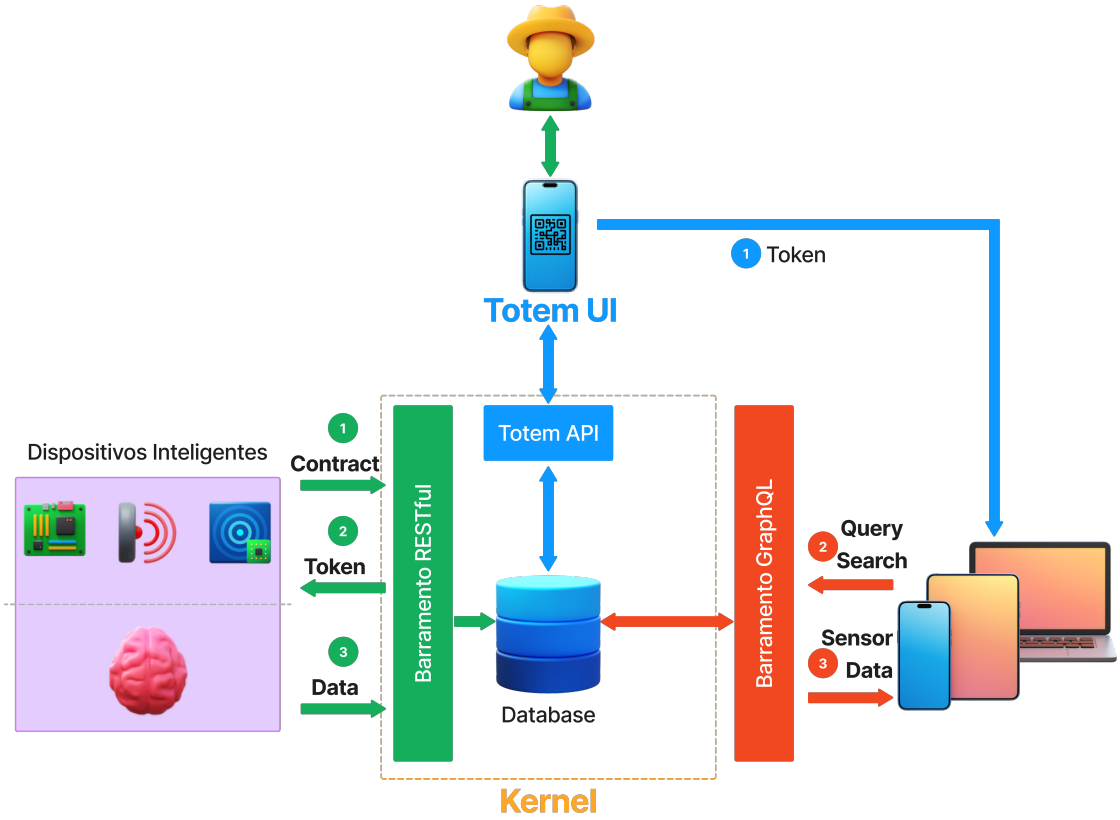


Figura 11 – Fluxo de funcionamento dos componentes que integram o BigBoxx.

Em suma, o fluxo inicia com credenciamento do dispositivo na plataforma por meio de uma requisição HTTP POST com os dados exigidos no formato JSON para criar o contrato, como detalhado na seção 4.4.

A aplicação *backend* Totem API dispõe de um barramento de dados exclusivo contendo todos os *endpoints* que a aplicação *frontend* Totem UI precisa para expor as informações ao usuário. Por esse barramento é possível credenciar e habilitar os dispositivos que enviam os dados sensoriais. Por outro lado, as aplicações de alto nível consomem os dados no barramento GraphQL, como ilustrado pela Figura 11.

Além do fluxo convencional do funcionamento dos componentes integrantes do BigBoxx é possível realizar o monitoramento de sensores e enviar notificações de acordo com valores pré-definidos de filtros, conforme ilustrado na Figura 12.

Uma aplicação de alto nível pode monitorar as coletas de dados sensoriais por meio de uma `Subscription` com o nome do sensor, o filtro com o valor que deseja receber a notificação e o `token JWT` da aplicação.

Um *WebSocket* é fornecido pelo Barramento GraphQL à aplicação cliente para que ela possa monitorar os eventos recebidos pelo Redis[1]. Dessa forma, a aplicação sempre será

[1] https://redis.io

notificada quando um evento que estiver em consenso com os requisitos do filtro ocorrer.

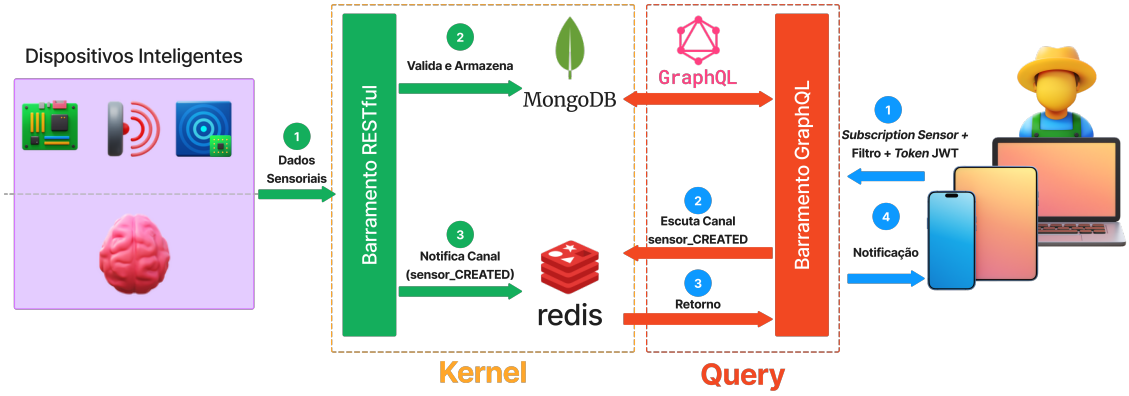


Figura 12 – Monitoramento dos sensores.

Até aqui, as arquiteturas e processos relatados têm como referência o trabalho de Cáceres (2020). Há uma ligação intrínseca entre os projetos, uma vez que este trabalho utiliza como principal fonte de dados o Barramento GraphQL e se comunica com o Totem UI.

5.3 Estrutura do Estudo de Caso

O método do Estudo de Caso (EC) é muito usado em diversos estudos de campo onde é possível fazer observações diretas, entrevistas sistemáticas e levantamentos, em especial ao procurar explicações aprofundadas sobre o fenômeno estudado. Ele tem sido visto como como uma maneira de gerar *insights* exploratórios e, apesar de suas fragilidades, este método tem tido um uso extensivo em diversas áreas pesquisa (ARAÚJO et al., 2008).

Apresentadas essas arquiteturas de componentes e como é realizado o monitoramento dos sensores, na sequência será mostrada a estrutura do estudo de caso projetada para este trabalho.

O *frontend* implementado emprega a arquitetura de micro-frontends por meio de federação de módulos e também a tecnologia de PWA em sua interface. Para a análise de viabilidade de uso concomitante dessas tecnologias, foram estabelecidas algumas aplicações autônomas e componentes para o *frontend* que podem ser reutilizados. Dessa forma, foi criada uma aplicação *host* e outras aplicações *remotes* que serão detalhadas mais adiante.

Como pode ser visto na Figura 13, cada aplicação possui seus componentes que podem ser expostos e/ou consumidos pelas outras aplicações. Neste caso, temos a aplicação *host* que recebe os componentes de outras aplicações e não expõe nenhum componente.

Apesar da autonomia das aplicações, há uma aplicação principal (*host*) e, como pode ser percebido, ela pode ser montada conforme a necessidade e disponibilidade de

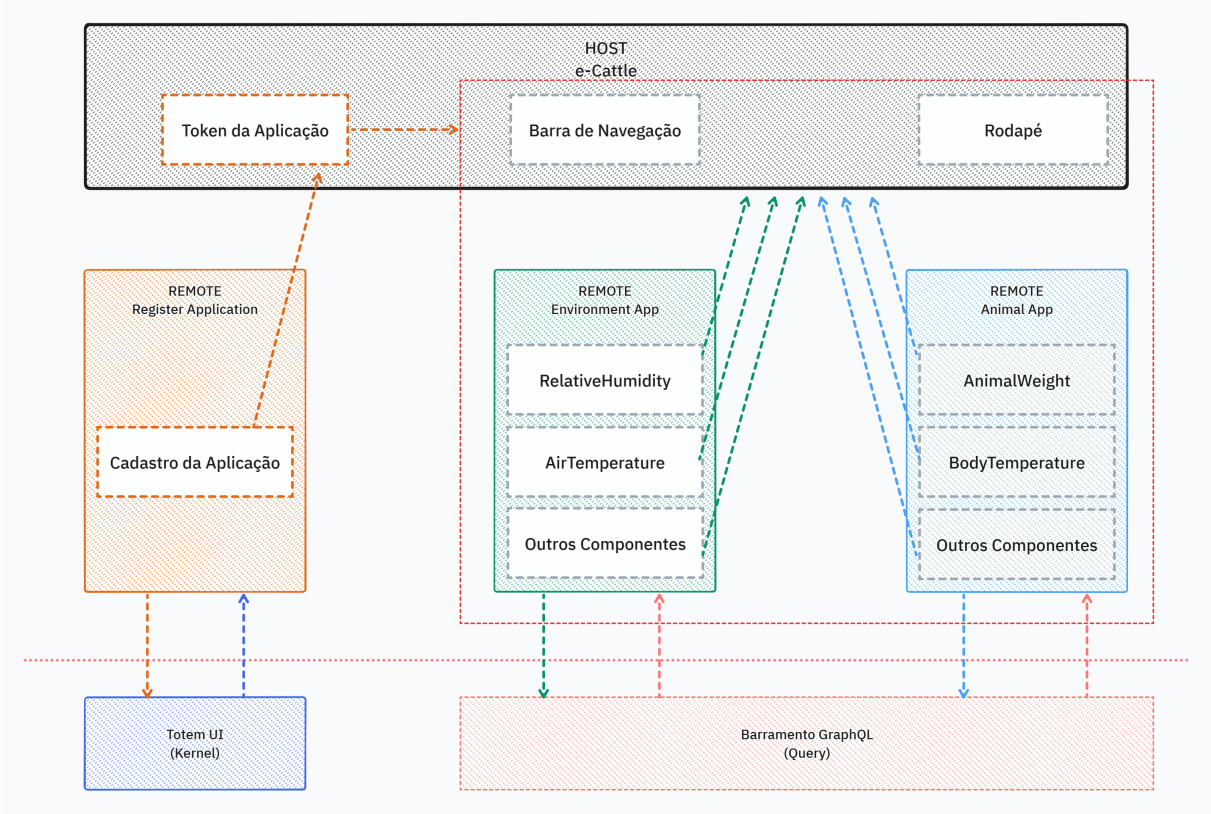


Figura 13 – Arquitetura do Micro-frontend.

sensores de cada propriedade rural. Bastando para isso que aplicações *remotes* sejam desenvolvidas e integradas à aplicação principal.

Ao trabalhar com a arquitetura de micro-frontend, é importante atentar-se para haver uma divisão adequada das aplicações. Aplicação muito pequena gera muita fragmentação, mas se for demasiadamente grande, a complexidade de acoplamento desnecessária será proporcional.

Normalmente, as aplicações remotas são desenvolvidas tendo como base um microsserviço, mas neste caso temos como fonte de dados apenas o Barramento GraphQL, embora possa haver outras fontes externas. Dessa forma, a segmentação foi feita por tipos de captação de dados dos sensores, ou seja, há sensores para coleta de informações ambientais e de animais. Assim, reduzimos a granularidades e simplificamos a manutenção.

Também foi criado um aplicativo remoto para o credenciamento da aplicação no BigBoxx, ele faz a leitura do QRCode no Totem UI, armazena os dados necessários da aplicação principal no Totem API e guarda o *token* definitivo da aplicação localmente. Apesar de ser uma aplicação pequena, ela foi criada separadamente para que aplicações remotas possam ser credenciadas, caso necessário, individualmente sem a necessidade de uma aplicação *host*.

5.4 Processo de Desenvolvimento

Este trabalho tem como usuário final desenvolvedores e *software houses*, visto que o produto resultante é uma CLI para simplificar a criação de aplicações *frontend* para a plataforma e-Cattle. Para validar a integração entre as diversas tecnologias e a arquitetura proposta para a implementação deste projeto, um estudo de caso foi implementado, seguindo a ordem dos objetivos específicos deste trabalho. Antes de desenvolver o estudo de caso, foram feitos alguns experimentos para analisar e compreender melhor as tecnologias e arquitetura utilizadas, como:

- A construção de protótipos micro-frontend utilizando somente Vue.js para o *template* do *host* e para os *templates* de *remotes*;
- A implementação de micro-frontend usando Vue.js no *template* do *host* e os *templates* dos *remotes* feitos com Vue.js e React.js; e
- As duas prototipagens anteriores integrando micro-frontend com PWA.

Essa integração de tecnologias, mencionada anteriormente, entre micro-frontends e *pwa*, foi complexa, uma vez que o material de pesquisa, sejam artigos, documentações oficiais ou mesmo literatura cinzenta (incluindo fóruns, artigos em formato de tutorial e vídeos), ainda é escasso. Para que tudo funcionasse adequadamente, foi necessário instalar alguns *plugins*, descritos no Capítulo 2, e realizar diversas configurações para que todos os recursos de ambas as tecnologias funcionassem corretamente. A prototipagem foi importante para o processo por indicar os *plugins* e configurações compatíveis.

Durante a implantação do estudo de caso foram criados três micro-frontends, sendo um *host* e dois *remotes*, que serão detalhados a seguir. Estas aplicações permitiram analisar e estudar as configurações em profundidade, necessário para abarcar todas as tecnologias utilizadas. A partir da homologação, os *templates* foram criados para serem parte da CLI, que serão apresentadas no decorrer deste capítulo.

5.4.1 Estudo de caso

O estudo de caso foi desenvolvido para identificar todos requisitos necessários para criar uma aplicação micro-frontend com recursos PWA para a plataforma e-Cattle.

A primeira aplicação do estudo de caso a ser criada foi a *host*, contendo uma página inicial desenvolvida para ser simples e intuitiva, como pode ser visto na Figura 14, contendo uma *landing page* que oferece as opções de cadastrar-se ou acessar a *dashboard*. Além disso, no canto inferior esquerdo, o usuário pode alterar o tema de visualização do conteúdo, optando entre claro e escuro.

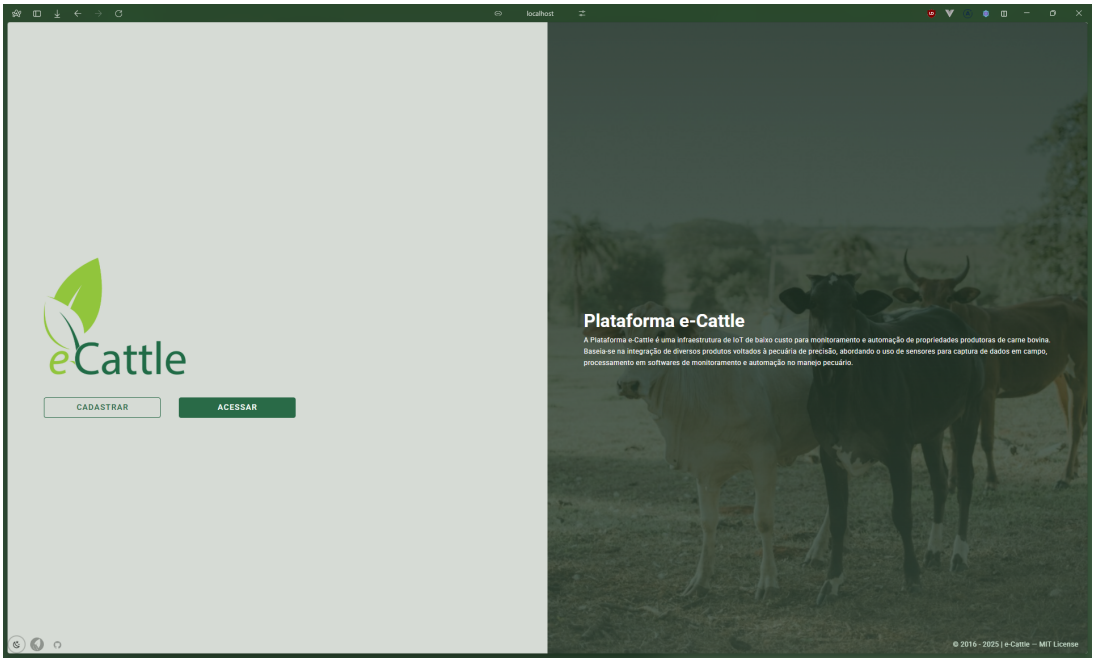
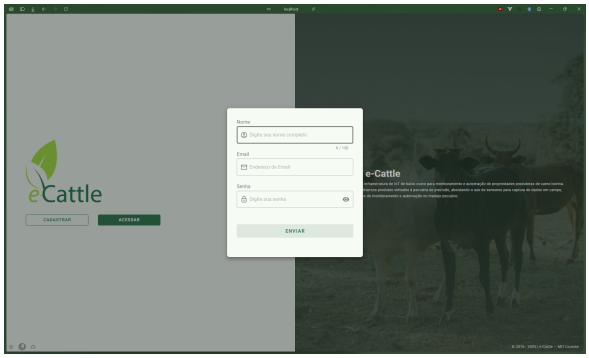
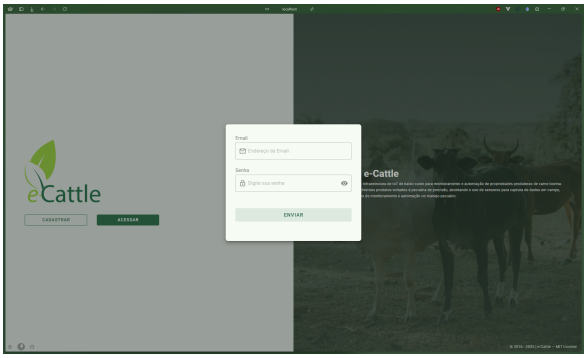


Figura 14 – Página inicial do *host*.

A opção Cadastrar apresenta um formulário com dados básicos (nome, *e-mail* e senha) para ser preenchido conforme mostra a Figura 15a; este formulário é um componente do *host*.



(a) Formulário de cadastro de usuário.

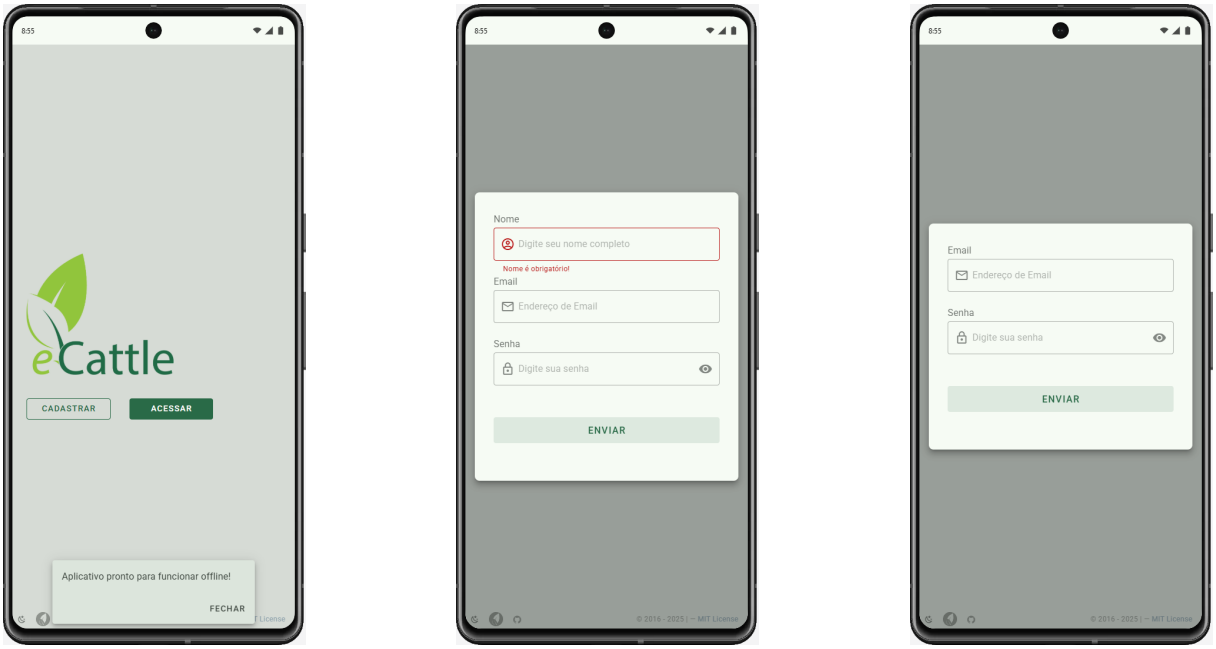


(b) Formulário de acesso do usuário.

Figura 15 – Cadastro e acesso para o *Dashboard*.

Algo similar ocorre ao clicar no botão Acessar, como pode ser visto na Figura 15b, a diferença é que o formulário que aparece solicita ao usuário apenas um *e-mail* e uma senha anteriormente registradas e que sejam válidas. Assim como o anterior, este formulário também é um componente do *host*.

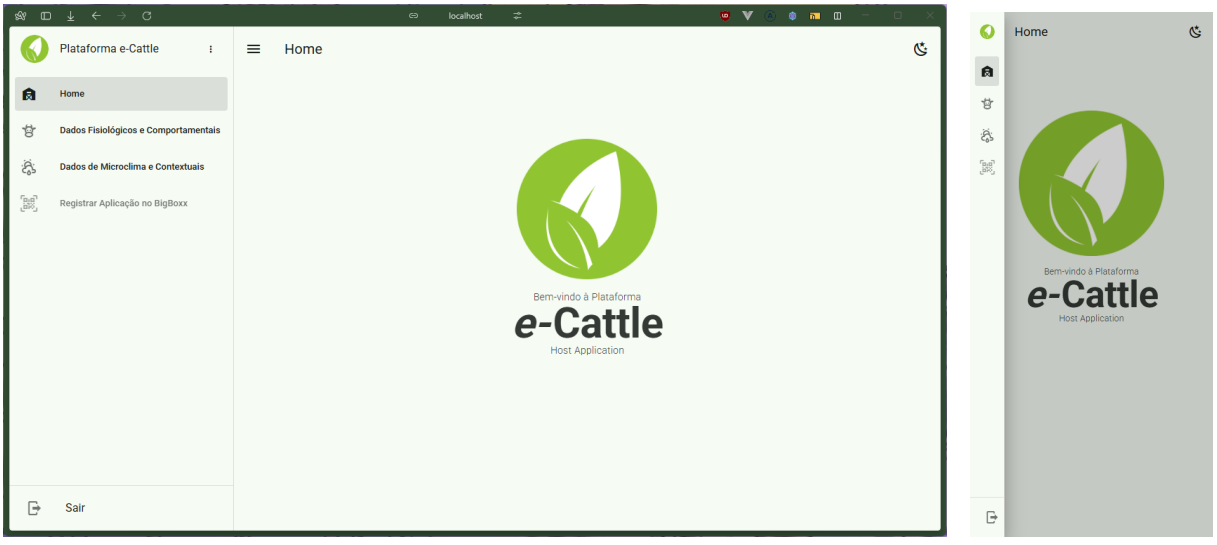
As aplicações e os *templates* desenvolvidos para este projeto são responsivos, como mostra a Figura 16. Eles se adequam às diferentes resoluções de tela e são tão imersivos quanto um aplicativo nativo, pois podem ser instaladas no dispositivo. Quando instalados no dispositivo, boa parte das funcionalidades podem ser utilizadas sem conexão com a Internet, isto porque foram implementadas para serem PWAs.



(a) *Landing Page*. (b) Cadastro de usuário. (c) Formulário de *login*

Figura 16 – *Template* apresentado em versão para dispositivo móvel.

Além dos componentes de formulário usados na *landing page* para acessar a área restrita, o *host* provê outros componentes que formam a página do *dashboard*, como a barra lateral para navegação principal, a barra de cabeçalho que mostra qual aplicação está sendo mostrada em determinado momento e o componente responsável por registrar a aplicação no BigBoxx, como pode ser visto na Figura 17. Este último pode tanto ser um componente do *host*, como pode ser utilizado por meio de uma aplicação *remote*.



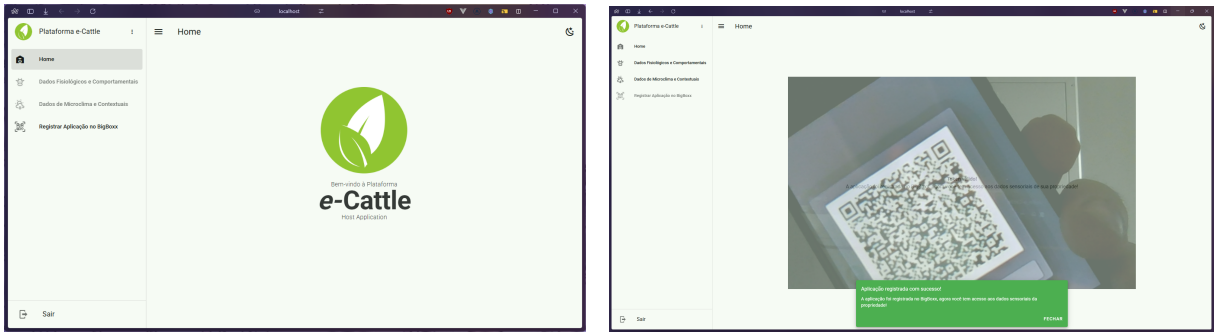
(a) Versão *web*. (b) Versão móvel.

Figura 17 – *Dashboard*

Para ter acesso aos dados dos sensores, é preciso cadastrar a aplicação no BigBoxx, responsável pela implementação das camadas de Coleta de Dados, Persistência e Serviço

dentro da plataforma e-Cattle. A opção de **Registrar Aplicação no BigBoxx** ficará acessível no menu lateral caso a aplicação ainda não tenha sido registrada, veja na Figura 18a. O credenciamento é simples, basta acessar o BigBoxx, selecionar a opção **Aplicativos** no menu inferior e em seguida clicar no botão **Conectar** no canto superior direito, em seguida surgirá um QrCode contendo o IP e um *token*, conforme descrito na seção 4.4.

Esse código deve ser capturado pela aplicação (Figura 18b) que deseja consumir os dados sensoriais. As informações serão extraídas do QrCode e, em seguida, deverá realizar uma requisição, enviando-as juntamente com outros dados, como: código, nome, descrição, usuário, *e-mail*, imagem e dispositivo. Os outros dados são coletados no `package.json` e no navegador (dados do dispositivo), não sendo necessário digitá-los em um formulário. Finalizado o credenciamento, será retornado um *token* que deverá ser usado durante as requisições aos dados sensoriais.



(a) Aplicação sem cadastro no BigBoxx. (b) Aplicação em cadastro no BigBoxx.

Figura 18 – Processo de registro de uma aplicação no BigBoxx.

Ainda como parte do estudo de caso, uma aplicação remota consumindo dados sensoriais foi implementada. A Figura 19 mostra esse *remote* sendo hospedado e apresentando informações após o credenciamento da aplicação *host* no BigBoxx.

Nessa aplicação remota são apresentados os dados de temperatura corporal dos animais. Os dados sensoriais podem, entre outras coisas, serem usados para gerar componentes gráficos, séries temporais, séries históricas e mostrar valores individuais ou agrupados. Ressalta-se que outros sensores podem ser utilizados, como pode ser visto na Tabela 4. Da mesma forma, outras aplicações poderão ser desenvolvidas e compartilhadas para serem usadas em *hosts* de outras propriedades implementadas por desenvolvedores distintos.

5.4.2 Modelos

As aplicações apresentadas serviram para análise e validação da arquitetura, módulos, *plugins* e configurações que foram utilizadas nos *templates* desenvolvidos para os *hosts* e para os *remotes*.

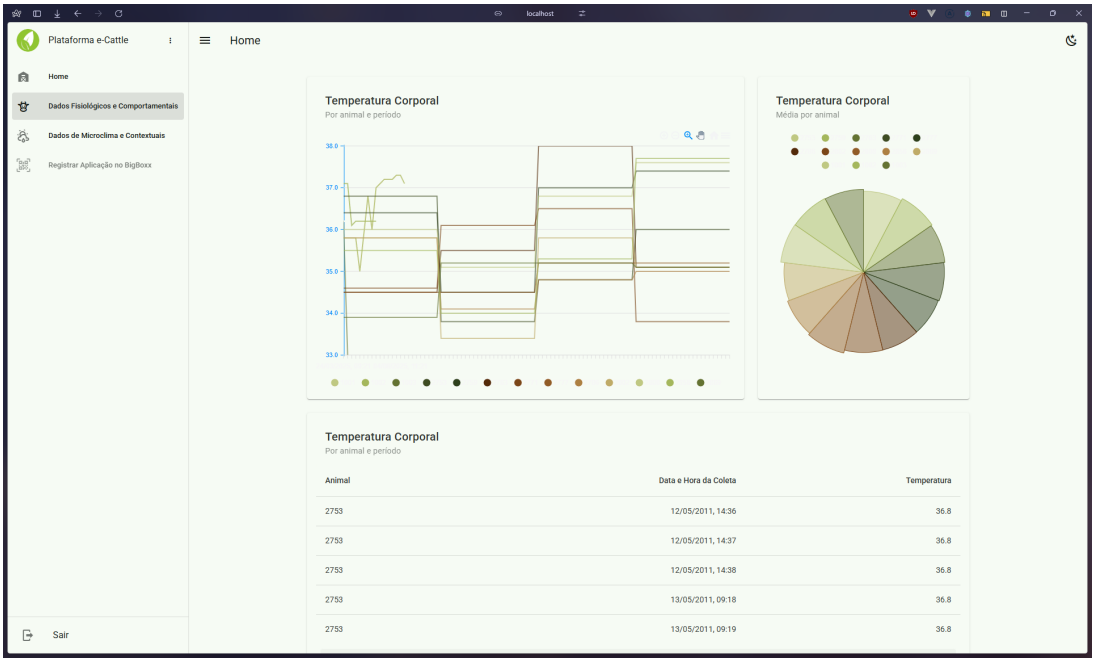


Figura 19 – Aplicação *remote* com dados fisiológicos.

Os *templates* principais foram desenvolvidos e disponibilizados em repositório do projeto e-Cattle. Inicialmente foram criados modelos:

- Base[2] - Esta aplicação é o ponto de partida para iniciar (*build*) todas as aplicações *host* e *remote* que integram o projeto. Entre os *templates* desenvolvidos, este é o mais simples, uma vez que seu objetivo é a inicialização (*build*), em ordem pré-definida, das aplicações que fazem parte do projeto micro-frontend.

- *Host*[3] - *Template* para uma aplicação hospedeira (Host), neste modelo estão instalados todas as bibliotecas e os *plugins*, assim como foram feitas todas as configurações necessárias para a construção de uma aplicação micro-frontend e com todos os recursos de uma PWA.

- *Host Credential*[4] - Estrutura e configurações iguais ao modelo Host, entretanto vem com um componente de credencial para aplicação no BigBoxx.

- *Base Host Credential*[5] - Mesma estrutura e funcionalidade da Base, porém já traz um *template host credential* integrado.

- *Remote*[6] - Similar ao *template* anterior, diferenciando-se pela criação de uma aplicação do tipo *remote* e com suas características e configurações específicas.

2 https://github.com/e-cattle/mfe-base
3 https://github.com/e-cattle/mfe-host-vue
4 https://github.com/e-cattle/mfe-host-credential-vue
5 https://github.com/e-cattle/mfe-base-host-vue
6 https://github.com/e-cattle/mfe-remote-vue

- *Remote with credentials*[7] - Este *template*, além da estrutura e configurações básicas de um *remote*, tem como cerne o credenciamento de uma aplicação no *middleware* BigBoxx, podendo ser utilizada tanto por aplicações *host*, como por outros *remotes*, uma vez que uma aplicação *remote* é totalmente funcional e independente.

5.4.3 CLI: Create e-Cattle

Para simplificar o processo de criação de estruturas iniciais para projetos de micro-frontend para a Plataforma e-Cattle, foi proposta e desenvolvida uma aplicação que utiliza modelos (*templates*) previamente desenvolvidos (subseção 5.4.2) com base no estudo de caso, mostrado na subseção 5.4.1. Create e-Cattle[8] é uma *Command-Line Interface* (CLI), em português, Interface de Linha de Comando, que simplifica o processo de criação e configuração de aplicações micro-frontends integradas com PWA para a Plataforma e-Cattle.

A CLI clona os *templates* armazenados em repositórios do Github, relacionados na subseção 5.4.2. Por se tratar de uma CLI, os comandos são executados no terminal ou *prompt* de comando, como mostra a Figura 20.

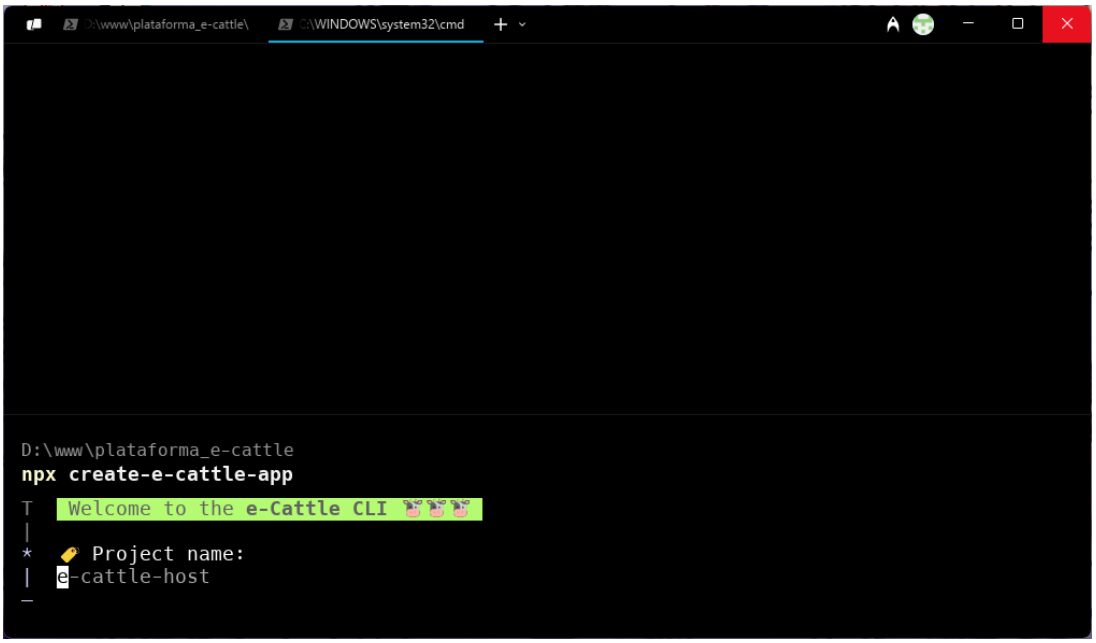


Figura 20 – Execução da CLI Create e-Cattle.

Esta CLI foi desenvolvida na linguagem JavaScript no ambiente Node.js[9] e publicada no gerenciador de pacotes NPM[10] e, por isso, pode ser usado em qualquer computador conectado à Internet, independente de sistema operacional. Isto permite o compartilha-

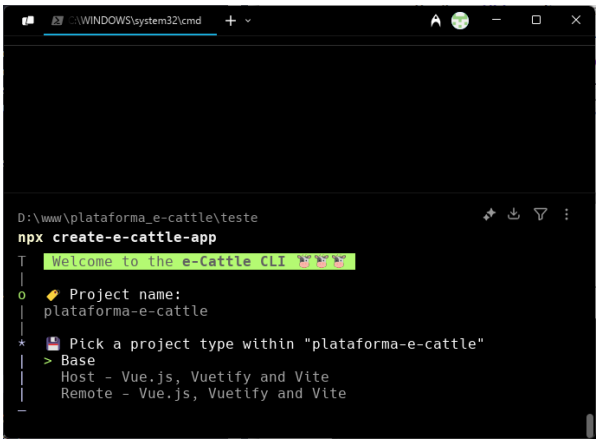
[7] https://github.com/e-cattle/mfe-remote-credential-vue
[8] https://www.npmjs.com/package/create-e-cattle-app
[9] https://nodejs.org
[10] https://www.npmjs.com

mento com outros programadores interessados em usar ou contribuir com a evolução, ampliação e manutenção do projeto.

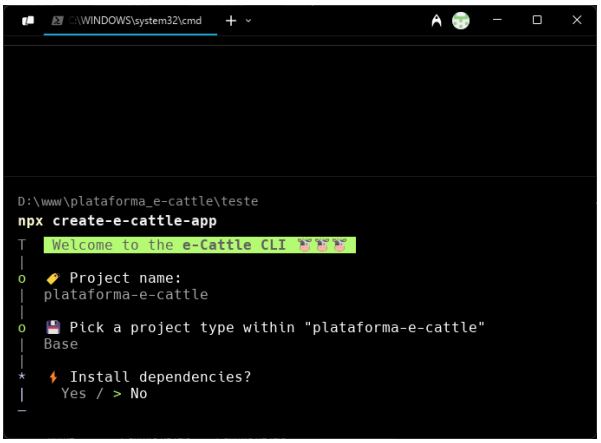
Uma vez executado o comando ‘npx create-e-cattle-app‘, será solicitado o nome do projeto, apresentado na Figura 20. Em seguida, deverá ser escolhido o tipo do projeto, conforme listado na subseção 5.4.2, veja a Figura 21a.

Inicialmente, os *templates* foram criados utilizando o padrão estabelecido na Embrapa Gado de Corte, ou seja, utilizando linguagem de programação JavaScript, Vue.js, Vuetify e PWA. Mas, por se tratar de uma arquitetura de micro-frontends, modelos baseados em outras *frameworks* poderão ser criados e incluídos futuramente.

Por fim, a CLI permite que o usuário escolha entre instalar ou não as dependências, como pode ser visto na Figura 21b.



(a) Seleção de tipo.



(b) Escolha sobre instalação das dependências.

Figura 21 – CLI Create e-Cattle em funcionamento.

Pensada para ser simples, a CLI apresenta apenas três perguntas e uma mensagem final orientando o usuário a acessar a pasta do projeto e executá-lo para ver o resultado. Seu código fonte está armazenado no repositório on-line da plataforma e-Cattle[11] de forma pública.

Como dito anteriormente, outros *templates* poderão ser incluídos. Da mesma forma, a estrutura e a coleta de informações da CLI podem ser alteradas adicionando novas questões conforme a necessidade e evolução da plataforma.

5.4.4 Manutenção e Ampliação

Uma aplicação costuma ser um similar a um organismo vivo e, dessa forma, requer cuidados ao longo de sua vida. Assim, a manutenção, ampliação e evolução das aplicações hospedeiras ou remotas contará com um repositório on-line específico para tal, conforme a

[11] https://github.com/e-cattle/create-e-cattle-app

seção anterior. Como trata de um projeto de código aberto, quem quiser contribuir com novas aplicações remotas, sugerir alterações, indicar erros ou apenas consumir, poderá fazê-lo.

Inicialmente, a CLI disponibiliza os *templates* listados na 5.4.2, que poderão ser utilizados como base ou como referência para a implementação de novos projetos.

O objetivo desse repositório é servir como ponto de partida para futuros desenvolvedores que adotarem a Plataforma e-Cattle, podendo ser utilizado também para compartilhar as aplicações desenvolvidas, propor novas implementações, reportar *bugs* e sugerir melhorias e evoluções.

5.5 Considerações Finais

Durante a revisão literária, Capítulo 3, não foi encontrado trabalho que incorporasse a arquitetura e as tecnologias proposta neste projeto. E, apesar das dificuldades em encontrar material de apoio até mesmo na literatura cinzenta, o resultado esperado foi atingido. Já que as metas propostas foram todas implementadas e as integrações de arquitetura e tecnologias puderam ser incluídas sem que uma interferisse na outra.

O resultado permitiu validar a estratégia de construção de modelos padronizados que agregam micro-frontends e PWA baseados em um estudo de caso. Possibilitando, dessa forma, a criação de uma ferramenta de automação, Create e-Cattle, CLI para agilizar e descomplicar a criação de aplicações *frontend* para consumir dados sensoriais da plataforma e-Cattle.

Esta CLI auxiliará os programadores, ou *software houses*, na criação de aplicações *frontend* utilizando os dados sensoriais coletados na propriedade rural, por meio da plataforma e-Cattle, trazendo benefícios gerenciais aos produtores que adotarem a plataforma. Dessa forma, o comportamento dos animais poderá ser monitorado e gerar informações da posição geográfica, trajetória na área de pastejo, permanência sob o sol ou sombra, entre outras. Além disso, poderão ser desenvolvidas aplicações sobre os dados fisiológicos como, por exemplo, peso, temperatura corporal, frequência cardíaca, entre outras. Além do rebanho, o ambiente e a propriedade também podem ser monitorados por meio de aplicações que podem receber dados sensoriais coletados acerca da umidade e temperatura do ar, da radiação solar, da situação das porteiras (aberta ou fechada), entre outras. Adicional, os dados semânticos podem ser cruzados para potencializar os recursos e ganhos de produtividade.

6 Conclusão

Este trabalho apresenta uma abrangente pesquisa entorno da integração de novas tecnologias aplicadas à pecuária de precisão, com foco principal na implementação de modelos micro-frontend para a plataforma e-Cattle. Esta plataforma foi projetada para aumentar a eficiência e a eficácia da pecuária, utilizando, para isso, infraestrutura de IoT, que auxilia o monitoramento em tempo real e a automação dos processos de produção em propriedades rurais. A adoção dessas tecnologias é essencial para o avanço da agropecuária de precisão no Brasil, setor que contribui significativamente com a economia nacional.

A estrutura da plataforma e-Cattle emprega uma arquitetura de barramento de serviços que permite a comunicação entre diversos componentes tecnológicos. Este projeto promove uma abordagem evolutiva e integrada para a automação da gestão rural, podendo simplificar as complexidades associadas às práticas agrícolas tradicionais. Este trabalho enfatiza a integração de diversas tecnologias, incluindo micro-frontends e PWAs, importante para enfrentar os desafios encontrados na apresentação dos dados coletados e processados.

A integração mencionada no parágrafo anterior só foi possível após muita análise, estudo e testes de validação. O Capítulo 5 apresentou um estudo de caso que foi fundamental para a conclusão deste trabalho, pois a partir dele surgiram modelos padronizados que serão utilizados para construção de novas aplicações micro-frontends. Estes modelos estão em repositórios online e servem como base para CLI criada como objetivo principal deste projeto.

A CLI Create e-Cattle agiliza a geração de novas aplicações *frontend* para a plataforma e-Cattle, pois se baseia em modelos pré-implementados com as principais configurações relacionadas a micro-frontends e às PWAs. Assim, o desenvolvedor que utilizar a ferramenta poderá se dedicar somente ao planejamento e execução das funcionalidades específicas de seu projeto.

Esta CLI poderá ser ampliada com a inclusão de novos *templates*, que poderão ser básicos com simples configurações ou complexos com funcionalidades avançadas, *layouts* bem elaborados ou com ambos. Além disso, novas tecnologias poderão ser agregadas e modelos com outras *frameworks* poderão ser criados e incorporados a CLI. Como proposta para futuras melhorias estão: novos módulos básicos; módulos específicos com uso de dados sensoriais coletados pela plataforma; e atualização dos módulos e *plugins* atuais.

A partir disso, a ferramenta Create e-Cattle permitirá criar diversas micro-aplicações *frontend* para gestão de microclima e de contexto da propriedade, assim como dados fisiológicos e comportamentais do rebanho. Utilizando os dados de microclima será possível criar aplicações para monitorar a temperatura do ar, estresse térmico, CH4, CO2, temperatura

do ponto de orvalho, temperatura do bulbo seco, PH, precipitação, umidade relativa, temperatura do solo, umidade do solo, potencial hídrico do solo, nitrogênio do solo, radiação solar, temperatura da água, temperatura do bulbo úmido e velocidade do vento. Da mesma forma, os dados contextuais poderão ser incorporados as aplicações que consumam às coordenadas geográficas da propriedade e para verificar a situação das porteiras (aberta ou fechada). Por outro lado, os dados do rebanho podem ser utilizados para monitorar as condições fisiológicos, como: peso animal, temperatura corporal, frequência cardíaca, frequência respiratória, e temperatura real. Adicionalmente, o comportamento dos animais geram informações importantes e podem ser acompanhados por meio de aplicações, tais como: posição geográfica do animal, trajetória do animal na área de pastejo, frequência sob sol ou sombra, entre outros.

Os dados sensoriais coletados, apresentados acima, podem gerar aplicações individuais, conforme sua classificação semântica. Mas também, dados de sensores distintos podem ser cruzados para gerar relatórios e analisar suas correlações como, por exemplo, observar se a temperatura corporal do animal interfere no ganho de peso. Ademais, novas tecnologias, como agentes de inteligência artificial, podem ser incorporados, potencializando os recursos e trazendo novos benefícios gerenciais ao produtor rural.

Em suma, este trabalho alcançou as metas e objetivos gerais e específicos definidos, contribuindo com a comunidade acadêmica e agropecuária no que tange a gestão e apresentação de informações para apoiar a tomada de decisões em propriedades com foco na pecuária de corte usando novas tecnologias para o avanço da pecuária de precisão.

Referências

ABDULLAH, M.; IQBAL, W.; ERRADI, A. Unsupervised learning approach for web application auto-decomposition into microservices. Journal of Systems and Software, v. 151, p. 243–257, 2019. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121219300408>. Citado na página 30.

ABIEC. Beef Report 2024 - Perfil da Pecuária no Brasil. [S.l.], 2024. Disponível em: <https://www.abiec.com.br/publicacoes/beef-report-2024-perfil-da-pecuaria-no-brasil/>. Acesso em: 23 abr 2025. Citado na página 18.

APOLLO. Vite module federation documentation. 2024. Apollo documentation. Disponível em: <https://www.apollographql.com/docs/react>. Acesso em: 12 nov 2024. Citado na página 39.

ARAÚJO, C. et al. Estudo de caso. 2008. Citado na página 60.

BELHADI, A.; ZHANG, M.; ARCURI, A. Random testing and evolutionary testing for fuzzing graphql apis. ACM Trans. Web, Association for Computing Machinery, New York, NY, USA, v. 18, n. 1, jan. 2024. ISSN 1559-1131. Disponível em: <https://doi.org/10.1145/3609427>. Citado na página 38.

BUCCI, G.; BENTIVOGLIO, D.; FINCO, A. Precision agriculture as a driver for sustainable farming systems: State of art in literature and research. Quality - Access to Success, v. 19, p. 114–121, 03 2018. Citado na página 18.

CARROMEU, C. e-Cattle: Uma Plataforma de IoT para Pecuária de Precisão. Tese (Doutorado) — Universidade Federal de Mato Grosso do Sul, nov 2019. Citado 6 vezes nas páginas 17, 19, 20, 47, 48 e 49.

CáCERES, B. de A. Uma arquitetura de consulta semântica de dados sensoriais no barramento de serviços IoT na Plataforma e-Cattle. Dissertação (Mestrado) — Universidade Federal de Mato Grosso do Sul - UFMS, nov 2020. Citado 3 vezes nas páginas 53, 54 e 60.

DOYLE, B.; LOPES, C. V. Survey of technologies for web application development. arXiv preprint arXiv:0801.2618, 2008. Citado na página 30.

GALLAUGHER, J. M.; RAMANATHAN, S. C. Choosing a client/server architecture: a comparison of two-and three-tier systems. Information Systems Management, Taylor & Francis, v. 13, n. 2, p. 7–13, 1996. Citado 2 vezes nas páginas 30 e 31.

GEERS, M. Micro frontends—extending the microservice idea to frontend development. https://micro-frontends.org, 2019. Acesso em: 26 mai 2025. Citado na página 29.

GOOGLE. PWA - Como começa. 2021. PWA - Como começar. Disponível em: <https://web.dev/learn/pwa/getting-started?hl=pt-br>. Acesso em: 07 nov 2024. Citado na página 27.

IBM. Architectural characteristics of web-based applications. January 2025. Disponível em: <https://www.ibm.com/docs/en/db2-for-zos/12.0.0?topic=environment-architectural-characteristics-web-based-applications>. Acesso em: 27 mai 2025. Citado na página 31.

KAUSHIK, N.; KUMAR, H.; RAJ, V. Micro frontend based performance improvement and prediction for microservices using machine learning. J. Grid Comput., Springer-Verlag, Berlin, Heidelberg, v. 22, n. 2, abr. 2024. ISSN 1570-7873. Disponível em: <https://doi.org/10.1007/s10723-024-09760-8>. Citado 4 vezes nas páginas 42, 43, 44 e 45.

MAZZARA, M.; MEYER, B. et al. Present and ulterior software engineering. [S.l.]: Springer, 2017. Citado 3 vezes nas páginas 30, 31 e 32.

MENA, M. et al. A progressive web application based on microservices combining geospatial data and the internet of things. IEEE Access, v. 7, p. 104577–104590, 2019. Citado 3 vezes nas páginas 41, 42 e 44.

MEZZALIRA, L. Building Micro-Frontends. [S.l.]: "O'Reilly Media, Inc.", 2021. Citado 7 vezes nas páginas 20, 23, 24, 25, 26, 28 e 32.

MILLER, B.; ATKINSON, R. D. Raising european productivity growth through ict. ITIF, June, 2014. Citado na página 19.

ORIGIN.JS. Vite module federation documentation. 2022. Vite plugin federation documentation. Disponível em: <https://github.com/originjs/vite-plugin-federation>. Acesso em: 12 nov 2024. Citado na página 37.

PAVLENKO, A. et al. Micro-frontends: application of microservices to web front-ends. J. Internet Serv. Inf. Secur., v. 10, n. 2, p. 49–66, 2020. Citado 3 vezes nas páginas 30, 32 e 39.

PELTONEN, S.; MEZZALIRA, L.; TAIBI, D. Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review. Inf. Softw. Technol., Butterworth-Heinemann, USA, v. 136, n. C, ago. 2021. ISSN 0950-5849. Disponível em: <https://doi.org/10.1016/j.infsof.2021.106571>. Citado 4 vezes nas páginas 42, 43, 44 e 45.

PWA, V. Vite PWA documentation. 2024. Vite PWA documentation. Disponível em: <https://vite-pwa-org.netlify.app/guide/>. Acesso em: 12 nov 2024. Citado na página 37.

RICHARDS, M.; FORD, N. Fundamentals of Software Architecture: An Engineering Approach. [S.l.]: O'Reilly Media, 2020. Citado 3 vezes nas páginas 29, 30 e 31.

SANTOS, V. B. dos. Desenvolvimento de software para plataforma de internet das coisas aplicada à pecuária de precisão. Dissertação (Mestrado) — Universidade Federal de Mato Grosso do Sul - UFMS, mai 2018. Citado na página 51.

SCHäFFER, E. et al. Microservice-based architecture for engineering tools enabling a collaborative multi-user configuration of robot-based automation solutions. Procedia CIRP, v. 86, p. 86–91, 2019. ISSN 2212-8271. 7th CIRP Global Web Conference – Towards shifted production value stream patterns

through inference of data, models, and technology (CIRPe 2019). Disponível em: <https://www.sciencedirect.com/science/article/pii/S2212827120300172>. Citado 3 vezes nas páginas 42, 44 e 45.

SHAKIL, M.; ZOITL, A. Towards a modular architecture for industrial hmis. In: 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). [S.l.: s.n.], 2020. v. 1, p. 1267–1270. Citado 4 vezes nas páginas 42, 43, 44 e 45.

SHAMSHIRI, R. et al. Research and development in agricultural robotics: A perspective of digital farming. International Journal of Agricultural and Biological Engineering, v. 11, p. 1–14, 07 2018. Citado na página 19.

SHAN, T. C.; HUA, W. W. Solution architecture for n-tier applications. In: IEEE. 2006 IEEE International Conference on Services Computing (SCC'06). [S.l.], 2006. p. 349–356. Citado na página 31.

SILVA, L. B. da. Modelagem e Persistência de Dados Sensoriais na Plataforma e-Cattle. Dissertação (Mestrado) — Universidade Federal de Mato Grosso do Sul - UFMS, set 2018. Citado 2 vezes nas páginas 50 e 53.

SIMÕES, B. et al. Implementing digital twins via micro-frontends, micro-services, and web 3d. Computers & Graphics, Elsevier, p. 103946, 2024. Citado 3 vezes nas páginas 41, 42 e 44.

TAIBI, D.; MEZZALIRA, L. Micro-frontends: Principles, implementations, and pitfalls. SIGSOFT Softw. Eng. Notes, Association for Computing Machinery, New York, NY, USA, v. 47, n. 4, p. 25–29, set. 2022. ISSN 0163-5948. Disponível em: <https://doi.org/10.1145/3561846.3561853>. Citado 4 vezes nas páginas 42, 43, 44 e 45.

VELEPUCHA, V.; FLORES, P. A survey on microservices architecture: Principles, patterns and migration challenges. IEEE access, IEEE, v. 11, p. 88339–88358, 2023. Citado na página 31.

VITE. Vite Documentation. 2024. Vite Documentation. Disponível em: <https://v2.vitejs.dev/guide/why.html>. Acesso em: 11 nov 2024. Citado na página 33.

VUE.JS. Vue.js Documentation. 2024. Vue.js Documentation. Disponível em: <https://pt.vuejs.org/guide/introduction.html>. Acesso em: 12 nov 2024. Citado na página 35.

VUETIFY. Vuetify Documentation. 2024. Vuetify Documentation. Disponível em: <https://vuetifyjs.com/en/introduction/why-vuetify/>. Acesso em: 12 nov 2024. Citado na página 36.

WANG, D. et al. A novel application of educational management information system based on micro frontends. Procedia Computer Science, v. 176, p. 1567–1576, 2020. ISSN 1877-0509. Knowledge-Based and Intelligent Information Engineering Systems: Proceedings of the 24th International Conference KES2020. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877050920320688>. Citado 3 vezes nas páginas 42, 44 e 45.

WOLFF, E. Microservices: flexible software architecture. [S.l.]: Addison-Wesley Professional, 2016. Citado 2 vezes nas páginas 30 e 31.