UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ISMAEL LIMA ROCHA

DESENVOLVIMENTO DE UMA PLATAFORMA WEB PARA MONITORAMENTO DE DISPOSITIVOS ELÉTRICOS DE BAIXA POTÊNCIA

FORTALEZA

2026

ISMAEL LIMA ROCHA

DESENVOLVIMENTO DE UMA PLATAFORMA WEB PARA MONITORAMENTO DE DISPOSITIVOS ELÉTRICOS DE BAIXA POTÊNCIA

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

Orientador: Prof. Dr. Raphael Amaral da Câmara

FORTALEZA

2026

ISMAEL LIMA ROCHA

DESENVOLVIMENTO DE UMA PLATAFORMA WEB PARA MONITORAMENTO DE DISPOSITIVOS ELÉTRICOS DE BAIXA POTÊNCIA

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

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Raphael Amaral da Câmara (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Raimundo Furtado Sampaio
Universidade Federal do Ceará (UFC)

Eng. Emanuel de Araújo Mota
Universidade Federal do Ceará (UFC)

À minha mãe, que sempre me deu suporte e acolhimento. À minhas irmãs, por acreditar no meu potencial. E a meu pai, que foi minha inspiração primordial para me tornar engenheiro eletricista.

AGRADECIMENTOS

Aos meus pais, Ivansildo e Marleiba, por serem a maior fonte de suporte para a minha formação acadêmica.

Às minhas irmãs, Melissa, Jéssica, Yara, e meu cunhado, Vandson, por sempre acreditarem no meu potencial e me darem incentivo para seguir em frente.

Aos meus amigos, por tornarem a jornada acadêmica mais leve, e por sempre estarem dispostos a ajudar.

Por fim, agradeço a todos os professores que fizeram parte da minha formação acadêmica, por compartilharem seus conhecimentos e experiências, e por me inspirarem a buscar sempre mais. Especialmente, agradeço ao meu orientador, Prof. Dr. Raphael Amaral da Câmara, por sua orientação e apoio durante a realização deste trabalho.

RESUMO

A crescente digitalização dos sistemas elétricos, impulsionada pela convergência entre dispositivos conectados à Internet e ferramentas *web* de visualização e análise de dados, tem ampliado a demanda por soluções acessíveis de monitoramento remoto. Nesse contexto, este trabalho tem como objetivo o desenvolvimento da plataforma SEMS, abreviação de *Smart Equipment Monitoring System* (Sistema Inteligente de Monitoramento de Equipamentos), concebida no âmbito desta pesquisa como uma solução baseada em IoT, *Internet of Things* (Internet das Coisas), para o monitoramento remoto de corrente elétrica em dispositivos de baixa potência. A arquitetura do sistema é estruturada em três camadas: um dispositivo embarcado, composto por um microcontrolador ESP32 e um sensor de corrente não invasivo SCT-013-100, responsável pela aquisição dos sinais e transmissão ao servidor via protocolo WebSocket; um servidor *backend* desenvolvido com o *framework* NestJS, sob os princípios da arquitetura hexagonal, que implementa um processamento de caminho duplo (*dual-path*) para a propagação de dados em tempo real e persistência de registros em um banco de dados PostgreSQL; e uma aplicação *web* reativa construída com React e Vite, que provê ao usuário um *dashboard* para monitoramento dinâmico e visualização de dados em tempo real e consulta a séries históricas. A plataforma incorpora mecanismos de reconexão automática e validação de dados em todas as interfaces de comunicação. Os resultados obtidos demonstram a viabilidade técnica da solução como ferramenta acessível para o monitoramento de grandezas elétricas, com potencial de aplicação em cenários de manutenção preditiva e eficiência energética.

Palavras-chave: Internet das Coisas; Monitoramento; Sistemas embarcados; Aplicação *web*; Tempo real.

ABSTRACT

The increasing digitalization of electrical systems, driven by the convergence of Internet-connected devices and web-based visualization and data analysis tools, has expanded the demand for accessible remote monitoring solutions. In this context, this work aims to develop the SEMS platform, an abbreviation of *Smart Equipment Monitoring System*, conceived within the scope of this research as an Internet of Things solution for remote electrical current monitoring in low-power devices. The system architecture is structured in three layers: an embedded device, composed of an ESP32 microcontroller and a non-invasive current sensor SCT-013-100, responsible for signal acquisition and transmission to the server via WebSocket protocol; a *backend* server developed with the NestJS framework, under the principles of hexagonal architecture, which implements dual-path processing for real-time data propagation and persistence of records in a PostgreSQL database; and a reactive web application built with React and Vite, which provides the user with a dashboard for dynamic monitoring and real-time data visualization and historical time series queries. The platform incorporates automatic reconnection and data validation mechanisms across all communication interfaces. The results obtained demonstrate the technical feasibility of the proposed solution as an accessible tool for monitoring electrical quantities, with potential application in predictive maintenance and energy efficiency scenarios.

Keywords: Internet of Things; Monitoring; Embedded systems; Web application; Real-time.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

ACID	*Atomicity, Consistency, Isolation, Durability*
ADC	*Analog-to-Digital Converter* (conversor analógico-digital)
API	*Application Programming Interface* (interface de programação de aplicações)
CORS	*Cross-Origin Resource Sharing*
CSS	*Cascading Style Sheets*
DC	*Direct Current* (corrente contínua)
DOM	*Document Object Model*
ESM	*ECMAScript Modules*
GPIO	*General Purpose Input/Output* (entrada/saída de propósito geral)
HMR	*Hot Module Replacement*
HTTP	*HyperText Transfer Protocol*
IIR	*Infinite Impulse Response* (resposta ao impulso infinita)
IoT	Internet das Coisas — *Internet of Things*
IP	*Internet Protocol*
JSON	*JavaScript Object Notation*
ORM	*Object-Relational Mapping* (mapeamento objeto-relacional)
REST	*Representational State Transfer*
RFC	*Request for Comments*
RMS	*Root Mean Square* (valor eficaz)
SEMS	*Smart Equipment Monitoring System* (Sistema Inteligente de Monitoramento de Equipamentos)
SGBD	sistema gerenciador de banco de dados
SoC	*System on Chip* (sistema em chip)
SPA	*Single-Page Application* (aplicação de página única)
SQL	*Structured Query Language*
SRAM	*Static Random-Access Memory*
SSE	*Server-Sent Events*
TC	transformador de corrente
TCP	*Transmission Control Protocol*
WSL	*Windows Subsystem for Linux*

LISTA DE SÍMBOLOS

b	Valor de *bias* DC do circuito de condicionamento, em contagens brutas do ADC
I_{RMS}	Valor eficaz (RMS) da corrente no condutor primário, em amperes
I_{inst}	Valor instantâneo da corrente, em amperes
K_{cal}	Fator de calibração do sensor de corrente
M	Número de amostras coletadas na janela de medição
n	Contagem bruta (digital) do conversor analógico-digital
n_i	i-ésima amostra bruta coletada pelo ADC na janela de medição
N	Número de níveis de quantização do ADC ($2^{12} = 4096$)
N_{TC}	Relação de espiras do transformador de corrente
R_{burden}	Resistência de carga (*burden resistor*) do transformador de corrente, em ohms
v_{AC_i}	Tensão da componente alternada da i-ésima amostra, em volts
V_{ADC}	Tensão correspondente à leitura digital do ADC, em volts
V_{ref}	Tensão de referência do conversor analógico-digital, em volts
V_{RMS}	Valor eficaz da tensão sobre o resistor de carga, em volts

SUMÁRIO

1 INTRODUÇÃO

O crescimento acelerado da Internet das Coisas — *Internet of Things* (IoT) tem transformado a maneira como sistemas físicos são monitorados e gerenciados (CHOUDHARY, 2024). Estimativas indicam que o número de dispositivos IoT conectados à internet ultrapassará 29 bilhões até 2030, abrangendo desde aplicações domésticas até ambientes industriais complexos (STATISTA, 2024). Nesse cenário, a capacidade de coletar grandezas elétricas em tempo real e disponibilizá-las por meio de interfaces *web* acessíveis representa uma oportunidade relevante para a manutenção preditiva e a eficiência energética.

Além disso, o crescente uso de microcontroladores de baixo custo com conectividade sem fio integrada, como o ESP32 da Espressif Systems, e a maturidade de *frameworks web* de código aberto têm viabilizado o desenvolvimento de soluções acessíveis e funcionais (ESPRESSIF SYSTEMS, 2025).

Apesar da disponibilidade dessas tecnologias, a integração das diferentes camadas necessárias a um sistema de monitoramento completo, como aquisição de sinais analógicos, comunicação em tempo real, persistência de dados e visualização interativa, ainda carece de soluções que conciliem escalabilidade e robustez de arquitetura. Observa-se que muitas propostas concentram-se apenas na etapa de aquisição dos dados ou recorrem a interfaces prontas, com baixo nível de customização, o que limita sua adaptação a diferentes contextos de uso. Em outros casos, embora funcionais, as soluções empregadas apoiam-se em tecnologias pouco alinhadas às demandas atuais de escalabilidade, manutenção e flexibilidade, evidenciando a necessidade de plataformas *web* mais atualizadas (DOMÍNGUEZ-BOLAÑO *et al.*, 2022).

A escolha por tecnologias de código aberto e amplamente documentadas (META PLATFORMS, 2024; MYSLIWIEC, 2024; THE POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2024; MICROSOFT, 2024) visa garantir que a solução proposta seja facilmente replicável em diferentes contextos, desde ambientes acadêmicos até instalações de pequeno porte. Além disso, a arquitetura distribuída adotada permite que cada camada seja desenvolvida, testada e substituída de forma independente, favorecendo a manutenção e a extensibilidade do sistema.

1.1 Objetivos

O objetivo geral deste trabalho consiste em apresentar o desenvolvimento de uma plataforma *web* IoT de código aberto para aquisição, análise e visualização de grandezas elétricas

de dispositivos de baixa potência, integrando dispositivo embarcado, servidor *backend* e interface gráfica.

1.1.1 Objetivos Específicos

Os objetivos específicos deste trabalho são:

a) projetar e implementar o circuito de condicionamento de sinal e o *firmware* de aquisição de corrente elétrica utilizando o microcontrolador ESP32 e o transformador de corrente não invasivo SCT-013-100;
b) desenvolver o servidor *backend* em NestJS com *gateway* WebSocket para recepção dos dados em tempo real, agregação temporal por janelas e persistência em banco de dados PostgreSQL;
c) desenvolver a interface *web* para visualização em tempo real e acesso a séries históricas;
d) validar a plataforma por meio de ensaios com um eletrodoméstico de baixa potência.

1.2 Estrutura do trabalho

Este trabalho está dividido em cinco seções, organizados da seguinte forma:

a) a Seção 1 introduz o tema, apresenta a justificativa, os objetivos e a organização do trabalho;
b) a Seção 2 aborda os fundamentos teóricos que sustentam o desenvolvimento da plataforma, incluindo conceitos de IoT, sistemas embarcados, medição de grandezas elétricas, protocolos de comunicação em tempo real e tecnologias de desenvolvimento *web*;
c) a Seção 3 descreve a arquitetura do sistema proposto e detalha a implementação de cada uma de suas camadas: dispositivo embarcado, servidor *backend* e interface *frontend*;
d) a Seção 4 apresenta os resultados obtidos nos ensaios de validação da plataforma e discute o desempenho do sistema;
e) a Seção 5 sintetiza as conclusões obtidas a partir dos resultados, aponta suas limitações e sugere direções para pesquisas futuras.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os fundamentos teóricos e conceituais que sustentam o desenvolvimento da plataforma SEMS. Inicialmente, são discutidos os princípios de Internet das Coisas e o papel dos sistemas embarcados em aplicações de monitoramento. Em seguida, abordam-se conceitos de medição de grandezas elétricas, com ênfase nos transformadores de corrente e na conversão analógico-digital. A comunicação em tempo real é realizada por meio dos protocolos WebSocket e *Server-Sent Events*. Por fim, são revisados os fundamentos das tecnologias de desenvolvimento *web* empregadas na camada de servidor e na interface do usuário.

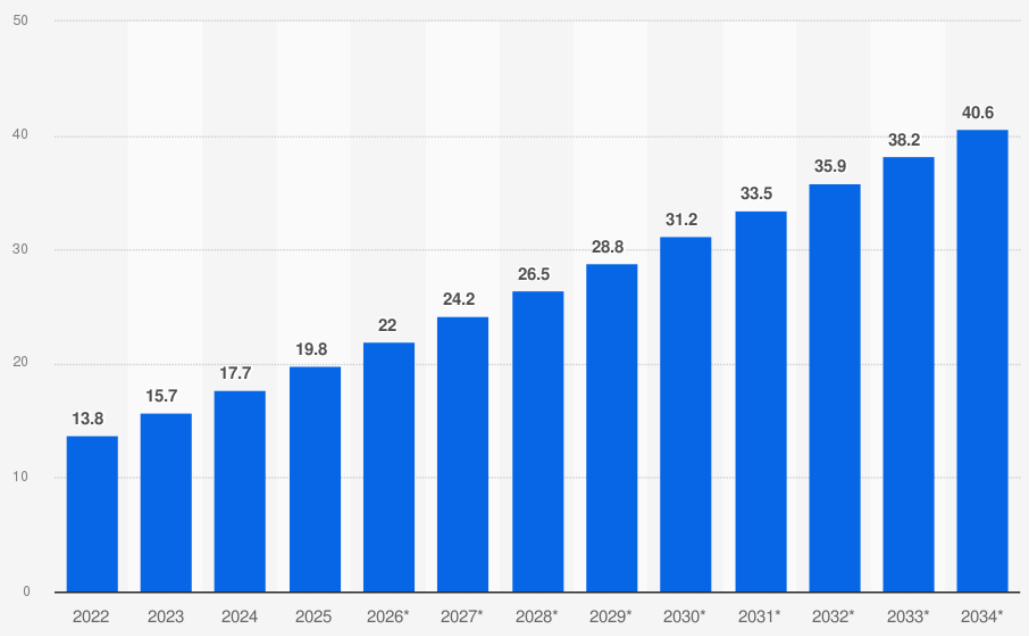
2.1 Internet das Coisas

O paradigma da IoT fundamenta-se na interconexão de objetos físicos à internet, permitindo que dispositivos dotados de sensores, atuadores e capacidade de processamento troquem dados entre si e com sistemas remotos sem intervenção humana direta.

2.1.1 Histórico

O termo foi popularizado por Kevin Ashton no final da década de 1990, ao propor que computadores pudessem coletar informações do mundo físico de forma autônoma por meio de identificadores e sensores (ASHTON, 2009). Desde então, o conceito expandiu-se para englobar uma ampla variedade de domínios, incluindo automação residencial, saúde, transporte, agricultura, monitoramento industrial e redes elétricas inteligentes (CHOUDHARY, 2024; ALI; VODAPALLY, 2026). O crescimento acelerado do número de dispositivos conectados, que é estimado em mais de 29 bilhões até 2030 (STATISTA, 2024), evidencia a relevância desse paradigma como infraestrutura tecnológica para aplicações de aquisição e análise de dados em tempo real. O gráfico representado pela Figura 1 apresenta a evolução do número de conexões IoT no mundo entre 2022 e 2023, bem como as projeções até 2034.

Figura 1 – Número de conexões de Internet das Coisas (IoT) no mundo de 2022 a 2023, com projeções de 2024 a 2034 (em bilhões).



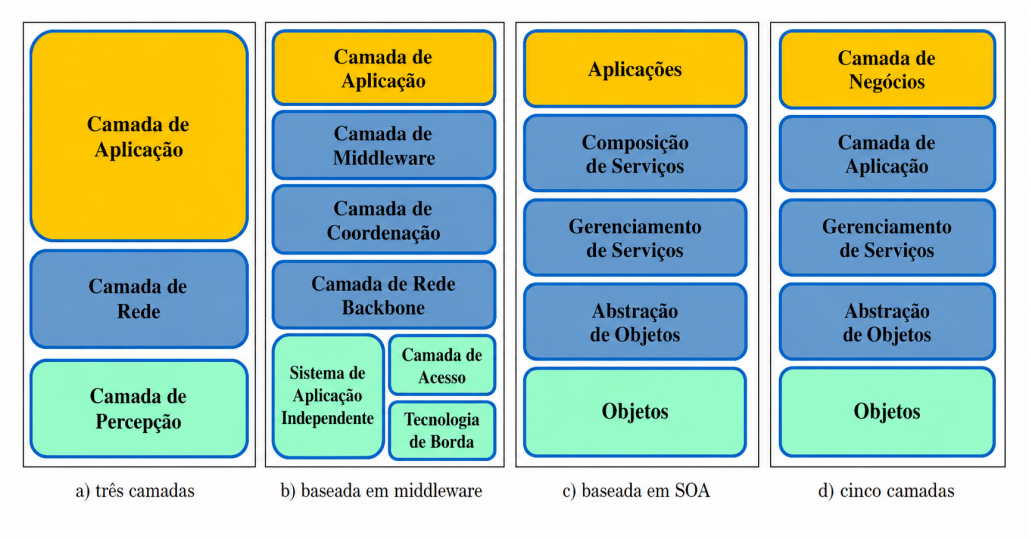
Fonte: STATISTA (2024).

2.1.2 Conceito e arquitetura de IoT

A IoT pode ser definida como uma rede de objetos físicos equipados com tecnologias de identificação, sensoriamento, comunicação e processamento, capazes de interagir com o ambiente e com outros sistemas por meio da internet (ATZORI *et al.*, 2010). Segundo AL-FUQAHA *et al.* (2015), a arquitetura de referência de um sistema IoT é frequentemente descrita em três camadas: (i) a camada de percepção, composta por sensores e atuadores que coletam dados do meio físico; (ii) a camada de rede, responsável pela transmissão dos dados por meio de protocolos de comunicação com ou sem fio; e (iii) a camada de aplicação, que processa, armazena e apresenta as informações ao usuário final.

A Figura 2 ilustra diferentes modelos de arquitetura propostos na literatura para sistemas IoT, desde a abordagem clássica de três camadas até variações com cinco camadas, incluindo modelos baseados em *middleware* e em arquitetura orientada a serviços (*SOA*).

Figura 2 – Modelos arquiteturais de referência para sistemas IoT



Fonte: Adaptado de AL-FUQAHA *et al.* (2015).

Essa organização em camadas permite que cada componente do sistema seja desenvolvido e evoluído de forma relativamente independente. Na camada de percepção, microcontroladores equipados com conversores analógico-digitais e interfaces de comunicação sem fio têm se consolidado como a solução predominante em aplicações de monitoramento de baixo custo (AL-FUQAHA *et al.*, 2015). A camada de rede pode empregar desde protocolos leves como *HyperText Transfer Protocol* (HTTP) e WebSocket até soluções específicas para IoT, dependendo dos requisitos de largura de banda e latência. A camada de aplicação, por sua vez, abrange desde o armazenamento em bancos de dados até interfaces de visualização interativas.

Nesse contexto, os sistemas embarcados desempenham papel central, pois constituem os elementos responsáveis pela interação direta com o ambiente físico. Um sistema embarcado pode ser definido como um sistema computacional de propósito específico, projetado para executar um conjunto limitado de funções dentro de um sistema maior (LEE; SESHIA, 2017). Na plataforma SEMS, o microcontrolador ESP32 atua como sistema embarcado responsável pela aquisição das grandezas elétricas e pelo envio dos dados ao servidor, ilustrando a integração típica entre o mundo físico e as camadas de software característica de soluções IoT.

A plataforma SEMS adere a essa arquitetura de três camadas: a camada de percepção é composta pelo microcontrolador ESP32 e pelo sensor de corrente SCT-013-100; a camada de rede utiliza Wi-Fi e o protocolo WebSocket para a transmissão dos dados ao servidor; e a camada de aplicação compreende o *backend* NestJS, responsável pelo processamento e persistência, e o *frontend* React, que disponibiliza a interface de visualização em tempo real.

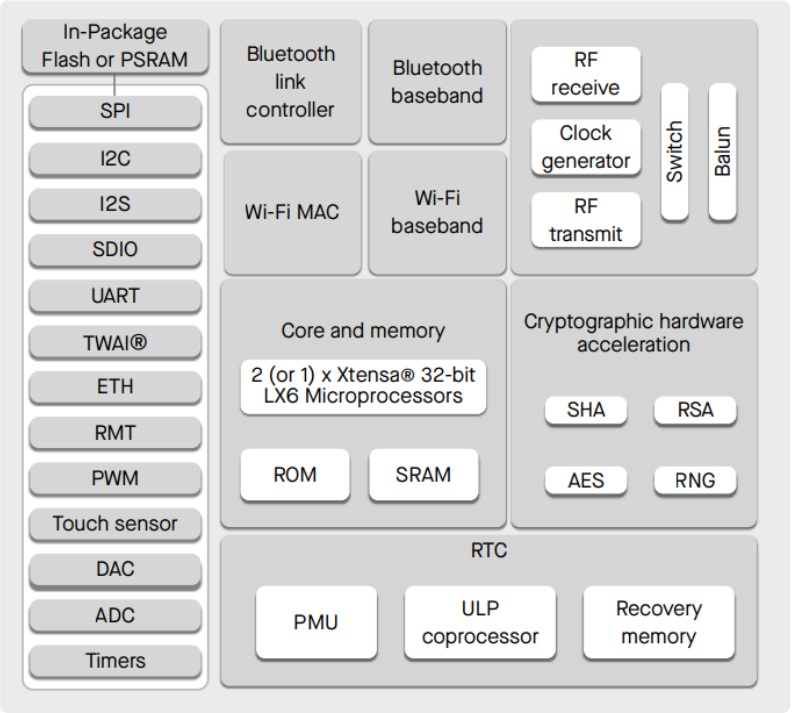
2.1.3 Microcontroladores e ESP32

Um microcontrolador é um circuito integrado que reúne, em um único *chip*, um processador, memória (volátil e não volátil) e periféricos de entrada e saída, sendo projetado para executar tarefas específicas de controle e aquisição de dados (LEE; SESHIA, 2017). Diferentemente dos microprocessadores de propósito geral, os quais dependem de componentes externos para formar um sistema funcional, os microcontroladores são autossuficientes, o que os torna adequados para aplicações embarcadas com restrições de espaço, custo e consumo energético.

O conceito de *System on Chip* (sistema em chip) (SoC) estende essa integração ao incorporar, além dos elementos tradicionais de um microcontrolador, módulos de comunicação sem fio, aceleradores de criptografia e outros periféricos avançados em um único substrato de silício. O ESP32, fabricado pela Espressif Systems, é um SoC que exemplifica essa abordagem: possui um processador *dual-core* Xtensa LX6 operando a até 240 MHz, 520 KB de *Static Random-Access Memory* (SRAM), conectividade Wi-Fi 802.11 b/g/n e Bluetooth 4.2 integradas, além de um *Analog-to-Digital Converter* (conversor analógico-digital) (ADC) de 12 bits com múltiplos canais (ESPRESSIF SYSTEMS, 2025). Essas características, associadas a um custo unitário tipicamente inferior a dez dólares, posicionam o ESP32 como uma das plataformas mais utilizadas em projetos de IoT.

A Figura 3 sintetiza essa integração ao apresentar, no mesmo *chip*, os blocos de processamento e memória, os módulos de comunicação Wi-Fi e Bluetooth e os principais periféricos de entrada e saída, incluindo o ADC.

Figura 3 – Diagrama funcional de blocos do ESP32



Fonte: ESPRESSIF SYSTEMS (2025).

Em comparação com alternativas comumente empregadas em projetos acadêmicos e de prototipagem, como o Arduino Uno (baseado no ATmega328P, sem conectividade sem fio nativa) ou o Raspberry Pi Pico (baseado no RP2040, também sem Wi-Fi na versão original), o ESP32 oferece a vantagem de dispensar módulos de comunicação externos, reduzindo a complexidade do circuito e o custo total do projeto (ESPRESSIF SYSTEMS, 2025). Essa característica foi determinante para a sua adoção na plataforma SEMS, na qual o microcontrolador é responsável tanto pela aquisição do sinal de corrente por meio do ADC integrado quanto pela transmissão dos dados ao servidor via Wi-Fi.

2.1.4 Aplicações de IoT no monitoramento de equipamentos elétricos

A aplicação de tecnologias de IoT ao monitoramento de grandezas elétricas tem se consolidado como uma abordagem eficaz para a gestão energética e a manutenção preditiva. No contexto de medição inteligente (*smart metering*), dispositivos conectados permitem o acompanhamento remoto e em tempo real do consumo de energia elétrica em residências, edifícios comerciais e instalações industriais, fornecendo dados que subsidiam a identificação de desperdícios e a otimização de padrões de uso (DEPURU *et al.*, 2011). A disponibilidade contínua dessas informações possibilita não apenas a redução de custos operacionais, mas

também a detecção precoce de anomalias no perfil de consumo, que podem indicar falhas em equipamentos ou condições de operação inadequadas.

No âmbito da manutenção preditiva, a análise de grandezas elétricas, particularmente a corrente consumida por motores e cargas, constitui uma técnica reconhecida para o diagnóstico de condições de operação. A análise de assinatura de corrente do motor (*Motor Current Signature Analysis*, MCSA) é uma abordagem amplamente documentada na literatura, na qual variações no espectro de frequência da corrente de estator são correlacionadas a defeitos mecânicos e elétricos, como barras de rotor quebradas, desalinhamento e desgaste de rolamentos (NANDI *et al.*, 2005). Embora a plataforma SEMS não implemente análise espectral em sua versão atual, a aquisição contínua de dados de corrente *Root Mean Square* (valor eficaz) (RMS) constitui o ponto de partida para a incorporação futura dessas técnicas de diagnóstico.

Diversos trabalhos na literatura propõem sistemas de monitoramento elétrico baseados em microcontroladores de baixo custo com conectividade sem fio. KABALCI (2016) apresentam uma revisão abrangente de arquiteturas de medição inteligente que utilizam tecnologias de comunicação sem fio para a coleta e transmissão de dados de consumo energético. Esses sistemas compartilham com a plataforma SEMS a premissa de oferecer soluções acessíveis e replicáveis, embora difiram em aspectos como o protocolo de comunicação adotado, o tipo de sensor empregado e a presença ou ausência de uma interface de visualização em tempo real.

2.2 Medição de grandezas elétricas

A medição de grandezas elétricas, como corrente, tensão e potência, é essencial para o monitoramento, a proteção e o diagnóstico de sistemas de energia elétrica. Em aplicações industriais e prediais, o conhecimento preciso dessas grandezas permite identificar sobrecargas, desequilíbrios entre fases e degradação progressiva de equipamentos (WEBSTER; EREN, 2014). Na plataforma SEMS, a corrente elétrica foi adotada como grandeza primária de monitoramento, pois sua medição pode ser realizada de forma não invasiva sem a necessidade de interromper o circuito monitorado, por meio de transformador de correntes (TCs), o que simplifica a instalação e reduz o custo do dispositivo de aquisição.

2.2.1 Transformadores de corrente

O TC é um instrumento de medição que opera segundo o princípio da indução eletromagnética: a corrente que percorre o condutor primário gera um campo magnético no núcleo ferromagnético, o qual induz uma corrente proporcional no enrolamento secundário. A relação entre as correntes primária (I_p) e secundária (I_s) é determinada pela relação de espiras do transformador, conforme a Equação (1) (INTERNATIONAL ELECTROTECHNICAL COMMISSION, 2012):

$$I_s = \frac{I_p}{N_{\mathrm{TC}}} \tag{1}$$

em que N_{TC} é a relação de espiras entre o primário e o secundário. No caso de TCs do tipo núcleo bipartido (*split-core*), o condutor monitorado atravessa a abertura do núcleo sem necessidade de desconexão, característica que confere ao dispositivo a natureza não invasiva. A tensão disponível nos terminais do secundário depende da impedância de carga conectada, denominada *burden*, de modo que a inserção de um resistor de carga R_b converte a corrente secundária em uma tensão mensurável:

$$V_b = I_s \cdot R_b \tag{2}$$

O modelo SCT-013-100, utilizado na plataforma SEMS, é um TC de núcleo bipartido com relação de espiras de 2000:1, projetado para medição de correntes alternadas de até 100 A no condutor primário. Diferentemente de variantes que incorporam um resistor de carga interno e fornecem saída em tensão, o SCT-013-100 possui saída em corrente, exigindo a instalação de um *burden resistor* externo para adequar o nível de sinal à faixa de entrada do ADC do microcontrolador (OPEN ENERGY MONITOR, 2016).

A Figura 4 demonstra as características físicas do sensor SCT-013-100, o qual está representado com seu conector de saída e o núcleo bipartido que permite a instalação sem interrupção do circuito monitorado.

Figura 4 – Sensor de Corrente Não Invasivo 100A SCT-013-100

Fonte: RoboCore (2026).

2.2.2 Conversão analógico-digital

O conversor analógico-digital (ADC) é o componente responsável por transformar um sinal de tensão contínuo em uma representação numérica discreta, viabilizando o processamento do sinal por um sistema digital. Dois parâmetros fundamentais definem o desempenho de um ADC: a resolução, expressa em bits, que determina o número de níveis de quantização disponíveis (2^b níveis para b bits); e a taxa de amostragem, que especifica quantas conversões são realizadas por segundo. A tensão correspondente a uma contagem digital n é dada pela Equação (3) (KESTER, 2005):

$$V = n \cdot \frac{V_{\text{ref}}}{2^b} \tag{3}$$

em que V_{ref} é a tensão de referência do conversor e b é a resolução em bits.

O ADC integrado ao ESP32 emprega a arquitetura de aproximações sucessivas (*Successive Approximation Register*, SAR), com resolução de 12 bits (4096 níveis) e tensão de referência de 3,3 V. Embora adequado para aplicações de monitoramento de baixo custo, esse conversor apresenta limitações documentadas de não linearidade, especialmente nos extremos da faixa de entrada (ESPRESSIF SYSTEMS, 2025). Além disso, como o ADC do ESP32 aceita

apenas tensões positivas (0 a 3,3 V), a medição de sinais alternados, que oscilam em torno de zero, requer um circuito de polarização (*bias Direct Current* (corrente contínua) (DC)) que desloque o sinal para o ponto médio da faixa de entrada, tipicamente 1,65 V.

2.2.3 Valor eficaz (RMS) de sinais alternados

O valor eficaz, designado pela sigla RMS (*Root Mean Square*), é a medida padrão para quantificar a magnitude de sinais alternados em engenharia elétrica. Fisicamente, o valor RMS de uma corrente alternada corresponde ao valor de corrente contínua que produziria a mesma dissipação de potência em uma carga resistiva. Para um sinal contínuo $x(t)$ de período T, o valor eficaz X_{RMS} é definido pela Equação (4) (NILSSON; RIEDEL, 2014):

$$X_{\mathrm{RMS}} = \sqrt{\frac{1}{T}\int_0^T \left[x(t)\right]^2 dt} \tag{4}$$

Na forma discreta, aplicável ao processamento digital de sinais amostrados por um ADC, a integral é substituída por um somatório sobre M amostras coletadas em uma janela de medição:

$$X_{\mathrm{RMS}} = \sqrt{\frac{1}{M}\sum_{i=1}^{M} x_i^2} \tag{5}$$

Para que a estimativa do valor RMS seja estável, a janela de amostragem deve compreender um número inteiro, ou suficientemente grande, de ciclos do sinal. No caso da rede elétrica brasileira, cuja frequência fundamental é de 60 Hz, uma janela de 100 ms abrange aproximadamente seis ciclos completos, intervalo considerado adequado para obter uma estimativa com baixa variância entre janelas consecutivas (WEBSTER; EREN, 2014).

2.3 Protocolos de comunicação em tempo real

Em aplicações IoT com interface *web*, a entrega de dados em tempo real pode ser implementada por diferentes estratégias. A abordagem mais elementar é o *polling*, na qual o cliente envia requisições HTTP periódicas ao servidor para verificar a existência de novos dados; embora simples, essa técnica introduz latência proporcional ao intervalo de consulta e gera tráfego desnecessário quando não há atualizações. Uma evolução é o *long-polling*, em que

o servidor mantém a conexão aberta até que haja dados disponíveis, reduzindo a latência à custa de maior uso de recursos no servidor. Para superar essas limitações, dois mecanismos foram padronizados: o protocolo WebSocket, que estabelece um canal bidirecional persistente sobre *Transmission Control Protocol* (TCP) (FETTE; MELNIKOV, 2011), e o *Server-Sent Events* (SSE), que permite ao servidor enviar eventos de forma unidirecional sobre uma conexão HTTP de longa duração (WHATWG, 2024).

A plataforma SEMS adota ambos os mecanismos de forma complementar: o WebSocket é utilizado no enlace entre o dispositivo embarcado e o servidor, onde a comunicação bidirecional é necessária; o SSE é empregado entre o servidor e o navegador do usuário, onde o fluxo de dados é exclusivamente do servidor para o cliente. As subseções seguintes detalham cada protocolo, bem como o formato de serialização adotado para a troca de mensagens.

2.3.1 Protocolo WebSocket

O protocolo WebSocket, padronizado pela *Request for Comments* (RFC) 6455, estabelece um canal de comunicação bidirecional e *full-duplex* sobre uma única conexão TCP. A conexão é iniciada por meio de um *handshake* HTTP, no qual o cliente envia um cabeçalho `Upgrade: websocket`; uma vez que o servidor aceita a solicitação, o protocolo é promovido e ambas as partes podem transmitir *frames* de texto ou binários de forma independente, sem a necessidade de novas requisições HTTP (FETTE; MELNIKOV, 2011). Essa característica elimina a sobrecarga (*overhead*) de cabeçalhos repetidos inerente ao modelo requisição-resposta do HTTP e reduz significativamente a latência na troca de mensagens.

Na plataforma SEMS, o protocolo WebSocket é empregado no enlace entre o microcontrolador ESP32 e o servidor NestJS. Essa escolha justifica-se pela necessidade de transmissão contínua de leituras elétricas com baixa latência: o dispositivo embarcado envia periodicamente pacotes com os valores de corrente, tensão e potência calculados, e o servidor pode, pelo mesmo canal, transmitir comandos de configuração ao dispositivo, cenário que explora a bidirecionalidade nativa do protocolo (FETTE; MELNIKOV, 2011).

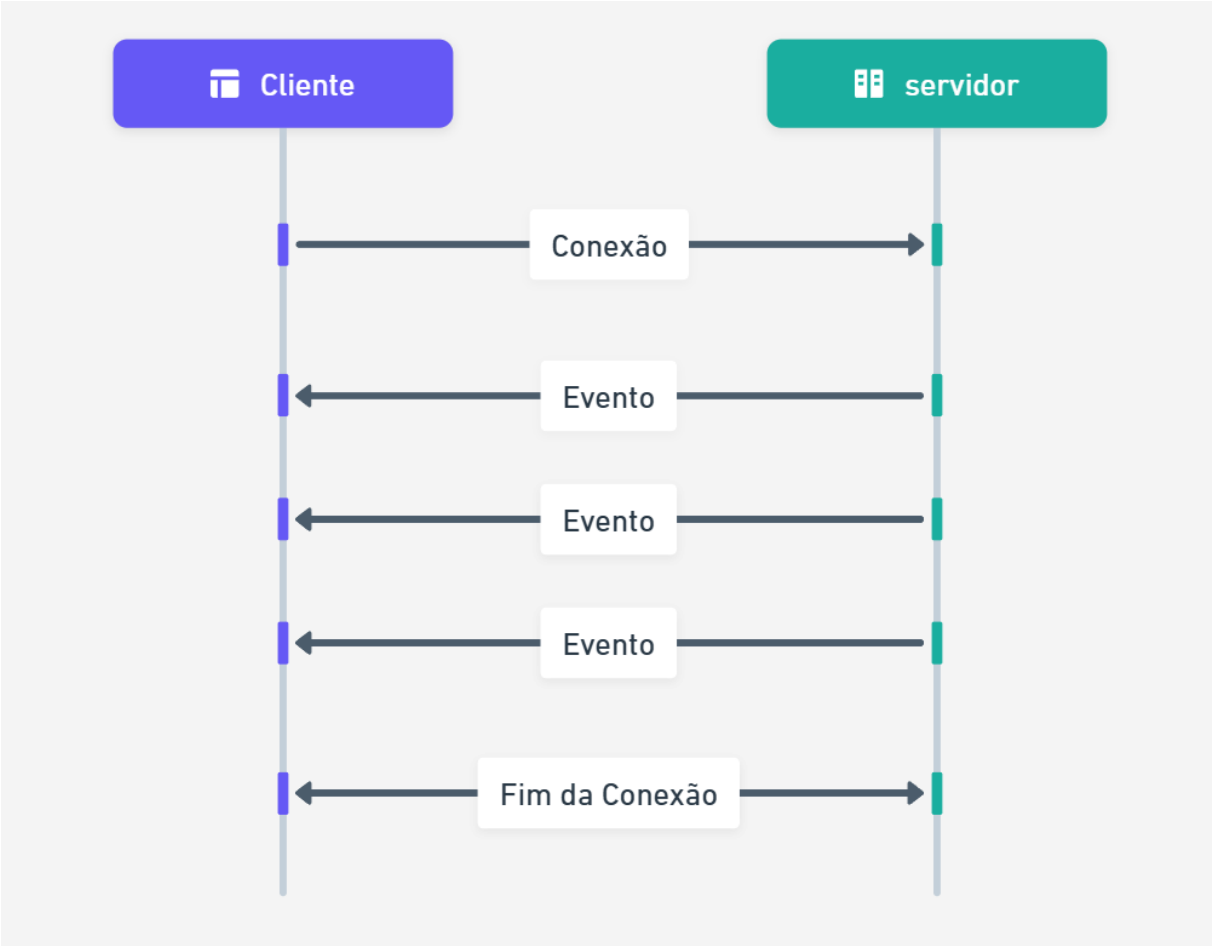
2.3.2 Server-Sent Events (SSE)

O mecanismo de SSE, definido na especificação HTML Living Standard, permite que o servidor envie eventos ao cliente de forma contínua por meio de uma conexão HTTP persistente com tipo de conteúdo `text/event-stream` (WHATWG, 2024). Do lado do nave-

gador, a *Application Programming Interface* (interface de programação de aplicações) (API) `EventSource` gerencia automaticamente a abertura da conexão, o tratamento de eventos recebidos e a reconexão em caso de falha. Diferentemente do WebSocket, o SSE opera exclusivamente no sentido servidor–cliente e não requer a promoção (*upgrade*) do protocolo HTTP, o que simplifica a implementação e a compatibilidade com *proxies* e balanceadores de carga.

Na Figura 5 é apresentado um diagrama que demonstra o fluxo de conexão e envio de dados entre o cliente e o servidor. Inicialmente, o cliente estabelece uma conexão HTTP de longa duração com o servidor. Em seguida, o servidor está apto para enviar eventos ao cliente de modo unidirecional, utilizando o formato *text/event-stream*. O cliente, por sua vez, recebe os eventos e pode processá-los conforme necessário. Por fim, a conexão pode ser interrompida tanto pelo cliente quanto pelo servidor, ou pode ser mantida aberta para o envio contínuo de eventos.

Figura 5 – Diagrama de conexão entre o servidor e o cliente utilizando Server-Sent Events (SSE).



Fonte: elaborado pelo autor (2026).

Na plataforma SEMS, o SSE é utilizado para transmitir as leituras elétricas do *backend* NestJS ao *frontend* React. Como o navegador apenas consome os dados, sem necessidade

de enviar informações ao servidor por esse canal, a unidirecionalidade do SSE é suficiente e evita a complexidade adicional de manter uma conexão WebSocket entre servidor e cliente *web* (WHATWG, 2024).

2.3.3 Formato JSON para serialização de dados

O *JavaScript Object Notation* (JSON) é um formato de intercâmbio de dados baseado em texto, cuja sintaxe utiliza pares chave-valor e estruturas ordenadas (*arrays*), conforme padronizado pela RFC 8259 (BRAY, 2017). Por ser derivado da notação literal de objetos JavaScript, o JSON possui suporte nativo em navegadores e em ambientes Node.js, dispensando bibliotecas externas de *parsing* no lado *web*. No domínio embarcado, bibliotecas como a ArduinoJson oferecem implementações eficientes para microcontroladores com recursos limitados de memória.

Na plataforma SEMS, o JSON é o formato de serialização adotado em todas as interfaces de comunicação: o ESP32 codifica as leituras elétricas em objetos JSON transmitidos via WebSocket, o *backend* processa e persiste esses objetos, e o SSE entrega os dados ao *frontend* no mesmo formato. Essa padronização reduz a complexidade de conversão entre camadas e facilita a depuração das mensagens trocadas (BRAY, 2017).

2.4 Desenvolvimento *web* moderno

A arquitetura das aplicações *web* evoluiu significativamente nas últimas duas décadas, migrando de páginas estáticas renderizadas integralmente no servidor para aplicações de página única (*Single-Page Application* (aplicação de página única) (SPA)), nas quais o navegador assume a responsabilidade pela renderização da interface e pela comunicação assíncrona com o servidor por meio de APIs *Representational State Transfer* (REST) ou mecanismos de tempo real (FLANAGAN, 2020). Essa separação entre *frontend* e *backend* permitiu que cada camada fosse desenvolvida, testada e implantada de forma independente, favorecendo a adoção de tecnologias especializadas em cada extremidade.

O ecossistema JavaScript, e sua extensão tipada TypeScript, consolidou-se como a plataforma predominante para esse modelo de desenvolvimento, pois possibilita o uso de uma mesma linguagem tanto no servidor (por meio do ambiente de execução Node.js) quanto no cliente (navegador). A plataforma SEMS adota essa abordagem: o *backend* é construído com

o *framework* NestJS, o *frontend* com a biblioteca React, e ambos compartilham TypeScript como linguagem de programação. As subseções seguintes apresentam os fundamentos de cada tecnologia empregada.

2.4.1 TypeScript

TypeScript é uma linguagem de programação de código aberto desenvolvida pela Microsoft que estende o JavaScript com um sistema de tipos estático opcional (MICROSOFT, 2024). O compilador `tsc` verifica a consistência de tipos em tempo de compilação e emite código JavaScript puro, compatível com qualquer ambiente de execução que suporte a linguagem base. Recursos como inferência de tipos, interfaces, tipos genéricos e tipos de união permitem modelar contratos entre módulos de forma explícita, o que reduz categorias inteiras de erros, como acessos a propriedades inexistentes ou passagem de argumentos com tipos incompatíveis, antes mesmo da execução do programa.

Na plataforma SEMS, o TypeScript é a linguagem única de desenvolvimento tanto no *backend* (NestJS) quanto no *frontend* (React). Essa unificação elimina a necessidade de traduzir manualmente estruturas de dados entre camadas: interfaces que descrevem o formato das leituras elétricas, por exemplo, podem ser compartilhadas entre servidor e cliente, garantindo que alterações no esquema de dados sejam detectadas pelo compilador em ambos os lados (MICROSOFT, 2024).

2.4.2 NestJS e arquitetura de servidor

NestJS é um *framework* para construção de aplicações *server-side* em Node.js que adota uma arquitetura modular inspirada no Angular (MYSLIWIEC, 2024). A organização do código é baseada em três artefatos principais: módulos, que agrupam funcionalidades coesas; controladores, que expõem *endpoints* HTTP, WebSocket ou SSE; e provedores (*providers*), que encapsulam a lógica de negócio e são injetados automaticamente pelo contêiner de injeção de dependências do *framework*. Essa estrutura favorece a separação de responsabilidades e facilita a escrita de testes unitários, uma vez que as dependências de cada componente podem ser substituídas por dublês (*mocks*) de forma transparente.

Na plataforma SEMS, o *backend* NestJS é responsável por receber as leituras elétricas via WebSocket, persistir os dados em banco de dados e disponibilizá-los ao *frontend* por meio de SSE e de APIs REST. O suporte nativo do NestJS a múltiplos protocolos de transporte,

incluindo WebSocket e SSE, o que permitiu que essas funcionalidades fossem implementadas dentro de uma mesma base de código, sem a necessidade de servidores ou bibliotecas adicionais (MYSLIWIEC, 2024).

2.4.3 React e interfaces reativas

React é uma biblioteca JavaScript de código aberto, mantida pela Meta, destinada à construção de interfaces de usuário por meio de componentes declarativos (META PLATFORMS, 2024). O mecanismo central da biblioteca é o *Virtual Document Object Model (DOM)*: a cada mudança de estado, React calcula uma representação virtual da árvore de elementos, compara-a com a versão anterior e aplica ao DOM real apenas as diferenças necessárias, minimizando operações custosas de manipulação direta. O modelo de *hooks*, como `useState` e `useEffect`, permite que componentes funcionais gerenciem estado local e efeitos colaterais de forma concisa, sem recorrer a classes.

Na plataforma SEMS, o *frontend* React é empacotado pelo Vite, uma ferramenta de *build* que utiliza módulos *ECMAScript Modules* (ESM) nativos do navegador durante o desenvolvimento, proporcionando inicialização rápida e substituição de módulos a quente (*Hot Module Replacement* (HMR)). A interface de visualização emprega componentes do sistema de design shadcn/ui, construídos sobre o Radix UI e estilizados com Tailwind *Cascading Style Sheets* (CSS), e utiliza a biblioteca Recharts para a renderização dos gráficos de grandezas elétricas em tempo real (META PLATFORMS, 2024).

2.4.4 Banco de dados relacional e ORM

Um banco de dados relacional organiza os dados em tabelas compostas por linhas e colunas, cujas relações são formalizadas por meio de chaves primárias e estrangeiras, e manipuladas pela linguagem *Structured Query Language* (SQL). O PostgreSQL é um sistema gerenciador de banco de dados (SGBD) relacional de código aberto que implementa integralmente as propriedades *Atomicity, Consistency, Isolation, Durability* (ACID) (atomicidade, consistência, isolamento e durabilidade), garantindo a integridade das transações mesmo em cenários de falha (THE POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2024). Sua maturidade, extensibilidade e suporte a tipos de dados avançados (como JSON e *arrays*) fazem dele um dos sistemas mais adotados tanto em ambientes de produção quanto em projetos acadêmicos.

Na plataforma SEMS, a interação entre o *backend* NestJS e o PostgreSQL é mediada

pelo TypeORM, uma biblioteca de *Object-Relational Mapping* (mapeamento objeto-relacional) (ORM) para TypeScript que mapeia classes decoradas, denominadas *entidades*, a tabelas do banco de dados (TYPEORM, 2024). Cada propriedade da entidade corresponde a uma coluna, e as operações de leitura e escrita são realizadas por meio de repositórios que abstraem as consultas SQL. Essa abstração permite que alterações no esquema de dados sejam versionadas por meio de migrações e que a lógica de persistência permaneça desacoplada dos detalhes específicos do SGBD.

Considerações finais

Os fundamentos teóricos apresentados neste capítulo formam a base técnica sobre a qual a plataforma SEMS foi concebida. A convergência entre a adoção de tecnologias IoT para monitoramento não invasivo de grandezas elétricas, a utilização de protocolos de comunicação modernos para a transmissão em tempo real, e a aplicação de tecnologias *web* atuais para a visualização dos dados, resulta em uma solução integrada e coesa.

Do ponto de vista de aquisição de dados, a escolha do microcontrolador ESP32 como sistema embarcado justifica-se por sua capacidade de integrar, em um único *chip*, o processamento necessário, a conectividade sem fio nativa e os periféricos de entrada analógica requeridos. A utilização do sensor SCT-013-100, combinada com técnicas de condicionamento de sinal e conversão analógico-digital, permite a medição não invasiva e de baixo custo de correntes alternadas, tornando a solução acessível e facilmente replicável em diferentes contextos.

Na perspectiva de transmissão de dados, o protocolo WebSocket viabiliza a comunicação bidirecional e de baixa latência entre o dispositivo embarcado e o servidor, enquanto o SSE oferece um mecanismo eficiente de entrega unidirecional ao navegador do usuário final. A adoção do formato JSON como padrão de serialização elimina as barreiras entre as camadas de software e simplifica o desenvolvimento e a depuração do sistema como um todo.

Por fim, a arquitetura do *backend* e *frontend* reflete as práticas contemporâneas de desenvolvimento *web*: a escolha de TypeScript garante a consistência de tipos em ambas as camadas; o NestJS oferece uma estrutura modular que favorece a separação de responsabilidades; o React viabiliza a construção de interfaces reativas e responsivas; e o PostgreSQL com TypeORM fornece persistência robusta e versionada dos dados. Essa confluência de tecnologias e conceitos fundamentais é o que permite que a plataforma SEMS cumpra seu objetivo de oferecer uma solução moderna, integrada e de código aberto para o monitoramento em tempo real de

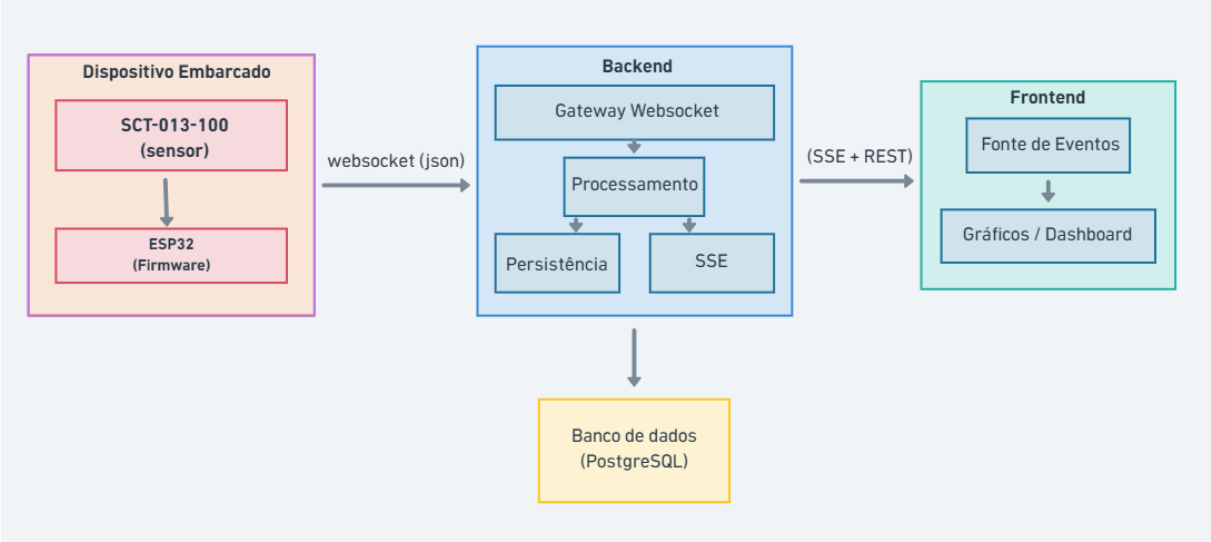
grandezas elétricas.

3 METODOLOGIA

A plataforma desenvolvida neste trabalho adota uma arquitetura distribuída em três camadas: dispositivo embarcado, servidor *backend* e interface *frontend*. Cada camada desempenha um papel específico na cadeia de aquisição, processamento e visualização dos dados elétricos, e a comunicação entre elas ocorre por meio de protocolos padronizados da *web*.

Na Figura 6 é apresentado o diagrama de blocos da arquitetura completa do sistema, em que cada bloco representa uma camada distinta da plataforma. A camada de aquisição é composta pelo dispositivo embarcado, que integra o sensor de corrente, o circuito de carga e o microcontrolador ESP32. A camada intermediária, denominada Backend, recebe os dados por meio de um *Gateway WebSocket*, processa essas informações para emissão via SSE e realiza a persistência em banco de dados. Por fim, o Frontend constitui a camada de apresentação, recebendo os dados em tempo real emitidos pelo Backend através do SSE por meio do *Event Source*, ou por requisições HTTP convencionais sob a arquitetura REST para análises históricas, exibindo-os na interface de usuário por meio de gráficos.

Figura 6 – Diagrama de blocos da arquitetura geral da plataforma SEMS



Fonte: elaborado pelo autor (2026).

A camada de aquisição é composta por um microcontrolador ESP32 acoplado a um sensor de corrente não invasivo do tipo transformador de corrente (modelo SCT-013-100). A ESP32 realiza a leitura analógica do sinal de corrente, calcula os valores eficazes (RMS) e transmite os dados ao servidor por meio de uma conexão WebSocket persistente. Os dados são encapsulados em mensagens no formato JSON e enviados a uma taxa aproximada de dez amostras por segundo.

A camada de servidor (*backend*) foi implementada com o *framework* NestJS, utilizando TypeScript como linguagem de programação. O servidor recebe os dados em tempo real por meio de um *gateway* WebSocket nativo e executa um processamento de caminho duplo (*dual-path*): por um lado, cada leitura recebida é imediatamente emitida para os clientes conectados via *Server-Sent Events* (SSE), garantindo baixa latência na visualização; por outro lado, as leituras de corrente eficaz são acumuladas em um *buffer* em memória e periodicamente agregadas, por meio da média aritmética de uma janela temporal configurável, antes de serem persistidas em um banco de dados PostgreSQL. Essa estratégia de agregação reduz significativamente a carga de escrita no banco, armazenando um único registro consolidado a cada intervalo de dez segundos, em vez de dez registros por segundo.

A camada de apresentação (*frontend*) consiste em uma aplicação *web* desenvolvida com a biblioteca React, utilizando Vite como ferramenta de *build* e a biblioteca de componentes shadcn/ui. A interface consome o *stream* SSE para exibir os dados de corrente em tempo real, por meio de gráficos dinâmicos e indicadores visuais, e acessa a API REST do *backend* para consulta e visualização de séries históricas com suporte a filtragem temporal e paginação.

A separação em camadas confere ao sistema características desejáveis de modularidade e extensibilidade. O dispositivo embarcado opera de forma independente do *backend*, necessitando apenas de conectividade Wi-Fi e do endereço do servidor para transmitir os dados. O servidor, por sua vez, desacopla a recepção dos dados brutos da persistência, permitindo que novos tipos de sensores ou dispositivos sejam integrados sem modificações na lógica de armazenamento ou na interface. O *frontend*, ao consumir exclusivamente as interfaces SSE e REST expostas pelo *backend*, pode ser substituído ou complementado por outras aplicações-cliente, como painéis móveis ou sistemas de alertas, sem impacto nas demais camadas.

3.1 Camada de aquisição

O dispositivo embarcado constitui a camada de aquisição da plataforma, sendo responsável por medir as grandezas elétricas do equipamento monitorado e transmiti-las ao servidor em tempo real. O módulo é construído em torno do microcontrolador ESP32, que reúne capacidade de processamento, conectividade Wi-Fi integrada e um conversor analógico-digital (ADC) de 12 bits, características que o tornam adequado para aplicações de Internet das Coisas (IoT) com restrições de custo e espaço físico. A corrente elétrica é medida de forma não invasiva por meio de um transformador de corrente (TC) do modelo SCT-013-100, cujo sinal analógico é

condicionado e convertido em valores de corrente eficaz (RMS) pelo *firmware* embarcado.

As subseções a seguir descrevem os componentes de *hardware* empregados (subseção 3.1.1), a organização do *firmware* (subseção 3.1.2), o processo de aquisição e cálculo da corrente (subseção 3.1.3) e o mecanismo de comunicação com o servidor (subseção 3.1.4).

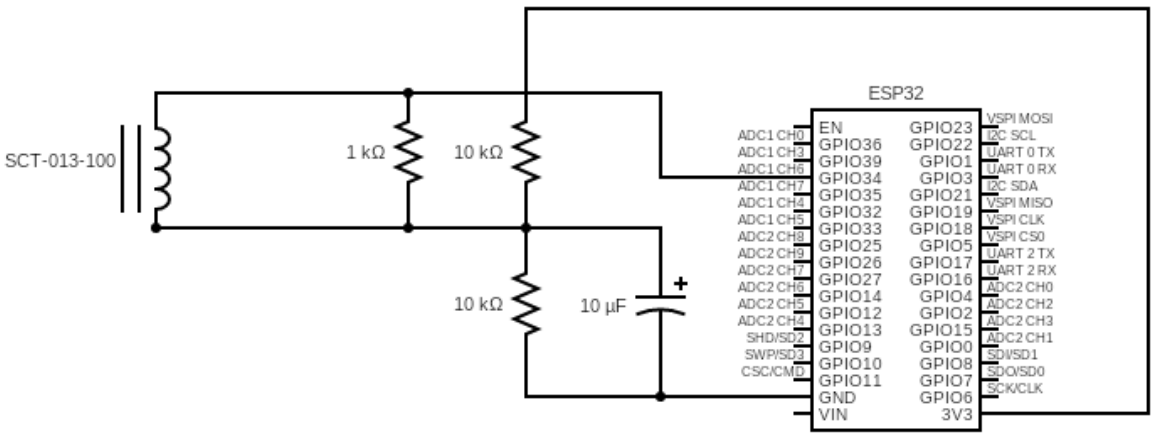
3.1.1 Hardware utilizado

O circuito do dispositivo embarcado é composto pelos seguintes elementos principais:

- Microcontrolador ESP32: SoC com conectividade Wi-Fi 802.11 b/g/n e Bluetooth 4.2. O ADC integrado possui resolução de 12 bits (4096 níveis) e faixa de entrada configurável por meio de atenuação programável. No presente trabalho, o pino *General Purpose Input/Output* (entrada/saída de propósito geral) (GPIO) 34 é utilizado para a leitura do sinal de corrente;
- Sensor de corrente SCT-013-100: transformador de corrente não invasivo com relação de espiras de 2000:1, projetado para medir correntes alternadas de até 100 A no enrolamento primário. O enrolamento secundário não possui resistor de carga (*burden resistor*) interno, de modo que a corrente secundária deve ser convertida em tensão por um resistor externo;
- Resistor de carga (*burden resistor*): resistor de 1 kΩ conectado aos terminais do secundário do SCT-013-100, responsável por converter a corrente secundária em uma tensão proporcional à corrente medida no primário;
- Circuito de polarização (*bias* DC): divisor resistivo que desloca o sinal alternado do sensor para um nível de tensão contínua de aproximadamente 1,65 V (metade da faixa de 3,3 V do ADC). Esse deslocamento é necessário porque o ADC da ESP32 aceita apenas tensões positivas, enquanto o sinal do TC oscila em torno de zero.

A Figura 7 apresenta o diagrama esquemático das conexões entre o sensor, o circuito de condicionamento e o microcontrolador. Nela, destaca-se os dois resistores de 10k Ω que formam o divisor de tensão para o *bias* DC, o capacitor de desacoplamento que estabiliza a tensão de referência, o resistor de carga de 1k Ω e a conexão do sinal condicionado ao pino GPIO 34 da ESP32 para leitura pelo ADC.

Figura 7 – Diagrama esquemático do circuito de aquisição de corrente com a ESP32 e o sensor SCT-013-100



Fonte: elaborado pelo autor (2026).

3.1.2 Firmware da ESP32

O *firmware* foi desenvolvido em linguagem C++ sobre o *framework* Arduino para ESP32, utilizando o editor de código Arduino IDE, que oferece integração facilitada com placas embarcadas e recursos de depuração. Para garantir a compatibilidade com o microcontrolador ESP32, foi instalada a extensão "ESP32 Arduino" no gerenciador de placas da IDE. Além disso, foram utilizadas bibliotecas específicas, como `WiFi.h` para conectividade sem fio, `WebSocketsClient.h` para comunicação via WebSocket, e `ArduinoJson.h` para manipulação de dados em formato JSON. O código foi estruturado de forma modular, empregando gerenciadores (*managers*) que encapsulam tanto a inicialização quanto a lógica cíclica de cada funcionalidade, promovendo organização e facilidade de manutenção. O Quadro 1 relaciona os módulos que compõem o *firmware*.

Quadro 1 – Módulos do *firmware* da ESP32 e suas responsabilidades

Módulo	Arquivos	Responsabilidade
`config.h`	`config.h`	Definições centralizadas de configuração
`wifi_manager`	`wifi_manager.{h,cpp}`	Conexão e reconexão Wi-Fi
`ws_manager`	`ws_manager.{h,cpp}`	Cliente WebSocket (conexão, envio, reconexão)
`sct_manager`	`sct_manager.{h,cpp}`	Leitura do sensor SCT-013 e cálculo RMS
`main.ino`	`main.ino`	Ponto de entrada (`setup` e `loop`)

Fonte: elaborado pelo autor (2026).

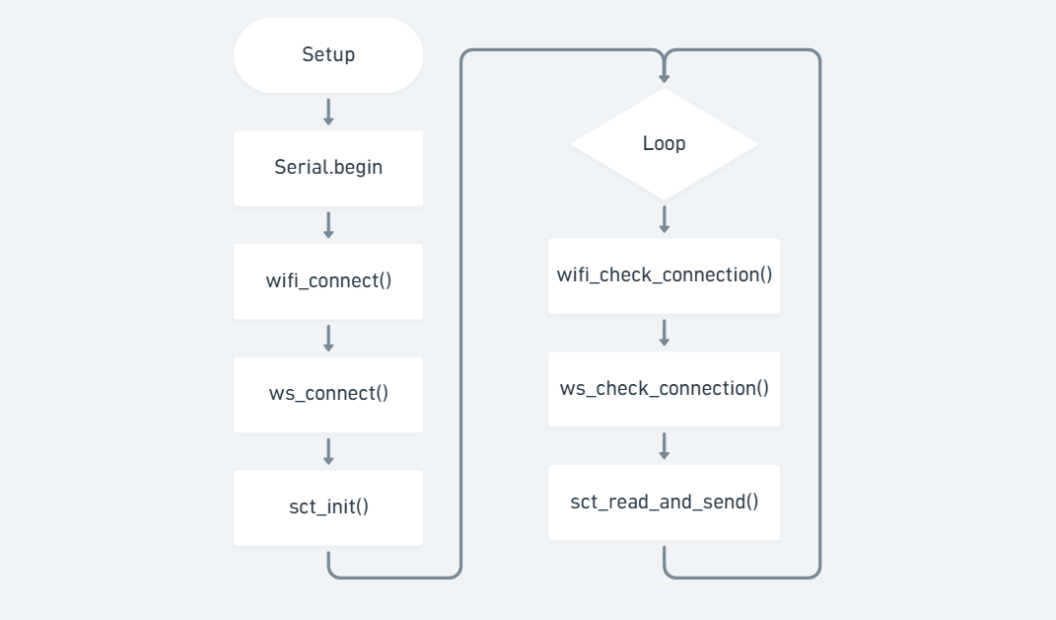
Todas as constantes de configuração, como credenciais Wi-Fi, endereço e porta do servidor WebSocket, parâmetros do ADC e do sensor de corrente, são definidas em um único

arquivo de cabeçalho (`config.h`), facilitando a adaptação do *firmware* a diferentes cenários de implantação sem a necessidade de modificar o código-fonte dos gerenciadores.

As bibliotecas externas utilizadas pelo *firmware* são: `ArduinoWebsockets`, que implementa o protocolo WebSocket conforme a RFC 6455 para microcontroladores; `ArduinoJson`, responsável pela serialização dos dados no formato JSON; e `WiFi.h`, fornecida pelo próprio *framework* Arduino-ESP32 para gerenciamento da interface Wi-Fi.

O ponto de entrada do *firmware* segue o padrão Arduino com duas funções principais. A função `setup()` é executada uma única vez na inicialização e realiza, em sequência: (i) a abertura da porta serial para depuração; (ii) a conexão à rede Wi-Fi; (iii) o estabelecimento da conexão WebSocket com o servidor; e (iv) a inicialização do módulo de leitura de corrente, incluindo a calibração do *offset* DC. A função `loop()` é executada continuamente e invoca, a cada iteração: a verificação da conectividade Wi-Fi, a verificação da conexão WebSocket e a rotina de leitura e envio dos dados de corrente. O Código-fonte 1 (Apêndice A) apresenta o arquivo `main.ino` na íntegra. O fluxo de execução é ilustrado na Figura 8.

Figura 8 – Fluxo de execução do programa principal (setup e loop) na ESP32



Fonte: elaborado pelo autor (2026).

3.1.3 Aquisição e processamento dos dados de corrente

A medição de corrente elétrica é realizada pelo módulo `sct_manager`, que configura o ADC, calibra o *offset* DC e calcula o valor eficaz (RMS) da corrente a partir das amostras

digitalizadas. Esta subseção detalha cada uma dessas etapas.

3.1.3.1 Configuração do ADC

O pino GPIO 34 é configurado com uma faixa de leitura de 0 a 3,3 V com resolução de 12 bits. A tensão correspondente a cada contagem bruta n do ADC é dada por:

$$V_{\mathrm{ADC}} = n \cdot \frac{V_{\mathrm{ref}}}{N} = n \cdot \frac{3{,}3}{4096} \tag{6}$$

em que $V_{\mathrm{ref}} = 3{,}3\,\mathrm{V}$ é a tensão de referência e $N = 4096$ é o número de níveis de quantização.

3.1.3.2 Calibração do offset DC

O circuito de polarização impõe ao sinal do sensor um *offset* DC próximo a 1,65 V, que corresponde a aproximadamente 2048 contagens no ADC. Entretanto, tolerâncias dos resistores e variações na referência interna do conversor introduzem um desvio em relação a esse valor nominal. Para compensar esse desvio, o *firmware* emprega uma estratégia de calibração em duas etapas:

1. Na inicialização (`sct_init`): são coletadas 300 amostras em um intervalo de aproximadamente 60 ms (taxa de cerca de 5 kHz), e a média aritmética dessas amostras é utilizada para inicializar a variável de *offset*. Esse procedimento pressupõe que, no instante da energização, não há corrente de carga significativa circulando pelo condutor monitorado, de modo que a média das leituras represente fielmente o nível DC do circuito;
2. Em tempo de execução: o *offset* é atualizado continuamente por meio de um filtro digital *Infinite Impulse Response* (resposta ao impulso infinita) (IIR) passa-baixa de primeira ordem, aplicado a cada amostra coletada pelo ADC. A equação de atualização é:

$$b_k = b_{k-1} + \frac{n_k - b_{k-1}}{1024} \tag{7}$$

em que b_k é o valor de *offset* após a k-ésima amostra e n_k é a contagem bruta do ADC. O coeficiente $\alpha = 1/1024$ resulta em uma constante de tempo elevada, de modo que o filtro acompanha lentamente derivas térmicas e variações de longo prazo no circuito de condicionamento, sem reagir à componente alternada do sinal de 60 Hz.

Essa abordagem é análoga à utilizada pela biblioteca de referência EmonLib (OPEN ENERGY MONITOR, 2024), na qual o *offset* DC é rastreado por um filtro digital em vez de ser

recalculado em janelas dedicadas. A principal vantagem é a eliminação da necessidade de uma fase exclusiva de recalibração, permitindo que todas as amostras coletadas sejam aproveitadas simultaneamente para o cálculo do valor eficaz.

3.1.3.3 Medição em passagem única

A função `sct_read_and_send()` realiza a aquisição e o cálculo do valor RMS em uma única passagem (*single-pass*) sobre uma janela temporal de 100 ms. Essa janela abrange aproximadamente seis ciclos completos da rede elétrica a 60 Hz, intervalo suficiente para obter uma estimativa estável do valor eficaz. Como o rastreamento do *offset* é realizado de forma integrada ao laço de amostragem pelo filtro IIR, não há necessidade de uma fase separada de recalibração. Cada chamada da função produz diretamente um resultado de medição, resultando em uma taxa efetiva de envio de aproximadamente dez leituras por segundo.

3.1.3.4 Cálculo do valor RMS

Durante a janela de medição, para cada amostra n_i coletada pelo ADC, a tensão AC no resistor de carga é obtida subtraindo-se o *offset* corrente do filtro IIR:

$$v_{\mathrm{AC}_i} = (n_i - b_i) \cdot \frac{V_{\mathrm{ref}}}{N} \tag{8}$$

em que b_i é o valor de *offset* atualizado pelo filtro IIR na i-ésima amostra (conforme a Equação (7)). O valor eficaz da tensão sobre o *burden* é então calculado pela raiz quadrada da média dos quadrados das M amostras coletadas na janela:

$$V_{\mathrm{RMS}} = \sqrt{\frac{1}{M} \sum_{i=1}^{M} v_{\mathrm{AC},i}^2} \tag{9}$$

Por fim, o valor eficaz da corrente no condutor primário é obtido multiplicando-se V_{RMS} pelo fator de calibração K_{cal}:

$$I_{\mathrm{RMS}} = V_{\mathrm{RMS}} \cdot K_{\mathrm{cal}} \tag{10}$$

O fator de calibração é definido como a razão entre o número de espiras do transformador de corrente e a resistência de carga:

$$K_{\text{cal}} = \frac{N_{\text{TC}}}{R_{\text{burden}}} = \frac{2000}{1000} = 2{,}0 \tag{11}$$

em que $N_{\text{TC}} = 2000$ é a relação de espiras do SCT-013-100 e $R_{\text{burden}} = 1000\,\Omega$ é a resistência de carga. Dessa forma, uma tensão eficaz de 1 V no *burden* corresponde a uma corrente eficaz de 2 A no condutor monitorado.

Os valores de corrente eficaz (I_{RMS}) são arredondados para três casas decimais antes da serialização e do envio via WebSocket. O Código-fonte 2 (Apêndice B) apresenta a implementação da função `sct_read_and_send()`, evidenciando a medição em passagem única com o filtro IIR de rastreamento do *offset*.

3.1.4 Comunicação via WebSocket

A transmissão dos dados ao servidor é realizada por meio do protocolo WebSocket, implementado pela biblioteca `ArduinoWebsockets`. Esse protocolo estabelece um canal de comunicação bidirecional e persistente sobre TCP, eliminando a sobrecarga de abertura e fechamento de conexões HTTP a cada envio de dados, característica relevante para aplicações que exigem baixa latência e alta frequência de transmissão.

3.1.4.1 Estabelecimento e manutenção da conexão

A conexão inicial com o servidor é estabelecida de forma bloqueante durante a execução de *setup()*: caso a tentativa falhe, o *firmware* aguarda 5 segundos e repete o processo até que a conexão seja bem-sucedida. O endereço de destino é composto pelo IP do servidor, a porta (3000) e o caminho do *endpoint* WebSocket (*/ws*), todos definidos em *config.h*.

Durante a operação em regime permanente, a função `ws_check_connection()` é invocada a cada iteração do `loop()`. Essa função verifica a disponibilidade da conexão por meio do método `available()` do cliente WebSocket e, caso detecte desconexão, aguarda o intervalo de reconexão e invoca novamente `ws_connect()`. O mesmo mecanismo é aplicado à conexão Wi-Fi pelo módulo `wifi_manager`, garantindo que tanto a camada de rede quanto a camada de aplicação sejam restauradas automaticamente após falhas transitórias.

3.1.4.2 Formato do payload

Cada leitura de corrente é encapsulada em um objeto JSON com a estrutura apresentada no Código-fonte 3 (Apêndice C).

Os campos possuem os seguintes significados:

- `type`: identificador do tipo de leitura, fixo em `"current"`. Esse campo permite ao servidor distinguir entre diferentes categorias de sensores em uma futura expansão do sistema;
- `time`: *timestamp* em milissegundos, obtido pela função `millis()` do Arduino, que representa o tempo decorrido desde a inicialização do microcontrolador;
- `current_rms`: valor eficaz da corrente, em amperes, com três casas decimais de precisão;

A serialização JSON é realizada pela biblioteca `ArduinoJson` com alocação estática de 96 bytes (`StaticJsonDocument<96>`), evitando o uso de alocação dinâmica de memória no *heap*, prática recomendada em sistemas embarcados para prevenir fragmentação de memória. O *buffer* serializado é então enviado ao servidor pela função `ws_send_json()`, que verifica a disponibilidade da conexão antes da transmissão e retorna um indicador de sucesso ou falha.

3.2 Backend

O *backend* constitui a camada intermediária da plataforma SEMS, responsável por receber os dados transmitidos pelo dispositivo embarcado, processá-los e disponibilizá-los tanto em tempo real quanto sob demanda para a interface *frontend*. O servidor foi implementado com o *framework* NestJS na versão 11, utilizando TypeScript como linguagem de programação. A escolha do NestJS justifica-se por três fatores principais: (i) o suporte nativo a WebSocket e SSE, protocolos centrais na arquitetura de tempo real adotada; (ii) o sistema de injeção de dependências e organização modular, que favorece a separação de responsabilidades; e (iii) a ampla compatibilidade com o ecossistema Node.js, incluindo bibliotecas como TypeORM, Zod e RxJS.

As subseções a seguir descrevem a arquitetura interna do servidor (subseção 3.2.1), o fluxo de recepção e processamento dos dados em tempo real (subseção 3.2.2), a estratégia de persistência no banco de dados (subseção 3.2.3) e os *endpoints* expostos pela API REST e pelo *stream* SSE (subseção 3.2.4).

3.2.1 Arquitetura do servidor

A organização interna do *backend* segue os princípios da Arquitetura Hexagonal (*Ports and Adapters*), proposta por (COCKBURN, 2005). Nesse modelo, a lógica de negócio é isolada em um núcleo de domínio que se comunica com o mundo externo exclusivamente por meio de interfaces (portas), implementadas por adaptadores concretos na camada de infraestrutura. A adoção desse padrão permite que componentes como o banco de dados, o protocolo de comunicação ou a biblioteca de geração de identificadores sejam substituídos sem alterações no código de domínio ou de aplicação.

Na prática, cada módulo de funcionalidade é estruturado em três diretórios:

- `domain/`: contém as entidades de domínio e as interfaces das portas (*ports*). As entidades são classes imutáveis que representam os conceitos do negócio, enquanto as portas definem contratos abstratos para operações como persistência e geração de identificadores;
- `application/`: abriga os serviços de aplicação, que orquestram a lógica de negócio utilizando as portas definidas no domínio. Os serviços recebem as dependências por injeção, referenciando apenas as interfaces, e não as implementações concretas;
- `infra/`: agrupa os adaptadores que implementam as portas: repositórios TypeORM, *gateways* WebSocket, controladores REST, esquemas de validação Zod e demais componentes acoplados a tecnologias específicas.

O servidor é composto por quatro módulos NestJS, registrados no módulo raiz `AppModule`. O Quadro 2 descreve cada módulo e sua responsabilidade.

Quadro 2 – Módulos do *backend* NestJS e suas responsabilidades

Módulo	Responsabilidade
`ConfigModule`	Carrega e valida as variáveis de ambiente com Zod; expõe o serviço `ConfigService` para acesso tipado às configurações
`DatabaseModule`	Configura a conexão com o PostgreSQL via TypeORM, com suporte a migrações manuais e contexto transacional
`HealthModule`	Disponibiliza o *endpoint* `GET /health` para verificação de disponibilidade do serviço
`DeviceModule`	Concentra toda a lógica de recepção, processamento, persistência e exposição dos dados de corrente elétrica

Fonte: elaborado pelo autor (2026).

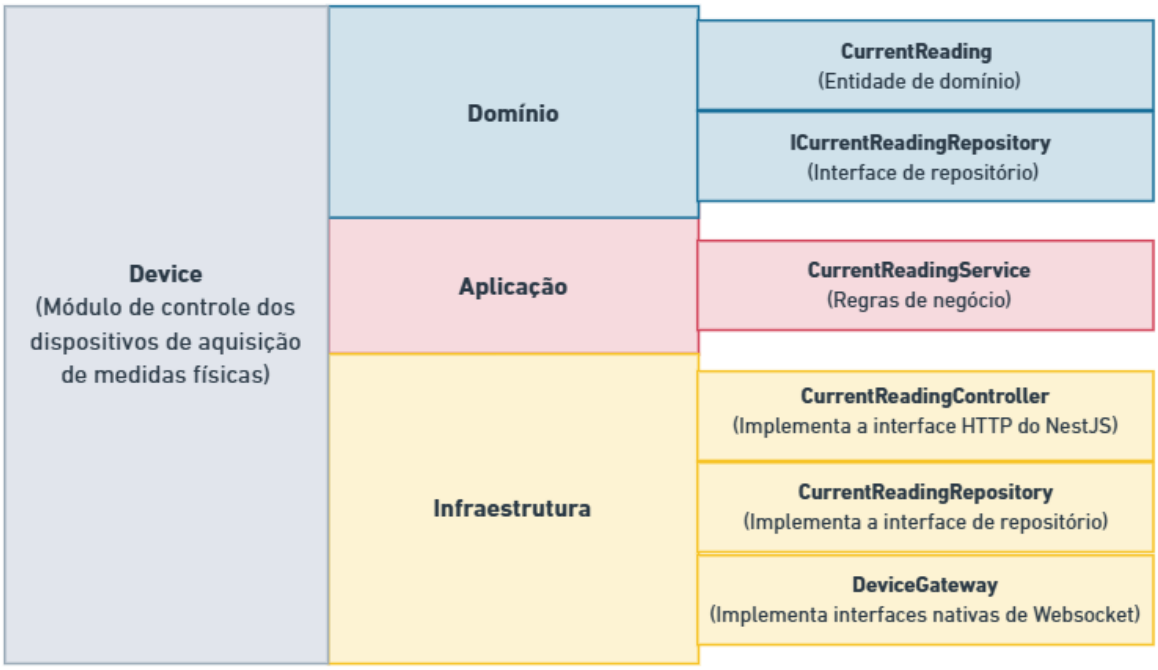
A validação de dados permeia todas as fronteiras de entrada do sistema. As variáveis de ambiente são validadas na inicialização por meio de esquemas Zod compostos (`CommonVarsSchema` e `DatabaseVarsSchema`), garantindo que valores obrigatórios, como credenciais de banco de dados e origens *Cross-Origin Resource Sharing* (CORS), estejam presentes

e corretamente tipados antes da criação do contexto da aplicação. O mesmo mecanismo é empregado para validar os *payloads* recebidos via WebSocket e os parâmetros de consulta da API REST, conforme será detalhado nas subseções seguintes.

O sistema de *logging* utiliza a biblioteca Pino, configurada como serviço de *log* customizado (`CustomLoggerService`) que implementa a interface `LoggerService` do NestJS. Em ambiente de desenvolvimento, os registros são formatados com cores e *timestamps* legíveis por meio do módulo `pino-pretty`; em produção, os registros são emitidos em formato JSON estruturado, facilitando a ingestão por ferramentas de observabilidade.

A Figura 9 apresenta um diagrama da arquitetura interna do *backend* utilizando como exemplo o principal módulo *Device*, que controla os dispositivos, evidenciando a organização em camadas e os principais componentes de cada módulo. Na camada de domínio, encontram-se a classe *CurrentReading*, a qual representa a leitura de corrente, e a interface do seu repositório, *ICurrentReadingRepository*, que define os métodos de leitura e persistência dessa entidade de domínio, sendo utilizada para implementar o `CurrentReadingRepository`. Na camada de aplicação, o serviço *CurrentReadingService* implementa a lógica de processamento e orquestra as operações de persistência e publicação. Na camada de infraestrutura, o *gateway* WebSocket (`DeviceGateway`) gerencia a recepção dos dados em tempo real, enquanto o repositório (`CurrentReadingRepository`) é responsável pela interação com o banco de dados PostgreSQL. Por fim, o *CurrentReadingController* é a classe padrão do NestJS que expõe os *endpoints* REST e SSE para consumo pelo *frontend*.

Figura 9 – Diagrama da arquitetura do módulo *devices*, organizado em camadas conforme o padrão *Ports and Adapters*



Fonte: elaborado pelo autor (2026).

O ponto de entrada da aplicação é o arquivo `main.ts`, cuja função `bootstrap()` executa a seguinte sequência de inicialização: (i) habilita o contexto transacional do TypeORM; (ii) cria a instância da aplicação NestJS com o *logger* Pino; (iii) configura o adaptador WebSocket nativo (`WsAdapter`); (iv) aplica os *middlewares* de segurança Helmet e `cookie-parser`; (v) habilita CORS com origens configuráveis; (vi) registra os filtros globais de exceção; e (vii) inicia a escuta na porta 3000. O Código-fonte 4 (Apêndice D) apresenta esse arquivo na íntegra.

3.2.2 Recepção e processamento dos dados em tempo real

A recepção dos dados provenientes do dispositivo embarcado é realizada por um *gateway* WebSocket nativo, implementado pela classe `DeviceGateway`. Diferentemente de bibliotecas de alto nível como Socket.IO, o servidor utiliza o módulo `ws`, uma implementação leve e conforme à RFC 6455, integrado ao NestJS por meio do adaptador `@nestjs/platform-ws`. O *endpoint* WebSocket é exposto no caminho `/ws`, e cada conexão é gerenciada individualmente: ao conectar-se, o *gateway* registra o endereço IP e a porta de origem do dispositivo; ao desconectar-se, emite um registro de *log* correspondente.

Para cada mensagem recebida, o *gateway* executa duas etapas de validação. Primeiro, o conteúdo bruto é convertido de `Buffer` para *string* UTF-8 e submetido a `JSON.parse()`;

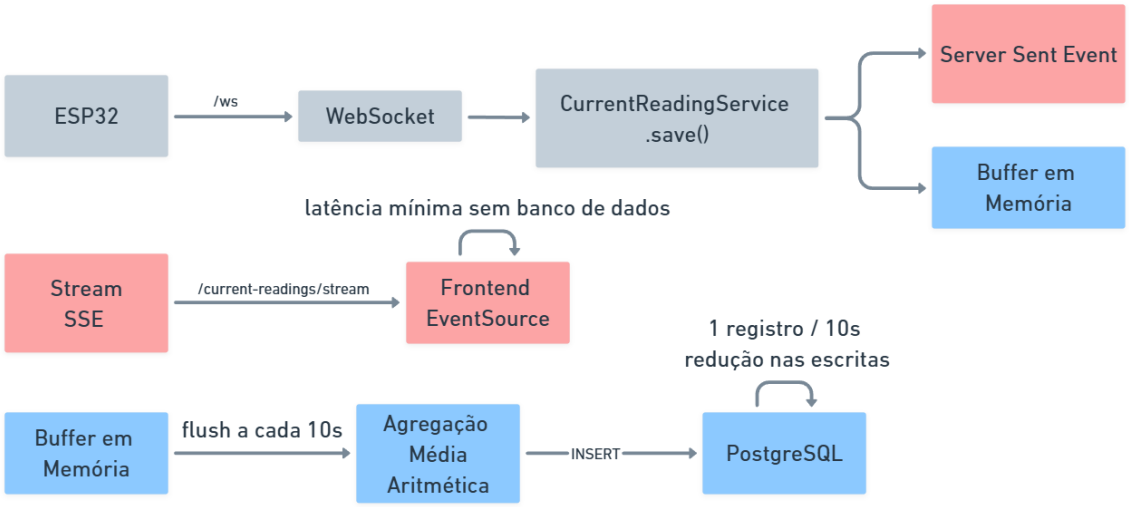
mensagens que não constituem JSON válido são descartadas com um aviso no *log*. Em seguida, o objeto resultante é validado contra o esquema Zod `CurrentReadingSchema`, que exige a presença dos campos `type` (literal `"current"`), `time` (número) e `current_rms` (número). Mensagens que não satisfazem o esquema são igualmente descartadas, e as violações específicas são registradas no *log* para fins de depuração.

Após a validação bem-sucedida, os valores extraídos são encaminhados ao *CurrentReadingService* por meio do método `save()`, que implementa o processamento de caminho duplo (*dual-path*) ilustrado na Figura 10:

1. **Caminho de tempo real (SSE):** o método `save()` invoca imediatamente a chamada `readingsSubject.next()`, publicando a leitura em um `Subject` da biblioteca RxJS. Esse `Subject` atua como barramento reativo: todos os clientes conectados ao *stream* SSE recebem o evento com latência mínima, sem passagem pelo banco de dados;
2. **Caminho de persistência (agregação):** simultaneamente, a leitura é adicionada a um *buffer* em memória (*array* de objetos `BufferedReading`). Um temporizador periódico, inicializado no ciclo de vida `onModuleInit()` do serviço, dispara a cada intervalo configurável (variável de ambiente `CURRENT_READING_WINDOW_MS`, com valor padrão de 10.000 ms) a função `flush()`, que calcula a média aritmética dos campos `time` e `current_rms` de todas as leituras acumuladas na janela e persiste um único registro consolidado no banco de dados.

A Figura 10 ilustra o processamento em caminho duplo dos dados provenientes da camada de percepção (ESP32). Os dados são transmitidos pelo microcontrolador e recebidos pelo gateway WebSocket do servidor, sendo então processados pelo serviço de leitura de corrente. Esse serviço executa duas operações distintas: (i) publica imediatamente cada leitura no ReadingSubject do NestJS, permitindo o envio de eventos SSE consumidos pela interface EventSource do frontend; e (ii) agrega as leituras em memória, realizando a inserção da média aritmética dos valores no banco de dados ao final de cada ciclo de agregação.

Figura 10 – Diagrama do processamento de caminho duplo (*dual-path*)



Fonte: elaborado pelo autor (2026).

Essa estratégia de *downsampling* temporal é fundamental para a viabilidade do armazenamento. Considerando que o dispositivo transmite aproximadamente 10 leituras por segundo, a persistência direta de todas as amostras resultaria em cerca de 864.000 registros por dia. Com a agregação em janelas de 10 segundos, esse volume é reduzido para 8.640 registros diários (uma redução de 99% na carga de escrita), mantendo-se a resolução temporal adequada para análises históricas.

O Código-fonte 5 (Apêndice E) apresenta a implementação completa do serviço `CurrentReadingService`, evidenciando a lógica de *buffering*, agregação e publicação reativa.

3.2.3 Persistência dos dados

A camada de persistência utiliza o PostgreSQL como sistema gerenciador de banco de dados relacional, acessado por meio do ORM TypeORM. A configuração da conexão é centralizada na classe `DatabaseConfig`, que obtém os parâmetros de conexão (host, porta, credenciais e nome do banco) a partir das variáveis de ambiente previamente validadas. Em ambientes de produção, a conexão é protegida por TLS mediante um certificado SSL configurável; em ambiente de desenvolvimento, essa exigência é dispensada para simplificar o fluxo de trabalho local.

Um aspecto importante da estratégia adotada é a desabilitação da sincronização automática de esquema (`synchronize: false`). Todas as alterações na estrutura do banco de

dados são gerenciadas por meio de migrações manuais, escritas em TypeScript e executadas pela CLI do TypeORM. Essa abordagem oferece rastreabilidade e reprodutibilidade: cada migração é versionada no repositório de código e pode ser aplicada ou revertida de forma controlada em qualquer ambiente. O Código-fonte 6 (Apêndice F) apresenta a migração que cria a tabela `current_readings`.

A entidade de domínio `CurrentReading` é uma classe imutável com seis atributos, descritos no Quadro 3.

Quadro 3 – Atributos da entidade de domínio `CurrentReading`

Atributo	Tipo	Descrição
`id`	UUID (*string*)	Identificador único gerado pelo serviço `IdGenerationService` com `crypto.randomUUID()`
`timeMs`	Inteiro	*Timestamp* médio da janela de agregação, em milissegundos
`currentRms`	Numérico (10,3)	Valor eficaz médio da corrente na janela, em ampères
`createdAt`	Timestamp com fuso	Data e hora de criação do registro
`updatedAt`	Timestamp com fuso	Data e hora da última atualização do registro

Fonte: elaborado pelo autor (2026).

O mapeamento entre a entidade de domínio e a entidade de persistência (ORM) é realizado pela classe `CurrentReadingEntity`, que estende uma classe base abstrata que possui os campos básicos de um registro, definida como `DatabaseEntity`. Essa classe base fornece os campos comuns id (chave primária UUID), `createdAt` e `updatedAt` com decoradores do TypeORM. A conversão entre as duas representações é efetuada por dois métodos estáticos: `fromDomain()`, que transforma um objeto de domínio em entidade persistível, e `toDomain()`, que realiza a operação inversa. Essa separação garante que a camada de domínio permaneça independente do ORM.

O repositório *CurrentReadingRepository* implementa a interface *ICurrentReadingRepository*, definida como porta na camada de domínio, e é injetado no serviço de aplicação por meio de um *token* simbólico (`CURRENT_READING_REPOSITORY_TOKEN`), seguindo o padrão de inversão de dependência. As operações disponíveis são: `save()`, que persiste um registro agregado, e `findMany()`, que realiza consultas paginadas com filtros temporais opcionais, construídas dinamicamente com o *QueryBuilder* do TypeORM.

3.2.4 Endpoints da API REST e SSE

O *backend* expõe dois *endpoints* HTTP para comunicação com a camada de apresentação e com sistemas externos: um de consulta paginada e um de *streaming* em tempo real. O Quadro 4 resume os *endpoints* disponíveis.

Quadro 4 – *Endpoints* HTTP expostos pelo *backend*

Método/Tipo	Caminho	Descrição
`GET`	`/current-readings`	Consulta paginada dos registros de corrente armazenados, com suporte a filtros temporais
`SSE`	`/current-readings/stream`	*Stream* de eventos em tempo real via *Server-Sent Events*, emitindo cada leitura recebida do dispositivo

Fonte: elaborado pelo autor (2026).

3.2.4.1 Consulta paginada de leituras

O *endpoint* `GET /current-readings` permite a consulta dos registros agregados armazenados no banco de dados. Os parâmetros de consulta são validados pelo esquema Zod `GetReadingsQuerySchema` e incluem:

- `from` (*string* ISO 8601, opcional): limite inferior do filtro temporal;
- `to` (*string* ISO 8601, opcional): limite superior do filtro temporal;
- `limit` (inteiro, padrão 100, máximo 1000): número máximo de registros por página;
- `offset` (inteiro, padrão 0): deslocamento para paginação.

A resposta é um objeto JSON com a estrutura ilustrada no Código-fonte 7 (Apêndice G), contendo o *array* de registros, o total de registros correspondentes ao filtro e os parâmetros de paginação aplicados.

Os registros são ordenados de forma decrescente pela data de criação (*createdAt DESC*), de modo que as leituras mais recentes aparecem primeiro. A validação dos parâmetros de consulta é integrada ao controlador por meio do decorador customizado *@QueryWithValidation*, que instancia um *ValidationPipe* com o esquema Zod apropriado. Caso algum parâmetro seja inválido, o filtro global *RequestValidationFilter* intercepta o erro Zod e retorna uma resposta HTTP 400 com a lista detalhada das violações.

3.2.4.2 Stream SSE de leituras em tempo real

O *endpoint /current-readings/stream* utiliza o mecanismo de *Server-Sent Events* (SSE), um protocolo baseado em HTTP que permite ao servidor enviar eventos unidirecionais ao cliente por meio de uma conexão persistente. No NestJS, esse comportamento é implementado pelo decorador *@Sse*, que espera como retorno um *Observable* de *MessageEvent*.

O controlador subscreve-se ao *Observable readings$* exposto pelo *CurrentReading-Service* e transforma cada evento em um *MessageEvent* por meio do operador *map()* do RxJS. Dessa forma, cada leitura de corrente validada e publicada pelo *gateway* WebSocket é imediatamente propagada a todos os clientes SSE conectados, sem intermediação do banco de dados. O Código-fonte 8 (Apêndice H) apresenta a implementação do método *stream()* no controlador.

No lado do cliente, a conexão SSE é estabelecida pela API nativa *EventSource* do navegador, que gerencia automaticamente a reconexão em caso de interrupções transitórias. Cada evento recebido contém, no campo *data*, um objeto JSON com os campos *timeMs* e *currentRms*, prontos para renderização imediata na interface.

3.3 Frontend

A camada de apresentação da plataforma SEMS consiste em uma aplicação *web* de página única (*Single-Page Application* — SPA) que oferece ao usuário duas funcionalidades principais: o monitoramento de corrente elétrica em tempo real, alimentado pelo *stream* SSE descrito na subseção 3.2.4, e a consulta de séries históricas armazenadas no banco de dados, acessadas por meio da API REST do *backend*. A interface foi projetada como um *dashboard* centralizado, com navegação por abas que permite alternar entre os modos de visualização sem recarregamento de página.

As subseções a seguir descrevem as tecnologias empregadas (subseção 3.3.1), a arquitetura e organização do código-fonte (subseção 3.3.2), os mecanismos de comunicação com o servidor (subseção 3.3.3), a interface de monitoramento em tempo real (subseção 3.3.4) e a visualização de dados históricos (subseção 3.3.5).

3.3.1 Tecnologias e ferramentas

A aplicação *frontend* foi desenvolvida com a biblioteca React na versão 19, utilizando TypeScript como linguagem de programação. O React adota um modelo de renderização

declarativo baseado em componentes, no qual a interface é descrita como uma função do estado da aplicação, abordagem que simplifica a atualização do DOM (*Document Object Model*) em cenários de alta frequência de mudanças, como a exibição de dados em tempo real. A adoção do TypeScript acrescenta tipagem estática ao JavaScript, contribuindo para a detecção precoce de erros e para a documentação implícita das interfaces entre componentes.

A ferramenta de *build* utilizada é o Vite na versão 8, que emprega o *bundler* nativo ESBuild para o servidor de desenvolvimento e o Rollup para a construção de produção. Em comparação com ferramentas tradicionais como o Webpack, o Vite oferece tempos de inicialização e de atualização a quente (*Hot Module Replacement* — HMR) significativamente menores, o que agiliza o ciclo de desenvolvimento.

A estilização da interface é realizada com o *framework* utilitário Tailwind CSS na versão 4, integrado ao Vite por meio do *plugin* oficial `@tailwindcss/vite`. O Tailwind CSS adota a abordagem *utility-first*, na qual classes CSS atômicas são compostas diretamente no *markup* dos componentes, eliminando a necessidade de folhas de estilo separadas e reduzindo conflitos de especificidade. O tema visual da aplicação é definido por variáveis CSS customizadas no espaço de cor OKLCH, configurando uma paleta escura (*dark theme*) com tons de azul-acinzentado e acentos em dourado.

Os componentes visuais reutilizáveis, como botões, *cards*, *badges*, campos de entrada e abas, seguem o padrão da biblioteca shadcn/ui, na qual cada componente é implementado localmente no projeto (e não importado de um pacote externo) utilizando a biblioteca `class-variance-authority` (CVA) para gerenciamento de variantes de estilo. Essa abordagem confere controle total sobre o código dos componentes, facilitando customizações sem dependência de atualizações externas.

A geração de gráficos é realizada pela biblioteca Recharts na versão 3.8, construída sobre a D3.js, que fornece componentes React declarativos para a criação de gráficos de linha, área, barra e outros tipos de visualização. Os ícones utilizados na interface provêm da biblioteca Lucide React, que disponibiliza um conjunto consistente de ícones vetoriais (SVG) otimizados para *tree-shaking*. O Quadro 5 resume as principais dependências do projeto.

O gerenciamento de pacotes é realizado pelo Yarn com o modo *Plug'n'Play* (PnP), que elimina a pasta `node_modules` tradicional e resolve as dependências diretamente de um cache compactado, reduzindo o uso de disco e acelerando a instalação.

Quadro 5 – Principais dependências do *frontend* e suas finalidades

Dependência	Versão	Finalidade
React / React DOM	19.2	Biblioteca de construção de interfaces
TypeScript	5.9	Tipagem estática sobre JavaScript
Vite	8.0	Ferramenta de *build* e servidor de desenvolvimento
React Router DOM	7.13	Roteamento de página única (*client-side routing*)
Tailwind CSS	4.2	*Framework* de estilização utilitário
Recharts	3.8	Biblioteca de gráficos declarativos para React
Lucide React	1.7	Conjunto de ícones vetoriais SVG
`class-variance-authority`	0.7	Gerenciamento de variantes de estilo em componentes
`clsx` / `tailwind-merge`	2.1 / 3.5	Composição e resolução de conflitos de classes CSS

Fonte: elaborado pelo autor (2026).

3.3.2 Arquitetura e organização do código

O código-fonte do *frontend* é organizado segundo o padrão *Atomic Design*, proposto por (FROST, 2016), que classifica os componentes de interface em níveis hierárquicos de complexidade crescente. A estrutura de diretórios do projeto reflete essa organização, conforme apresentado no Quadro 6.

Quadro 6 – Estrutura de diretórios do *frontend*

Diretório	Conteúdo
`src/components/ui/`	Componentes primitivos da interface (botões, *cards*, *badges*, campos de entrada, abas), implementados no padrão shadcn/ui
`src/components/atoms/`	Componentes atômicos específicos da aplicação (`Logo`, `ModeToggle`)
`src/components/molecules/`	Componentes compostos que agrupam átomos e primitivos para formar unidades funcionais (`Hero`, `ReadingStats`, `RealtimeChart`, `HistoricalChart`, `DateRangeFilter`)
`src/hooks/`	*Hooks* customizados que encapsulam a lógica de comunicação com o *backend* e o gerenciamento de estado
`src/lib/`	Funções utilitárias e módulo de acesso à API
`src/pages/`	Componentes de página (`Dashboard`)

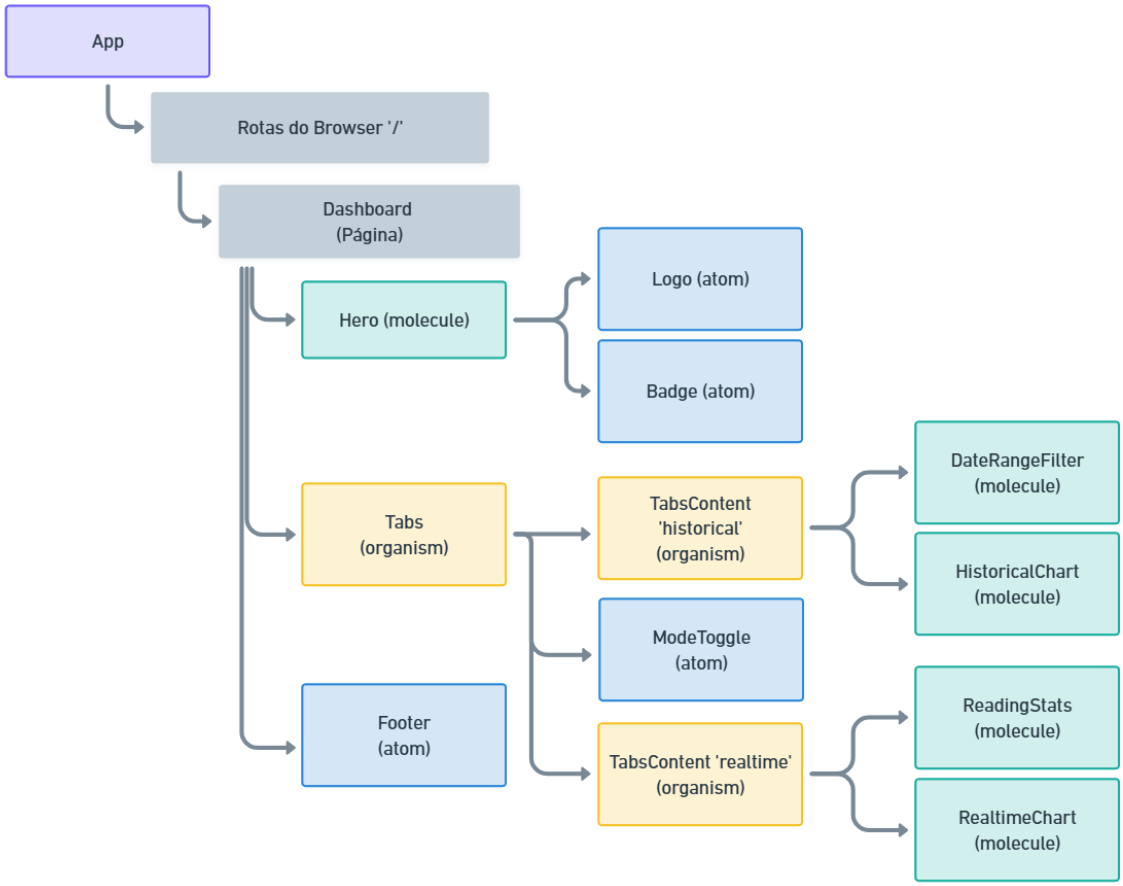
Fonte: elaborado pelo autor (2026).

A aplicação possui uma única rota (`/`), configurada com a biblioteca React Router DOM, que renderiza o componente `Dashboard`. Esse componente atua como a camada inteligente (*smart component*) da aplicação: instancia os *hooks* de dados, gerencia o estado de navegação entre abas e distribui as informações aos componentes de apresentação por meio de propriedades (*props*). Não é utilizada nenhuma biblioteca externa de gerenciamento de estado global (como Redux ou Zustand); todo o estado é mantido localmente nos *hooks* customizados e propagado de forma descendente pela árvore de componentes.

A Figura 11 apresenta a hierarquia de componentes da interface do frontend, es-

truturada segundo o padrão Atomic Design. O componente principal, Dashboard, atua como orquestrador e distribui os dados para os componentes filhos, como Hero, Tabs, ReadingStats, RealtimeChart, DateRangeFilter e HistoricalChart. As setas indicam o fluxo de propriedades (props) entre os componentes, evidenciando a separação entre lógica de controle e apresentação visual.

Figura 11 – Hierarquia de componentes do *frontend*



Fonte: elaborado pelo autor (2026).

3.3.3 Comunicação com o backend

A interface consome duas interfaces de comunicação expostas pelo *backend*: o *stream* SSE para dados em tempo real e a API REST para dados históricos. Cada canal é encapsulado em um *hook* customizado do React, que gerencia o ciclo de vida da conexão, o armazenamento local dos dados recebidos e o tratamento de erros. Essa separação permite que a lógica de comunicação seja reutilizada e testada independentemente dos componentes visuais.

3.3.3.1 Consumo do stream SSE

O *hook* `useCurrentReadingStream` utiliza a API nativa `EventSource` do navegador para estabelecer uma conexão SSE com o *endpoint* `/current-readings/stream` do *backend*. A URL base do servidor é obtida da variável de ambiente `VITE_API_URL`, definida no arquivo `.env` do projeto.

O ciclo de vida da conexão é gerenciado automaticamente pelo *hook*: a conexão é estabelecida na montagem do componente e encerrada na desmontagem, seguindo o padrão do *hook* `useEffect` do React. Para cada evento recebido, o conteúdo JSON é deserializado e transformado em um objeto `ReadingDataPoint`, que contém os valores de corrente RMS (arredondados a três casas decimais), o *timestamp* do sensor em milissegundos e uma representação textual do horário em formato local (`HH:MM:SS`).

Os dados são mantidos em um *buffer* circular de 100 pontos: quando o limite é atingido, as amostras mais antigas são descartadas para dar lugar às novas. Essa estratégia limita o consumo de memória do navegador e mantém a janela de visualização do gráfico em tempo real restrita aos últimos 100 eventos recebidos.

Em caso de perda de conexão, o *hook* implementa um mecanismo de reconexão automática: o manipulador do evento `onerror` fecha a instância do `EventSource`, atualiza o estado de conexão para desconectado e agenda uma nova tentativa de conexão após um intervalo de 3 segundos. Para evitar problemas de referência obsoleta (*stale closure*) na função de reconexão, a referência à função `connect()` é mantida em um `useRef`, garantindo que a versão mais recente seja sempre invocada.

O *hook* expõe a seguinte interface para os componentes consumidores:

- `data`: *array* de pontos de leitura no *buffer* circular;
- `connected`: indicador booleano do estado da conexão SSE;
- `error`: mensagem de erro em caso de falha de conexão;
- `latestReading`: última leitura recebida, ou `null` se nenhuma leitura foi processada;
- `clearData`: função para limpar o *buffer* de dados;
- `reconnect`: função para forçar uma reconexão manual.

3.3.3.2 Consumo da API REST

O *hook useHistoricalReadings* encapsula a lógica de consulta paginada ao *endpoint GET /current-readings* do *backend*. A função *fetch()* do *hook* recebe os parâmetros de filtro temporal (*from* e *to*, em formato ISO 8601) e um limite opcional de registros (padrão de 500), constrói a URL com *URLSearchParams* e realiza a requisição HTTP por meio da API nativa *fetch()* do navegador, sem dependência de bibliotecas externas como Axios.

A resposta do servidor, tipada como *PaginatedResponse<CurrentReadingRecord>*, é processada da seguinte forma: o *array* de registros é invertido (pois o *backend* retorna os mais recentes primeiro, enquanto o gráfico necessita da ordem cronológica crescente), e cada registro é mapeado para um objeto *HistoricalDataPoint* com o horário formatado em localidade brasileira (*DD/MM, HH:MM:SS*). O *hook* expõe os estados *data*, *loading*, *error* e *total*, bem como as funções *fetch()* e *clear()*.

O módulo *lib/api.ts* centraliza a construção de URLs e a definição dos tipos TypeScript compartilhados entre os *hooks*, incluindo as interfaces *CurrentReadingEvent* (dados do *stream* SSE), *CurrentReadingRecord* (registros do banco de dados) e *PaginatedResponse<T>* (envelope de respostas paginadas).

3.3.4 Interface de monitoramento em tempo real

A aba de monitoramento em tempo real constitui a visão principal da plataforma e é composta por três elementos: o cabeçalho com indicador de conexão, o painel de estatísticas e o gráfico dinâmico de corrente.

O **cabeçalho** (`Hero`) exibe o logotipo da aplicação, um ícone SVG estilizado representando um medidor elétrico com um raio e uma onda senoidal, acompanhado do nome “SEMS” e de um *badge* que indica o estado da conexão SSE. Quando a conexão está ativa, o *badge* exibe “Conectado” em verde; caso contrário, exibe “Desconectado” em vermelho, alertando o usuário sobre a indisponibilidade do *stream* de dados.

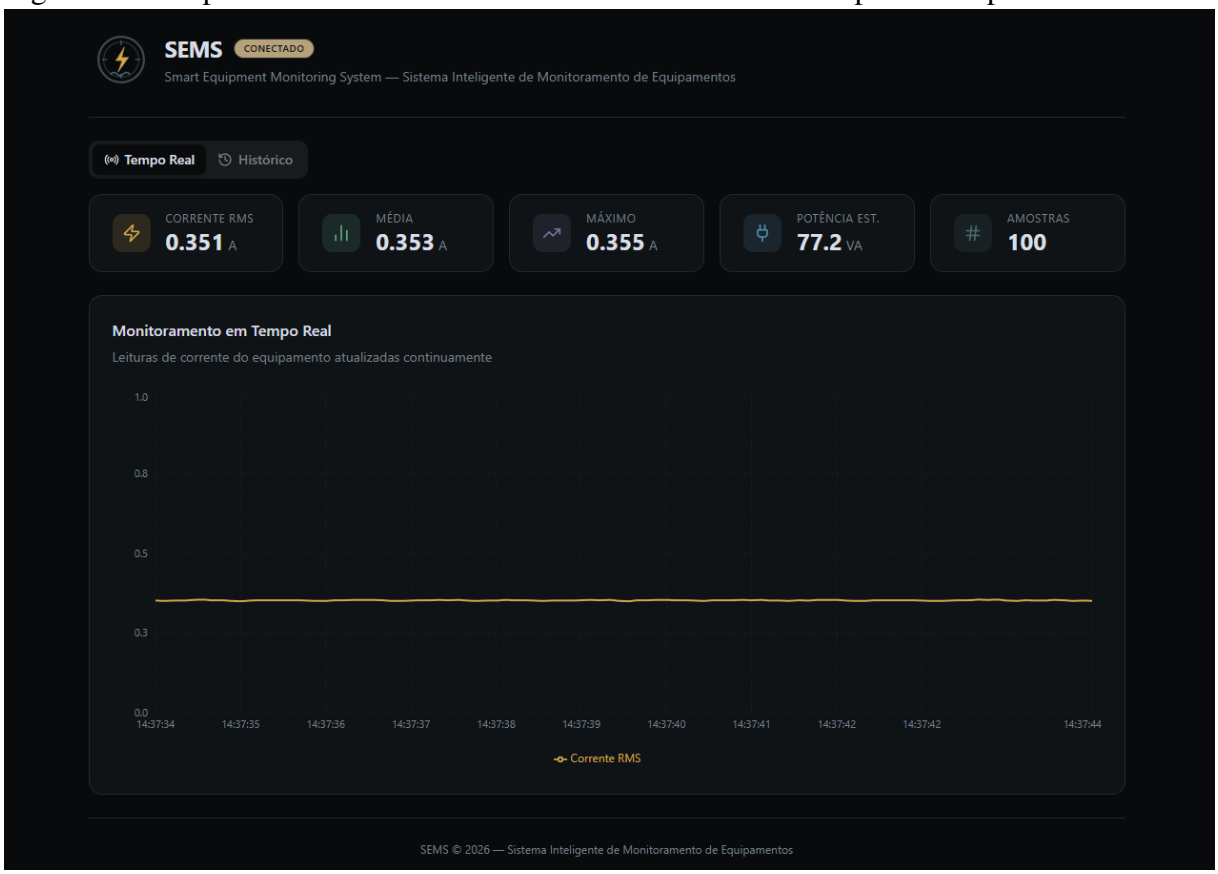
O **painel de estatísticas** (`ReadingStats`) apresenta cartões dispostos em uma grade responsiva: o primeiro exibe o valor de corrente RMS da última leitura recebida (em amperes, com três casas decimais) e o último exibe o número total de amostras acumuladas no *buffer*. Nessa seção, será possível adicionar futuramente outros dados estatísticos, tais como o valor de pico ou a estimativa de potência consumida.

O **gráfico de corrente em tempo real** (`RealtimeChart`) é implementado com o componente `LineChart` da biblioteca Recharts, encapsulado em um `ResponsiveContainer` que ocupa toda a largura disponível e possui altura fixa de 350 pixels (400 pixels em telas maiores). O gráfico renderiza a série de corrente RMS em dourado. O eixo horizontal exibe os horários das leituras em formato local: hora, minuto e segundo (`HH:MM:SS`), enquanto o eixo vertical apresenta os valores de corrente em amperes com uma casa decimal.

Para garantir o desempenho da renderização em alta frequência de atualização, duas otimizações são aplicadas: a animação de transição entre quadros é desabilitada (*isAnimationActive={false}*), e os marcadores de ponto sobre a linha são suprimidos (*dot={false}*). Essas configurações eliminam cálculos de interpolação e renderizações adicionais que seriam disparados a cada novo dado recebido, aproximadamente a cada 100 ms, resultando em uma experiência visual fluida mesmo em dispositivos com menor capacidade de processamento gráfico.

A Figura 12 ilustra a interface da aba de monitoramento em tempo real. No cabeçalho, estão o símbolo da plataforma, o título e um *badge* que indica o status da conexão. Logo abaixo, o painel de estatísticas apresenta os valores atuais de corrente RMS e o número de amostras acumuladas no buffer. O gráfico, localizado na parte inferior da tela, exibe a curva de corrente RMS. O eixo horizontal mostra os horários das leituras, enquanto o eixo vertical apresenta os valores de corrente em amperes. Essa disposição facilita a visualização rápida do estado do sistema e o acompanhamento das variações em tempo real.

Figura 12 – Captura de tela da interface de monitoramento em tempo real da plataforma SEMS



Fonte: elaborado pelo autor (2026).

3.3.5 Visualização de dados históricos

A aba de dados históricos permite ao usuário consultar e visualizar os registros de corrente armazenados no banco de dados, possibilitando a análise de comportamentos ao longo do tempo para fins de diagnóstico e manutenção preditiva.

O **filtro de período** (`DateRangeFilter`) apresenta dois campos de entrada do tipo `datetime-local` para a seleção das datas de início (“De”) e fim (“Até”) do intervalo de consulta. Ao ser montado, o componente inicializa automaticamente o intervalo com a última hora, ou seja, de uma hora atrás até o momento atual. Dois botões acompanham os campos: “Buscar”, que dispara a requisição à API REST com os parâmetros selecionados convertidos para formato ISO 8601, e “Limpar”, que reinicia os campos e remove os dados da visualização. Ambos os botões são desabilitados durante o carregamento de uma requisição em andamento, prevenindo requisições duplicadas.

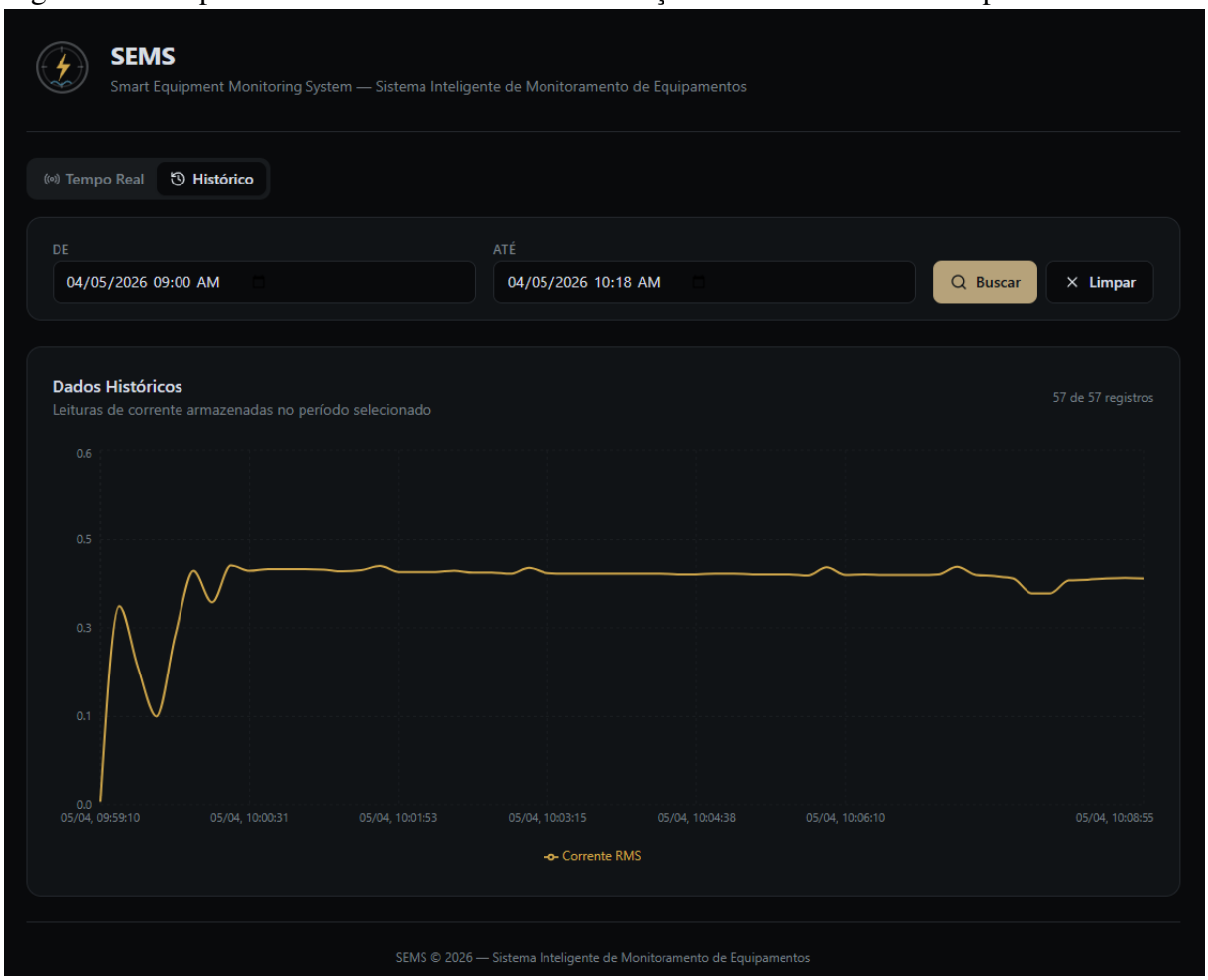
O **gráfico histórico** (`HistoricalChart`) utiliza o mesmo componente `LineChart` do Recharts empregado no gráfico de tempo real, com a mesma série de corrente e a mesma paleta de cores. As diferenças em relação ao gráfico de tempo real são:

- o eixo horizontal exibe datas no formato `DD/MM, HH:MM:SS`, adequado para intervalos que podem abranger múltiplos dias;
- os marcadores de ponto são exibidos condicionalmente, apenas quando o número de registros é inferior a 50, evitando sobrecarga visual em séries com alta densidade de dados;
- as animações de transição são mantidas habilitadas, visto que a renderização ocorre uma única vez após o carregamento dos dados, sem atualizações contínuas.

O componente gerencia três estados visuais distintos além da exibição do gráfico: um indicador de carregamento (ícone rotativo com a mensagem “Carregando dados...”), uma mensagem de erro em vermelho caso a requisição falhe, e um estado vazio inicial com a orientação “Selecione um período e clique em Buscar para visualizar os dados”. O número de registros carregados é exibido em um *badge* no cabeçalho do cartão, no formato “X de Y registros”, onde X representa a quantidade retornada na consulta e Y o total de registros correspondentes ao filtro no banco de dados.

A Figura 13 apresenta a interface da aba de visualização de dados históricos. É possível constatar o período em que o dispositivo é energizado por meio das oscilações características de corrente RMS no período transitório, seguidas por uma estabilização em torno de um valor constante, com perturbações menores.

Figura 13 – Captura de tela da interface de visualização de dados históricos da plataforma SEMS



Fonte: elaborado pelo autor (2026).

3.4 Fluxo geral de operação

Após a descrição individual de cada camada, esta seção apresenta o fluxo completo de operação da plataforma SEMS, desde a energização do dispositivo embarcado até a visualização dos dados pelo usuário na interface *web*. O objetivo é evidenciar como os componentes previamente descritos interagem entre si para formar um sistema coeso de monitoramento em tempo real. O fluxo é dividido em duas etapas: a **inicialização** do sistema e a **operação em regime permanente**.

3.4.1 Inicialização

A inicialização do sistema ocorre em duas frentes independentes, o dispositivo embarcado e o servidor, que convergem no momento em que a conexão WebSocket é estabelecida entre ambos.

No lado do **dispositivo embarcado**, a energização da ESP32 desencadeia a execução da função `setup()`, que realiza as seguintes operações de forma sequencial e bloqueante:

1. **Abertura da porta serial**: a interface serial é configurada a 115.200 bps para emissão de mensagens de depuração;
2. **Conexão Wi-Fi**: o módulo `wifi_manager` configura a ESP32 no modo estação (STA) e tenta conectar-se à rede sem fio com as credenciais definidas em `config.h`. O *firmware* permanece em espera ativa, verificando o estado da conexão a cada 500 ms, até que o vínculo com o ponto de acesso seja confirmado;
3. **Conexão WebSocket**: o módulo `ws_manager` tenta estabelecer uma conexão WebSocket com o servidor no endereço e porta configurados (caminho `/ws`). Caso a tentativa falhe, o módulo aguarda 5 segundos e repete o processo indefinidamente até obter sucesso;
4. **Calibração do sensor de corrente**: o módulo `sct_manager` coleta 300 amostras do ADC em aproximadamente 60 ms e calcula a média aritmética para inicializar o filtro IIR de rastreamento do *offset* DC do circuito de condicionamento. Essa etapa pressupõe ausência de corrente de carga no condutor monitorado.

No lado do **servidor**, a aplicação NestJS é inicializada pela função `bootstrap()` do arquivo `main.ts`, que executa a seguinte sequência: habilita o contexto transacional do TypeORM, cria a instância da aplicação com o *logger* Pino, configura o adaptador WebSocket nativo (`WsAdapter`), aplica os *middlewares* de segurança (Helmet, `cookie-parser`, CORS), registra os filtros globais de exceção e inicia a escuta na porta 3000. Após a criação do contexto da aplicação, o *hook* de ciclo de vida `onModuleInit()` do `CurrentReadingService` é invocado automaticamente pelo NestJS, inicializando o temporizador periódico de agregação e persistência dos dados.

A partir do momento em que a ESP32 consegue estabelecer a conexão WebSocket, o *gateway* `DeviceGateway` no servidor registra o evento de conexão e instala o *listener* para mensagens recebidas. O sistema está então pronto para operar em regime permanente.

A **camada de apresentação** (*frontend*) não possui uma etapa de inicialização vinculada ao dispositivo embarcado. A aplicação React é carregada sob demanda quando o usuário acessa a URL do *dashboard* pelo navegador. Ao montar o componente principal, o *hook* `useCurrentReadingStream` estabelece automaticamente a conexão SSE com o *endpoint* `/current-readings/stream` do *backend*. A partir desse instante, o *frontend* começa a receber as leituras em tempo real, caso o dispositivo já esteja transmitindo dados.

3.4.2 Operação em regime permanente

Uma vez concluída a inicialização, o sistema entra em operação contínua. A cada iteração do `loop()` do *firmware*, três verificações são realizadas em sequência: a integridade da conexão Wi-Fi, a disponibilidade da conexão WebSocket e a leitura do sensor de corrente. Caso alguma das conexões esteja indisponível, o *firmware* entra em modo de reconexão bloqueante, suspendendo a coleta e o envio de dados até que a conectividade seja restaurada.

Quando ambas as conexões estão operacionais, o módulo `sct_manager` executa a medição em passagem única descrita na subseção 3.1.3: durante uma janela de 100 ms, as amostras do ADC são coletadas, o *offset* DC é rastreado continuamente pelo filtro IIR e o valor RMS da corrente é calculado. O resultado é então encapsulado em um *payload* JSON e transmitido ao servidor via WebSocket. Como cada janela de medição produz diretamente uma leitura, o ciclo consome apenas 100 ms, resultando na transmissão de aproximadamente dez leituras por segundo.

No *backend*, cada mensagem recebida pelo `DeviceGateway` é submetida a duas etapas de validação, análise sintática JSON e verificação do esquema Zod, antes de ser encaminhada ao `CurrentReadingService`. Nesse ponto, ocorre a bifurcação em caminho duplo (*dual-path*) que caracteriza o processamento central do sistema:

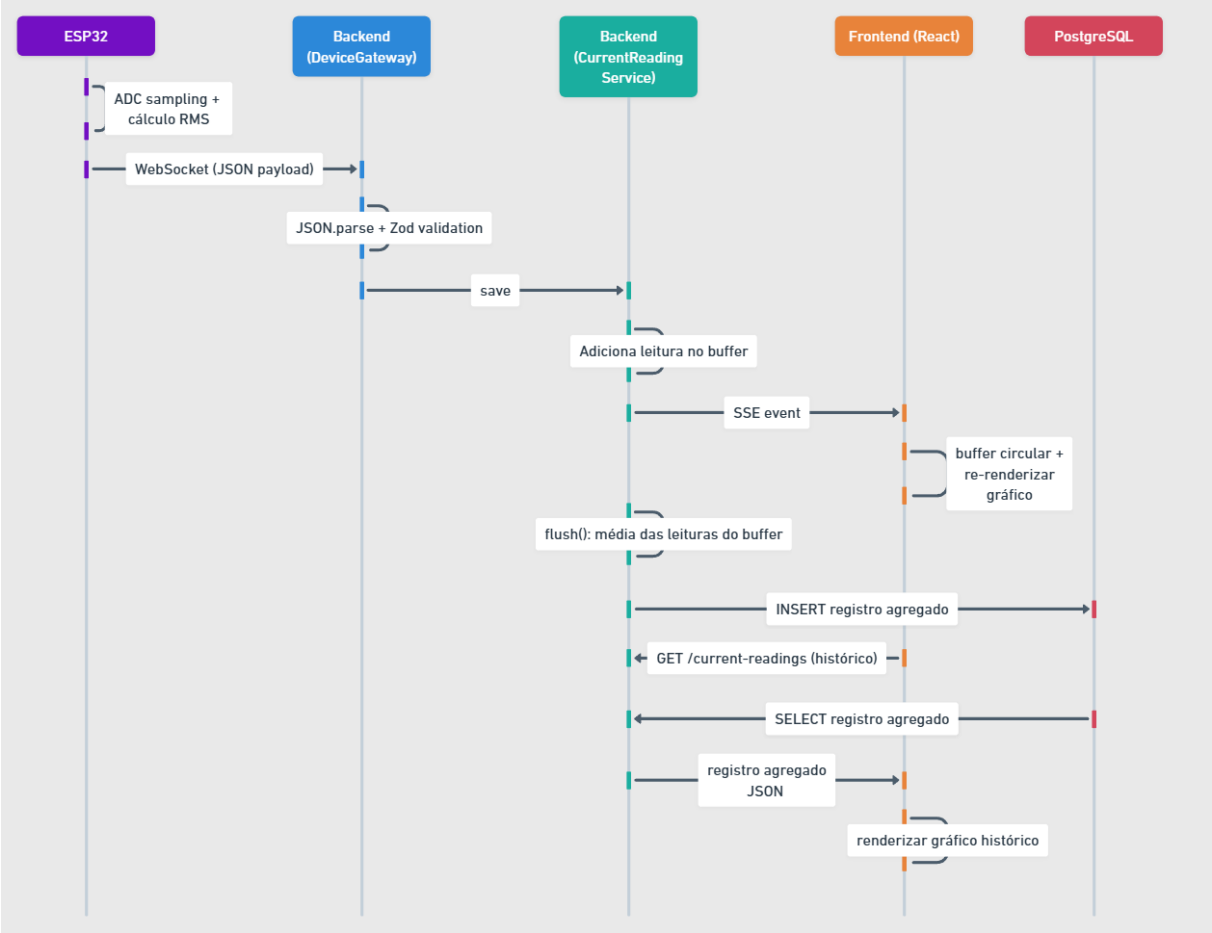
- **Caminho de tempo real:** a leitura é imediatamente publicada no `Subject` RxJS por meio de `readingsSubject.next()`, propagando-se para todos os clientes SSE conectados ao *endpoint* `/current-readings/stream`. Esse caminho não envolve acesso ao banco de dados, o que garante latência mínima entre a aquisição no sensor e a exibição na interface;
- **Caminho de persistência:** a mesma leitura é adicionada ao *buffer* em memória. A cada 10 segundos, o temporizador de agregação calcula a média aritmética de todas as leituras acumuladas na janela e persiste um único registro consolidado na tabela `current_readings` do PostgreSQL.

No *frontend*, o *hook* `useCurrentReadingStream` recebe cada evento SSE, deserializa o conteúdo JSON e o armazena em um *buffer* circular de 100 pontos. Os componentes de apresentação, painel de estatísticas e gráfico dinâmico, são re-renderizados a cada nova leitura, exibindo os valores atualizados de corrente RMS. Paralelamente, o usuário pode alternar para a aba de dados históricos e consultar os registros agregados armazenados no banco de dados por meio do *endpoint* REST `GET /current-readings`, com suporte a filtros temporais e paginação.

A Figura 14 apresenta o diagrama de sequência que sintetiza o fluxo de operação em

regime permanente, desde a aquisição da amostra de corrente até a renderização na interface do usuário. Primeiro, o dispositivo embarcado realiza a leitura das amostras de corrente (sampling) e realiza o cálculo do valor RMS, em seguida envia esses dados via *WebSocket* para a porta de entrada do *backend* (*DeviceGatway*), em que são realizadas as validações dos dados e a chamada do método *save* do serviço de leitura de corrente (*CurrentReadingService*). O serviço, por sua vez, adiciona a leitura ao *buffer* em memória e publica o evento para os clientes SSE conectados. No *frontend*, o *hook* de consumo do *stream* SSE processa os dados recebidos, atualiza o *buffer* circular e aciona a re-renderização dos componentes de estatísticas e gráfico. Periodicamente, o serviço de leitura de corrente também executa a função de flush do *buffer*, calculando a média das leituras acumuladas e persistindo um registro agregado no banco de dados. Sob demanda, o usuário pode consultar os dados históricos por meio da rota */current-readings*, o serviço de leituras por sua vez faz uma query SELECT bo banco e envia os dados em formato JSON para a interface do usuário, resultando na renderização do gráfico histórico com os registros retornados.

Figura 14 – Diagrama de sequência do fluxo de operação em regime permanente da plataforma SEMS



Fonte: elaborado pelo autor (2026).

3.4.3 Tolerância a falhas de conectividade

O sistema incorpora mecanismos de reconexão automática em todas as interfaces de comunicação, conferindo robustez diante de interrupções transitórias na rede.

No dispositivo embarcado, a perda da conexão Wi-Fi é detectada a cada iteração do `loop()` por meio da verificação do estado da interface de rede. Quando identificada, o módulo `wifi_manager` desconecta a interface, aguarda 5 segundos e reinicia o processo de associação à rede, bloqueando a execução até a reconexão. A mesma lógica é aplicada à conexão WebSocket pelo módulo `ws_manager`: a indisponibilidade do canal é detectada pela ausência de resposta do método `available()`, e a reconexão é tentada em intervalos de 5 segundos. Durante o período de reconexão, a coleta e o envio de dados ficam suspensos, mas nenhuma leitura anterior é perdida no lado do servidor, pois o *buffer* de agregação retém os dados já recebidos.

No *frontend*, o *hook* `useCurrentReadingStream` implementa reconexão automática ao *stream* SSE. Quando o manipulador `onerror` do `EventSource` é acionado, a instância corrente é encerrada, o estado de conexão é atualizado para desconectado, refletido visualmente pelo *badge* "Desconectado" no cabeçalho, e uma nova tentativa de conexão é agendada após 3 segundos. O *badge* retorna ao estado "Conectado" assim que o *stream* é restabelecido, fornecendo *feedback* imediato ao usuário.

3.5 Ferramentas e tecnologias utilizadas

Esta seção consolida as principais ferramentas, linguagens, bibliotecas e plataformas empregadas no desenvolvimento da plataforma SEMS. Embora cada tecnologia tenha sido apresentada em contexto ao longo das seções anteriores, a reunião dessas informações em um único ponto de referência facilita a reprodutibilidade do projeto e permite ao leitor identificar rapidamente a *stack* tecnológica adotada em cada camada.

3.5.1 Camada embarcada

O *firmware* do dispositivo de aquisição foi desenvolvido em C++ sobre o *framework* Arduino para ESP32, utilizando a Arduino IDE como ambiente de compilação e gravação. A escolha desse *framework* decorre da ampla disponibilidade de bibliotecas compatíveis e da extensa documentação comunitária, fatores que aceleram o ciclo de prototipagem sem comprometer o acesso aos periféricos de baixo nível do microcontrolador. O Quadro 7 relaciona os componentes

de *hardware* e *software* empregados nessa camada.

Quadro 7 – Ferramentas e componentes da camada embarcada

Componente	Versão/Modelo	Finalidade
ESP32 (Espressif)	SoC dual-core Xtensa LX6	Microcontrolador com Wi-Fi integrado e ADC de 12 bits
SCT-013-100	—	Transformador de corrente não invasivo (até 100 A)
Arduino *Framework* (ESP32)	—	Camada de abstração de *hardware* e ambiente de desenvolvimento
ArduinoWebsockets	—	Cliente WebSocket conforme RFC 6455 para microcontroladores
ArduinoJson	—	Serialização e desserialização de *payloads* JSON com alocação estática
WiFi.h (ESP32 core)	—	Gerenciamento da interface Wi-Fi em modo estação (STA)

Fonte: elaborado pelo autor (2026).

3.5.2 Camada de servidor (backend)

O servidor foi construído com o *framework* NestJS na versão 11, executado sobre o ambiente Node.js 22. O TypeScript, na versão 5.9, é utilizado como linguagem única, tanto para o código de aplicação quanto para as migrações de banco de dados. O gerenciamento de pacotes é realizado pelo Yarn 4 (Berry), configurado com o *linker* `node-modules`.

A comunicação em tempo real com o dispositivo embarcado emprega o módulo `ws`, uma implementação leve do protocolo WebSocket, integrado ao NestJS por meio do pacote `@nestjs/platform-ws`. A persistência dos dados é realizada pelo ORM TypeORM conectado a um banco de dados PostgreSQL 17, e a validação de dados nas fronteiras de entrada utiliza a biblioteca Zod na versão 4. O sistema de *logging* estruturado é provido pela biblioteca Pino, com formatação legível em desenvolvimento (`pino-pretty`) e saída JSON em produção. O Quadro 8 apresenta as principais dependências do servidor.

3.5.3 Camada de apresentação (frontend)

A aplicação *web* foi desenvolvida com React 19 e Vite 8, conforme detalhado na subseção 3.3.1. O gerenciamento de pacotes utiliza o Yarn em modo *Plug'n'Play* (PnP), que substitui a pasta `node_modules` por um cache compactado, reduzindo o consumo de disco. A estilização segue a abordagem *utility-first* do Tailwind CSS 4, complementada pela biblioteca de componentes shadcn/ui e pelo sistema de variantes CVA (`class-variance-authority`). O Quadro 5, apresentado anteriormente, relaciona as dependências do *frontend* com suas respectivas

Quadro 8 – Principais dependências do *backend* e suas finalidades

Dependência	Versão	Finalidade
NestJS (`@nestjs/core`)	11.1	*Framework* modular com suporte nativo a WebSocket e SSE
TypeScript	5.9	Tipagem estática sobre JavaScript
Node.js	22.18	Ambiente de execução do servidor
TypeORM	0.3	Mapeamento objeto-relacional e migrações
PostgreSQL	17.5	Sistema gerenciador de banco de dados relacional
Zod	4.3	Validação e *parsing* de esquemas em tempo de execução
RxJS	7.8	Programação reativa (barramento SSE via `Subject`)
Pino / pino-pretty	10.3	*Logging* estruturado de alto desempenho
`ws`	8.19	Servidor WebSocket nativo (RFC 6455)
Helmet	8.1	*Middleware* de segurança HTTP
`typeorm-transactional`	0.5	Gerenciamento declarativo de transações

Fonte: elaborado pelo autor (2026).

versões e finalidades.

3.5.4 Ferramentas de desenvolvimento e infraestrutura

Além das dependências de produção, o projeto utiliza um conjunto de ferramentas auxiliares para garantir a qualidade do código, a reprodutibilidade do ambiente e a automação de tarefas. O Quadro 9 descreve essas ferramentas.

Quadro 9 – Ferramentas de desenvolvimento e infraestrutura

Ferramenta	Versão	Finalidade
Docker	—	Conteinerização do *backend* para implantação e provisionamento do banco de dados em desenvolvimento local
Docker Compose	—	Orquestração do contêiner PostgreSQL em ambiente de desenvolvimento
ESLint	10 / 9	Análise estática de código (*linting*) com regras tipadas para TypeScript
Prettier	3.4	Formatação automática de código no *backend*
Husky	9.1	Gerenciamento de *hooks* Git (`pre-commit` executa *lint* e formatação)
Jest	30.2	*Framework* de testes unitários e de integração
Supertest	7.0	Asserções HTTP para testes de integração dos *endpoints*
Testcontainers	11.12	Provisionamento de contêineres PostgreSQL efêmeros para testes de integração
Yarn (Berry)	4.9	Gerenciador de pacotes com suporte a *workspaces* e *Plug'n'Play*
Git	—	Controle de versão do código-fonte

Fonte: elaborado pelo autor (2026).

A conteinerização do *backend* é realizada por meio de um `Dockerfile` multi-estágio baseado na imagem `node:22.18-alpine`, que instala as dependências de forma imutável (`yarn`

`install -immutable`), compila o código TypeScript e executa as migrações do banco de dados antes de iniciar o servidor. Para o ambiente de desenvolvimento local, um arquivo Docker Compose provisiona uma instância PostgreSQL 17.5 Alpine na porta 5432, eliminando a necessidade de instalação local do banco de dados.

O fluxo de qualidade de código é automatizado por um *hook* `pre-commit` gerenciado pelo Husky: a cada tentativa de *commit*, o ESLint verifica a conformidade do código com as regras de tipagem estrita e o Prettier aplica a formatação padronizada. Essa automação impede que código com erros de *lint* ou formatação inconsistente seja incorporado ao repositório.

Os testes automatizados do *backend* são organizados em dois níveis. Os testes unitários verificam a lógica isolada dos serviços de aplicação, utilizando *mocks* para as dependências de infraestrutura. Os testes de integração validam o comportamento completo dos *endpoints* HTTP e do fluxo de persistência, utilizando a biblioteca Testcontainers para provisionar uma instância PostgreSQL efêmera e descartável a cada execução, garantindo isolamento total entre os testes.

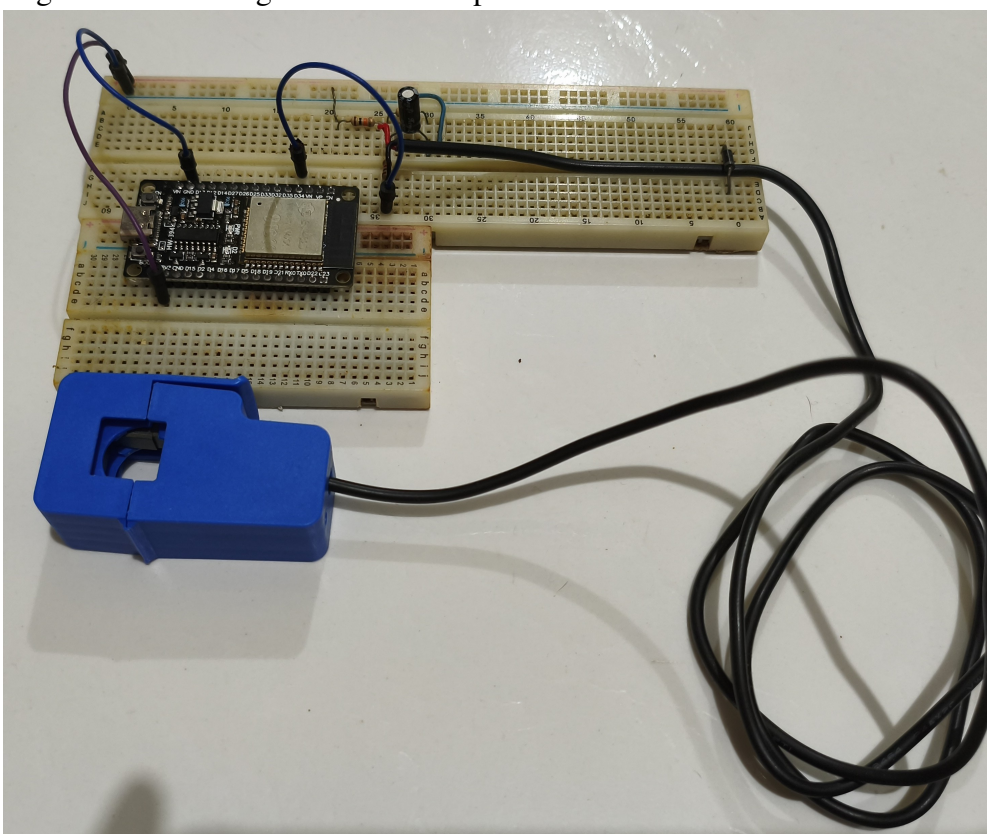
4 RESULTADOS E DISCUSSÃO

4.1 Ambiente de testes

Os testes da plataforma SEMS foram conduzidos em um ambiente composto por dois elementos principais: a montagem física do dispositivo de aquisição e a infraestrutura computacional que executa o *backend* e o *frontend*. Esta seção descreve cada um desses elementos, de modo a permitir a reprodutibilidade dos resultados apresentados nas seções subsequentes.

O dispositivo embarcado consiste em um microcontrolador ESP32 montado sobre uma *protoboard*, conectado ao circuito de condicionamento de sinal descrito na subseção 3.1.1. O sensor de corrente SCT-013-100 é acoplado ao circuito por meio de um cabo com comprimento aproximado de 60 cm, suficiente para posicionar o TC no condutor do equipamento monitorado sem a necessidade de deslocar a placa de prototipagem. O equipamento utilizado como carga de teste foi um ventilador com potência nominal de 96 W; o fator de potência do equipamento não é informado pelo fabricante. A Figura 15 apresenta a montagem física utilizada durante os ensaios.

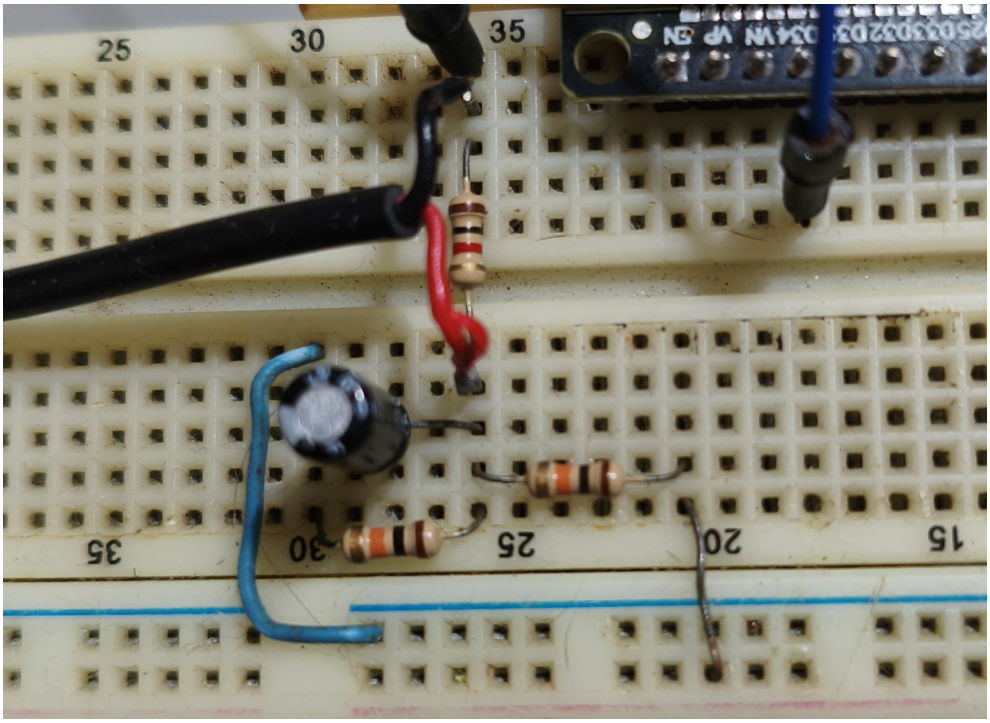
Figura 15 – Montagem física do dispositivo embarcado utilizado nos testes



Fonte: elaborado pelo autor (2026).

A Figura 16 apresenta a montagem física do circuito de polarização sobre a protoboard. Na configuração ilustrada, observa-se dois resistores de 10 kΩ conectados em série entre si e em paralelo com a fonte de alimentação de 3,3 V da ESP32, ambos conectados nas trilhas de energia da protoboard. Um capacitor estabilizador encontra-se conectado em paralelo com o resistor esquerdo, funcionando como elemento de filtragem. Adicionalmente, um resistor de *burden* de 1 kΩ está posicionado em paralelo com o transformador de corrente, conectado simultaneamente ao ponto central do divisor de tensão e à porta GPIO do microcontrolador, permitindo a conversão da corrente secundária do transformador em uma tensão adequada para aquisição analógica.

Figura 16 – Montagem detalhada do circuito de polarização (Bias DC)



Fonte: elaborado pelo autor (2026).

No lado computacional, o *backend* NestJS e o *frontend* React foram executados em uma mesma máquina local com sistema operacional Windows, utilizando o *Windows Subsystem for Linux* (WSL) como camada de virtualização para o ambiente Linux. A escolha pelo WSL justifica-se pela compatibilidade nativa do ecossistema Node.js com distribuições Linux, o que proporciona maior estabilidade na execução de ferramentas como Docker e gerenciadores de pacotes. O banco de dados PostgreSQL foi provisionado como contêiner Docker dentro do próprio WSL.

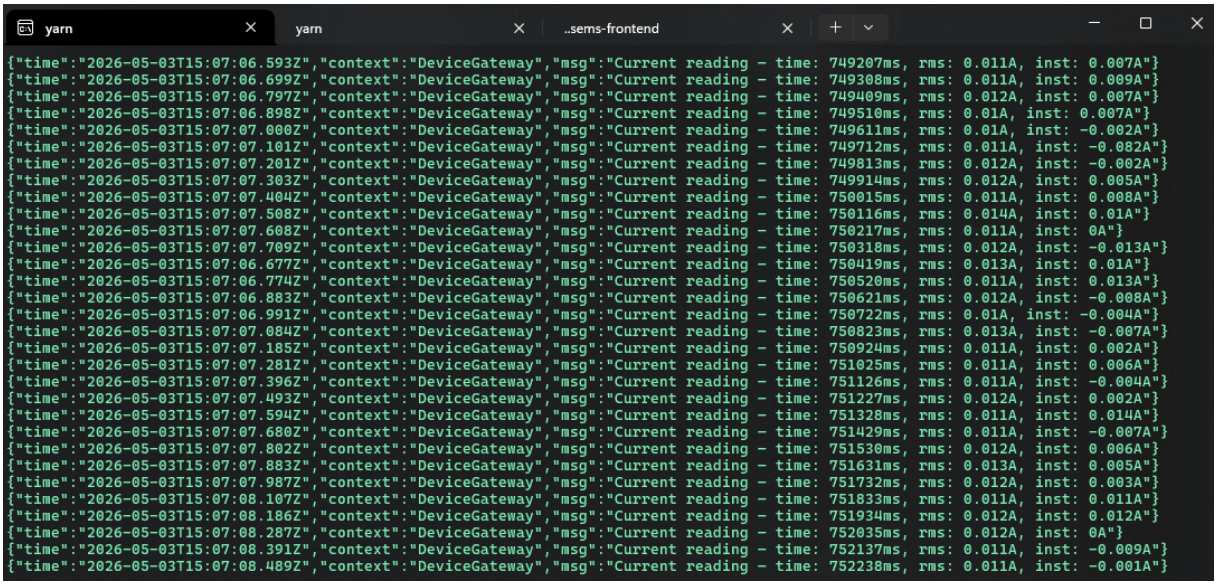
Uma particularidade da configuração adotada é que o WSL opera com um endereço

Internet Protocol (IP) interno distinto do endereço da máquina hospedeira no Windows. Assim, para que a ESP32, conectada à rede Wi-Fi local, pudesse alcançar o servidor NestJS executado dentro do WSL, foi necessário redirecionar a porta 3000 do sistema Windows para o endereço IP atribuído à instância WSL. O endereço IP local da máquina hospedeira foi inserido diretamente no código-fonte do *firmware* (no arquivo `config.h`), uma abordagem adequada para o cenário de testes, embora não recomendável para implantações em ambiente de produção.

A interface *web* foi acessada por meio do navegador Brave, no qual foram verificadas tanto as leituras em tempo real, exibidas na aba de monitoramento com gráfico dinâmico e *cards* de estatísticas, quanto a consulta de dados históricos armazenados no banco de dados.

A Figura 17 apresenta os registros de logs em tempo real gerados pelo servidor *backend*. Nela, é possível observar os eventos de conexão e desconexão do microcontrolador, bem como os valores de corrente RMS recebidos via WebSocket, incluindo *timestamps* e metadados das medições que validam a transmissão adequada dos dados de sensoriamento para o servidor.

Figura 17 – Logs em tempo real do servidor backend com registros de corrente via WebSocket do microcontrolador



Fonte: elaborado pelo autor (2026).

4.2 Calibração e validação do sensor de corrente

A calibração implementada no *firmware*, descrita na subseção 3.1.3, foi avaliada em um ensaio prático com uma carga real. Para essa validação, utilizou-se um ventilador doméstico

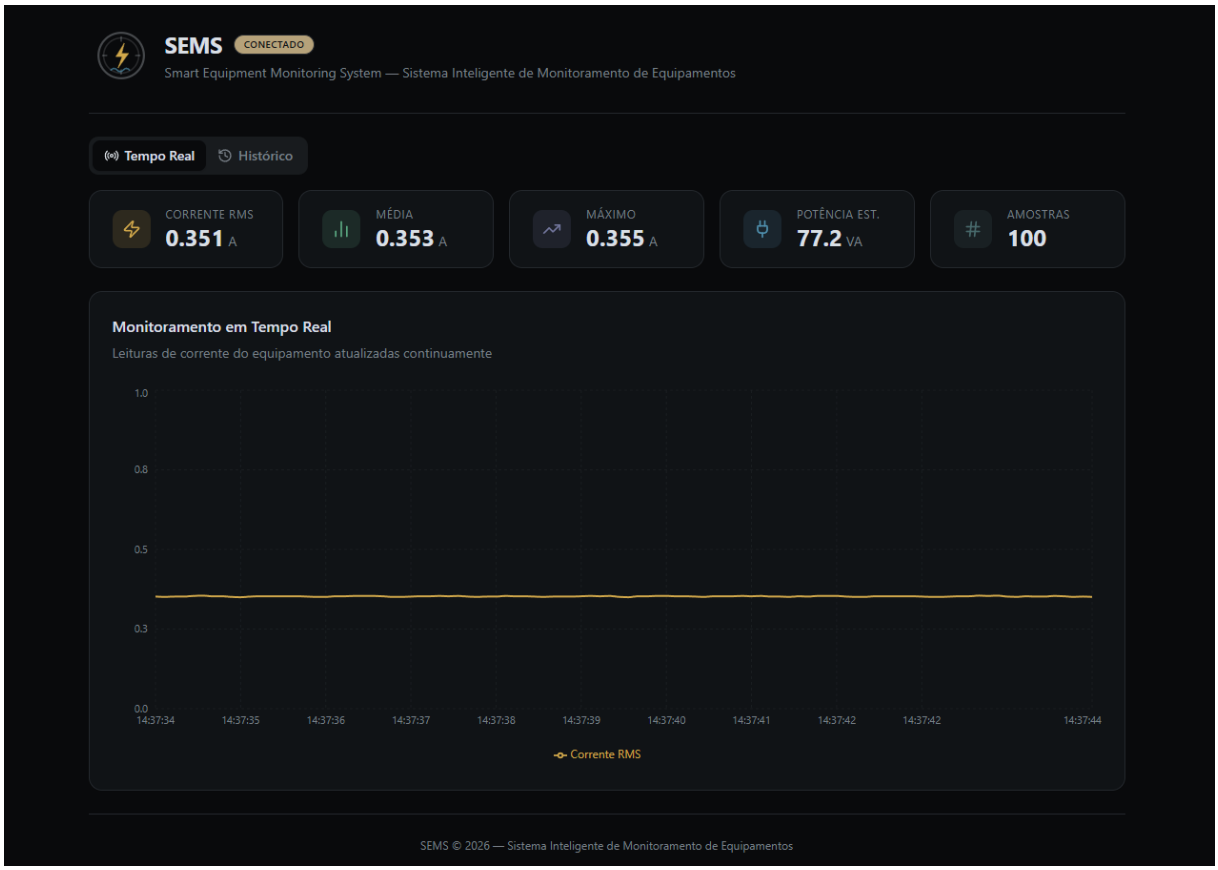
com potência nominal de 96 W, acionado na menor das três velocidades disponíveis.

Conforme mostrado na Figura 18, o valor de corrente RMS permaneceu estável em torno de 0,353 A durante a operação em regime permanente, com leitura de 0,351 A no instante registrado. Esse valor de corrente sob uma tensão alternada de 220 V corresponde a uma potência aparente de aproximadamente 77,2 VA.

Essa leitura é coerente com a potência nominal do equipamento (96 W), considerando o fator de potência e o fato de que a medição corresponde ao nível mais baixo de velocidade do ventilador. Portanto, os resultados obtidos validam a precisão do circuito de condicionamento de sinal e da rotina de calibração implementada no *firmware*.

Cabe observar que uma calibração metrológica mais rigorosa exigiria a comparação sistemática das medições com um instrumento de referência do tipo *true RMS*. No entanto, dentro do escopo deste trabalho, os resultados obtidos demonstram adequação aos objetivos propostos, pois comprovam a capacidade da plataforma em registrar leituras estáveis e coerentes com o comportamento esperado de cargas indutivas reais. A validação realizada com carga conhecida confirma que o sensor de corrente está devidamente calibrado para monitoramento remoto em tempo real.

Figura 18 – Leituras de corrente RMS obtidas durante o monitoramento do ventilador em baixa velocidade



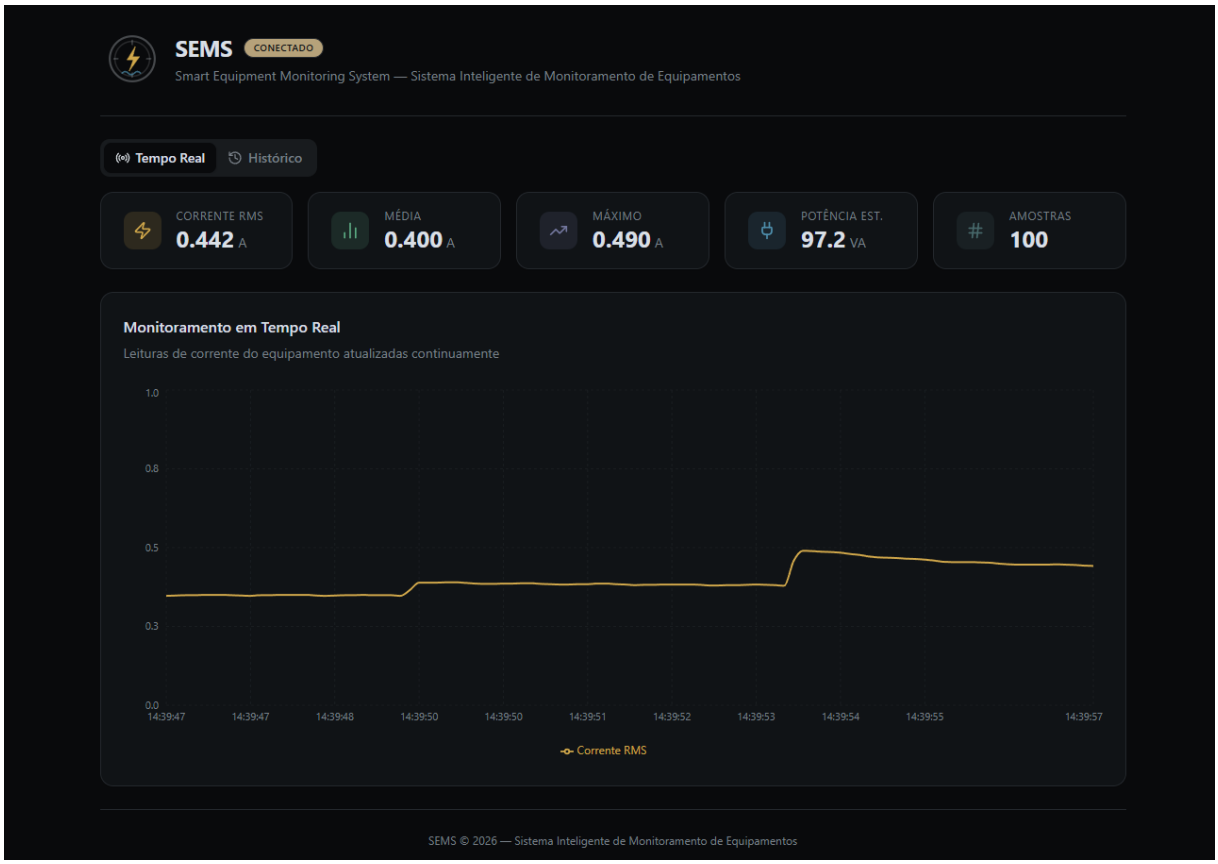
Fonte: elaborado pelo autor (2026).

Na Figura 19, observa-se a detecção da variação da corrente durante a transição entre os três estágios de operação do motor. Considerando a tensão nominal da rede igual a 220 V e assumindo que o fator de potência do ventilador é suficientemente elevado para aproximar a potência aparente da potência ativa, é possível estimar a potência consumida em cada estágio e comparar os resultados com as especificações do equipamento.

No estágio inicial, correspondente à menor velocidade, o motor opera com uma corrente em torno de 350 mA, o que resulta em uma potência aparente de aproximadamente 77 VA. Em seguida, ao habilitar o segundo estágio, observa-se um aumento gradual da corrente para cerca de 400 mA, equivalente a aproximadamente 88 VA. Por fim, na velocidade máxima, a corrente atinge valores próximos de 440 mA, resultando em uma potência de aproximadamente 97 VA.

Esse valor está em concordância com a potência nominal de 96 W especificada no equipamento. Considerando o fator de potência e possíveis incertezas associadas à calibração do sistema de medição, os resultados obtidos apresentam boa aderência às especificações, validando a consistência das medições realizadas.

Figura 19 – Leituras de corrente RMS obtidas durante variação dos 3 estágios de velocidade do motor

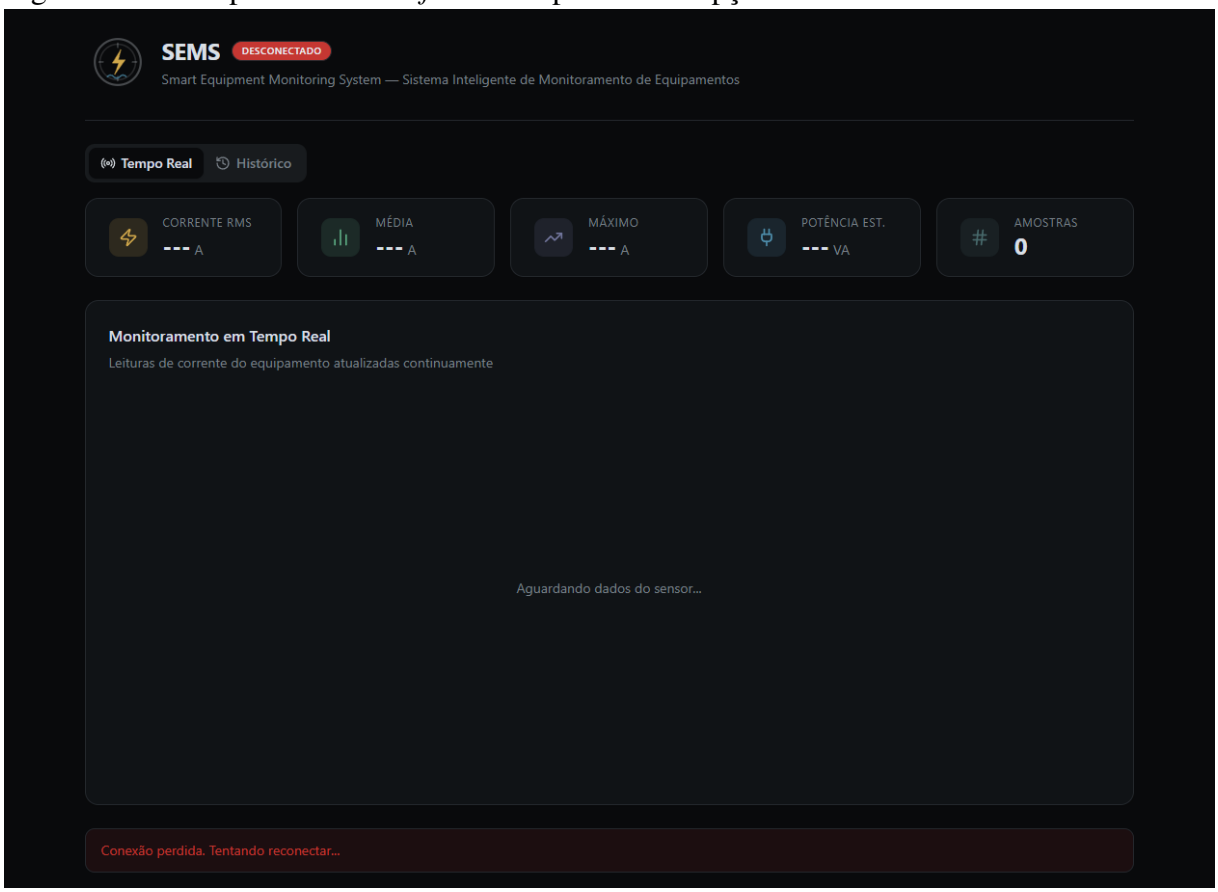


Fonte: elaborado pelo autor (2026).

4.3 Tolerância a falhas e reconexão

Com o objetivo de avaliar a robustez da interface *web* diante da indisponibilidade do servidor, realizou-se um teste de falha controlada mediante o desligamento do *backend* NestJS durante a execução normal da aplicação cliente. Esse ensaio permitiu observar, de forma direta, como o *frontend* reage à interrupção do fluxo de dados em tempo real e quais mecanismos de recuperação são oferecidos ao usuário.

Figura 20 – Comportamento do *frontend* após a interrupção do servidor NestJS



Fonte: elaborado pelo autor (2026).

Conforme apresentado na Figura 20, a interrupção do servidor não provoca o encerramento nem o travamento da interface. Em vez disso, o *frontend* permanece operacional e sinaliza explicitamente a perda de conectividade. Observa-se, no cabeçalho da aplicação, a alteração do indicador de estado para "Desconectado", além da exibição de um banner na parte inferior da tela com a mensagem "Conexão perdida. Tentando reconectar...". Dessa forma, o usuário é informado de maneira imediata sobre a condição do sistema, sem ambiguidade quanto à origem da ausência de novas leituras.

Além da sinalização visual, verifica-se que a aplicação cliente continua tentando restabelecer a comunicação automaticamente, sem exigir recarregamento manual da página ou reinicialização da interface. Durante esse intervalo, os componentes de visualização permanecem carregados, ainda que sem atualização dos dados em tempo real. Tal comportamento demonstra que a camada de apresentação foi projetada de forma desacoplada em relação ao servidor, preservando sua operação mesmo quando o serviço de dados se torna temporariamente indisponível.

Esse resultado evidencia uma característica importante da arquitetura proposta: a

independência entre os serviços. Embora o *backend* seja responsável pelo recebimento, processamento e distribuição das leituras, sua interrupção não invalida a execução do *frontend*, que continua apto a informar o estado da conexão e a aguardar a retomada do serviço. Em termos práticos, essa separação amplia a flexibilidade de operação da plataforma, pois permite reinicializações, manutenções ou falhas transitórias em uma camada sem comprometer integralmente as demais.

5 CONCLUSÃO

O monitoramento de grandezas elétricas em dispositivos de baixa potência constitui uma necessidade crescente em ambientes residenciais e industriais. Entretanto, as soluções existentes frequentemente carecem de acessibilidade devido ao alto custo, à complexidade de implementação ou à dependência de plataformas proprietárias, criando barreiras para a adoção de sistemas de medição e análise em tempo real. Este trabalho apresentou o desenvolvimento de uma plataforma IoT *web* de código aberto, denominada SEMS, com objetivo de disponibilizar aquisição, processamento e visualização de corrente elétrica em eletrodomésticos de baixa potência de forma acessível, escalável e facilmente replicável.

A plataforma foi estruturada em três camadas funcionais, seguindo uma arquitetura distribuída que promove modularidade e extensibilidade. Na camada de aquisição, implementou-se um dispositivo embarcado baseado no microcontrolador ESP32 acoplado ao sensor não invasivo SCT-013-100, dotado de circuito de condicionamento de sinal e firmware que realiza o cálculo de valor eficaz (RMS) da corrente a cada 100 ms, transmitindo os dados ao servidor por meio de protocolo WebSocket. Na camada intermediária, desenvolveu-se um servidor *backend* em NestJS que implementa uma arquitetura de dual-path: simultaneamente, fornece dados em tempo real aos clientes via SSE com latência mínima, e agrega as leituras em janelas de 10 segundos antes de persistir em banco de dados PostgreSQL, reduzindo a carga de escrita em aproximadamente 99%. Na camada de apresentação, implementou-se uma interface *web* em React que permite tanto a visualização em tempo real do consumo de corrente, por meio de gráficos dinâmicos e indicadores visuais, quanto a consulta de séries históricas com suporte a filtros temporais e paginação.

Os ensaios de validação realizados com um ventilador doméstico de 96 W demonstraram que a plataforma atinge com êxito os objetivos propostos. O circuito de aquisição e o firmware da ESP32 funcionaram de forma estável, calibrando-se automaticamente e fornecendo leituras de corrente coerentes com as especificações do equipamento testado: em operação na menor velocidade, o valor medido de 0,353 A corresponde a uma potência aparente de 77 VA, alinhado ao esperado; na velocidade máxima, 0,440 A resultou em potência de 97 VA, em acordo com a nominal. O servidor *backend* recebeu continuamente os dados em tempo real, executou a agregação temporal conforme projetado e armazenou os registros consolidados no banco de dados sem perdas. A interface *frontend* consumiu com sucesso o *stream* SSE para exibição instantânea, bem como a API REST para recuperação de dados históricos, permitindo o usuário

acompanhar o comportamento da carga em diferentes escalas temporais. Adicionalmente, os testes de tolerância a falhas comprovaram que a plataforma mantém a operação da camada de apresentação mesmo durante indisponibilidade temporária do *backend*, com mecanismos automáticos de reconexão que informam claramente ao usuário o estado da conectividade.

A arquitetura distribuída e modular da plataforma demonstrou atender tanto aos requisitos de funcionalidade quanto aos de manutenibilidade e extensibilidade. A separação em camadas independentes permite que cada componente seja desenvolvido, testado e eventualmente substituído isoladamente, e o uso exclusivo de tecnologias de código aberto e amplamente documentadas (ESP32, NestJS, React, PostgreSQL) garante que a solução seja facilmente replicável em diferentes contextos. Os resultados obtidos validam a viabilidade da abordagem proposta como alternativa de baixo custo para monitoramento de consumo elétrico em dispositivos de baixa potência.

5.1 Limitações e trabalhos futuros

Apesar dos resultados obtidos, o projeto apresenta algumas limitações. As leituras de corrente não foram comparadas com um equipamento de referência, então não é possível garantir a precisão absoluta dos valores medidos. Além disso, o sistema atual monitora apenas a corrente elétrica, sem contemplar outras grandezas como tensão, potência ou fator de potência.

Outra limitação está na configuração dos dispositivos embarcados. No modelo atual, o endereço IP do servidor, o SSID e a senha da rede Wi-Fi precisam ser definidos manualmente no *firmware*, o que dificulta o uso do sistema em ambientes com vários equipamentos.

Como sugestões para trabalhos futuros, destacam-se:

a) a adição de sensores para monitoramento de tensão, potência ativa, potência reativa e fator de potência;
b) a implementação de alertas automáticos para condições de operação anormais;
c) o desenvolvimento de um mecanismo de configuração automática dos dispositivos IoT, eliminando a necessidade de ajustes manuais no *firmware*;
d) o gerenciamento de múltiplos microcontroladores a partir da interface *web*;
e) a aplicação de técnicas de manutenção preditiva com base nos dados coletados;
f) a otimização do consumo de energia do dispositivo embarcado, implementando modos de economia como o modo *sleep* durante períodos sem medição ativa e outras estratégias de redução de consumo;

g) a otimização dos algoritmos de coleta e filtragem do sistema, com especial atenção ao tratamento adequado de períodos transitórios nas medições;
h) a redução dos tempos de processamento das leituras do conversor analógico-digital mediante o uso de técnicas avançadas como o filtro de Kalman, processamento em ponto fixo e transmissão do valor bruto do ADC quando apropriado;
i) a implementação de estratégias eficientes de cálculo do valor RMS na ESP32, com otimizações específicas para reduzir o tempo de processamento quando a aplicação necessitar do valor em tempo real.

REFERÊNCIAS

AL-FUQAHA, A.; GUIZANI, M.; MOHAMMADI, M.; ALEDHARI, M.; AYYASH, M. Internet of Things: a survey on enabling technologies, protocols, and applications. **IEEE Communications Surveys & Tutorials**, IEEE, v. 17, n. 4, p. 2347–2376, 2015.

ALI, M. H.; VODAPALLY, S. N. Internet-of-Things (IoT) applications in modern power grids. **Frontiers in Energy Research**, v. 14, 2026.

ASHTON, K. **That "Internet of Things" thing**. 2009. Disponível em: <https://www.rfidjournal.com/that-internet-of-things-thing>. Acesso em: 20 mar. 2026.

ATZORI, L.; IERA, A.; MORABITO, G. The Internet of Things: a survey. **Computer Networks**, Elsevier, v. 54, n. 15, p. 2787–2805, 2010.

BRAY, T. **The JavaScript Object Notation (JSON) Data Interchange Format**. 2017. RFC 8259, Internet Engineering Task Force. Disponível em: <https://datatracker.ietf.org/doc/html/rfc8259>. Acesso em: 28 mar. 2026.

CHOUDHARY, A. Internet of Things: a comprehensive overview, architectures, applications, simulation tools, challenges and future directions. 2024.

COCKBURN, A. **Crystal Clear: A Human-Powered Methodology for Small Teams**. Boston: Addison-Wesley, 2005. ISBN 978-0-201-69947-2.

DEPURU, S. S. S. R.; WANG, L.; DEVABHAKTUNI, V. Smart meters for power grid: challenges, issues, advantages and status. **Renewable and Sustainable Energy Reviews**, Elsevier, v. 15, n. 6, p. 2736–2742, 2011.

DOMÍNGUEZ-BOLAÑO, T.; CAMPOS, O.; BARRAL, V.; ESCUDERO, C. J.; GARCÍA-NAYA, J. A. An overview of IoT architectures, technologies, and existing open-source projects. **Internet of Things**, Elsevier, v. 20, 2022.

ESPRESSIF SYSTEMS. **ESP32 Series Datasheet**. Shanghai, 2025. Disponível em: <https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf>. Acesso em: 17 mai. 2026.

FETTE, I.; MELNIKOV, A. **The WebSocket Protocol**. 2011. RFC 6455, Internet Engineering Task Force. Disponível em: <https://datatracker.ietf.org/doc/html/rfc6455>. Acesso em: 28 mar. 2026.

FLANAGAN, D. **JavaScript: The Definitive Guide**. 7. ed. Sebastopol: O'Reilly Media, 2020. ISBN 978-1-491-95202-3.

FROST, B. **Atomic Design**. Brad Frost, 2016. Disponível em: <https://atomicdesign.bradfrost.com>. Acesso em: 03 mai. 2026.

INTERNATIONAL ELECTROTECHNICAL COMMISSION. **IEC 61869-2: Instrument transformers — Part 2: Additional requirements for current transformers**. Geneva, 2012.

KABALCI, Y. A survey on smart metering and smart grid communication. **Renewable and Sustainable Energy Reviews**, Elsevier, v. 57, p. 302–318, 2016.

KESTER, W. **Data Conversion Handbook**. Burlington: Elsevier/Newnes, 2005. ISBN 978-0-7506-7841-4.

LEE, E. A.; SESHIA, S. A. **Introduction to Embedded Systems: a Cyber-Physical Systems Approach**. 2. ed. Cambridge: MIT Press, 2017. ISBN 978-0-262-53381-2.

META PLATFORMS. **React Documentation**. 2024. Disponível em: <https://react.dev>. Acesso em: 29 mar. 2026.

MICROSOFT. **TypeScript Documentation**. 2024. Disponível em: <https://www.typescriptlang.org/docs/>. Acesso em: 29 mar. 2026.

MYSLIWIEC, K. **NestJS Documentation**. 2024. Disponível em: <https://docs.nestjs.com>. Acesso em: 29 mar. 2026.

NANDI, S.; TOLIYAT, H. A.; LI, X. Condition monitoring and fault diagnosis of electrical motors — a review. **IEEE Transactions on Energy Conversion**, IEEE, v. 20, n. 4, p. 719–729, 2005.

NILSSON, J. W.; RIEDEL, S. A. **Electric Circuits**. 10. ed. Upper Saddle River: Pearson, 2014. ISBN 978-0-13-376003-3.

OPEN ENERGY MONITOR. **CT sensors — Introduction**. 2016. Disponível em: <https://docs.openenergymonitor.org/electricity-monitoring/ct-sensors/introduction.html>. Acesso em: 22 mar. 2026.

OPEN ENERGY MONITOR. **EmonLib — Arduino library for energy monitoring**. 2024. Disponível em: <https://github.com/openenergymonitor/EmonLib>. Acesso em: 05 mai. 2026.

RoboCore. **Sensor de Corrente Não Invasivo 100A SCT-013**. 2026. Disponível em: <https://www.robocore.net/sensor-corrente-tensao/sensor-de-corrente-nao-invasivo-100a-sct-013>. Acesso em: 19 abr. 2026.

STATISTA. **Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030**. 2024. Disponível em: <https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/>. Acesso em: 15 mar. 2026.

THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. **PostgreSQL 16 Documentation**. 2024. Disponível em: <https://www.postgresql.org/docs/16/>. Acesso em: 29 mar. 2026.

TYPEORM. **TypeORM — Official Documentation**. 2024. Disponível em: <https://typeorm.io>. Acesso em: 29 mar. 2026.

WEBSTER, J. G.; EREN, H. **Measurement, Instrumentation, and Sensors Handbook**. 2. ed. Boca Raton: CRC Press, 2014. ISBN 978-1-4398-4888-3.

WHATWG. **HTML Living Standard — Server-sent events**. 2024. Disponível em: <https://html.spec.whatwg.org/multipage/server-sent-events.html>. Acesso em: 28 mar. 2026.

APÊNDICE A – CÓDIGO-FONTE DO *FIRMWARE*: PONTO DE ENTRADA (`MAIN.INO`)

Código-fonte 1 – Ponto de entrada do *firmware* (`main.ino`)

```
#include "wifi_manager.h"
#include "ws_manager.h"
#include "sct_manager.h"

void setup() {
  Serial.begin(115200);
  wifi_connect();
  ws_connect();
  sct_init();
}

void loop() {
  wifi_check_connection();
  ws_check_connection();
  sct_read_and_send();
}
```

APÊNDICE B – CÓDIGO-FONTE DO *FIRMWARE*: FUNÇÃO `SCT_READ_AND_SEND`

Código-fonte 2 – Função de medição em passagem única (`sct_read_and_send`)

```
void sct_read_and_send() {
  const uint32_t window_start = millis();
  double   sum_sq = 0.0;
  uint32_t count  = 0;

  while (millis() - window_start < SCT_SAMPLE_WINDOW_MS) {
    int raw = analogRead(SCT_ADC_PIN);
    s_offset += ((double)raw - s_offset) / 1024.0;
    float v_ac = ((double)raw - s_offset)
                 * (SCT_VREF / SCT_ADC_COUNTS);
    sum_sq += (double)v_ac * v_ac;
    count++;
  }

  float irms = 0.0f;
  if (count > 0) {
    irms = sqrtf((float)(sum_sq / count)) * SCT_ICAL;
  }

  StaticJsonDocument<96> doc;
  doc["type"]         = "current";
  doc["time"]         = millis();
  doc["current_rms"]  = roundf(irms * 1000.0f) / 1000.0f;

  char buf[96];
  serializeJson(doc, buf, sizeof(buf));
  ws_send_json(buf);
}
```

APÊNDICE C – ESTRUTURA DO *PAYLOAD* JSON TRANSMITIDO PELA ESP32

Código-fonte 3 – Estrutura do *payload* JSON transmitido pela ESP32

```
{
  "type":          "current",
  "time":          <millis>,
  "current_rms":   <float>,
}
```

APÊNDICE D – CÓDIGO-FONTE DO *BACKEND*: PONTO DE ENTRADA (`MAIN.TS`)

Código-fonte 4 – Ponto de entrada do *backend* (`main.ts`)

```
import { NestFactory } from '@nestjs/core';
import { WsAdapter } from '@nestjs/platform-ws';
import helmet from 'helmet';
import cookieParser from 'cookie-parser';
import { AppModule } from './app.module';
import { ConfigService } from '@config/config.service';
import { Logger } from '@modules/shared/infra/logging/
   logger';
import { GlobalExceptionFilter }
  from '@modules/shared/infra/api/filters/global-exception.
     filter';
import { RequestValidationFilter }
  from '@modules/shared/infra/api/filters/request-
     validation.filter';
import {
  initializeTransactionalContext,
  StorageDriver,
} from 'typeorm-transactional';

async function bootstrap() {
  initializeTransactionalContext({
    storageDriver: StorageDriver.AUTO,
  });

  const app = await NestFactory.create(AppModule, {
    logger: Logger,
  });
  const configService = app.get(ConfigService);

```

```
  app.useWebSocketAdapter(new WsAdapter(app));
  app.use(helmet());
  app.use(cookieParser());
  app.enableCors({
    credentials: true,
    origin: configService.getRequired<string[]>('
       CORS_ORIGINS'),
  });
  app.useGlobalFilters(
    new GlobalExceptionFilter(),
    new RequestValidationFilter(),
  );

  await app.listen(process.env['PORT'] ?? 3000, '0.0.0.0');
}

void bootstrap();
```

APÊNDICE E – CÓDIGO-FONTE DO *BACKEND*: CURRENTREADINGSERVICE

Código-fonte 5 – Serviço de leituras de corrente (*current-reading.service.ts*)

```
@Injectable()
export class CurrentReadingService
  implements OnModuleInit, OnModuleDestroy
{
  private readonly windowMs: number;
  private buffer: BufferedReading[] = [];
  private flushTimer: ReturnType<typeof setInterval> | null
    = null;
  private readonly readingsSubject
    = new Subject<RawReadingEvent>();
  readonly readings$: Observable<RawReadingEvent>
    = this.readingsSubject.asObservable();

  constructor(
    @Inject(CURRENT_READING_REPOSITORY_TOKEN)
    private readonly currentReadingRepository:
      ICurrentReadingRepository,
    @Inject(ID_GENERATION_SERVICE_TOKEN)
    private readonly idGenerationService:
      IIdGenerationService,
    private readonly configService: ConfigService,
  ) {
    this.windowMs = this.configService.getRequired<number>(
      'CURRENT_READING_WINDOW_MS',
    );
  }

  onModuleInit(): void {
    this.flushTimer = setInterval(() => {
```

```
      void this.flush();
    }, this.windowMs);
  }

  onModuleDestroy(): void {
    if (this.flushTimer !== null) {
      clearInterval(this.flushTimer);
    }
    this.readingsSubject.complete();
  }

  save(
    time: number,
    currentRms: number,
  ): void {
    this.buffer.push({ time, currentRms });
    this.readingsSubject.next({
      timeMs: time, currentRms,
    });
  }

  private async flush(): Promise<void> {
    if (this.buffer.length === 0) return;

    const snapshot = this.buffer;
    this.buffer = [];

    const count = snapshot.length;
    const avgTime = snapshot.reduce(
      (sum, r) => sum + r.time, 0) / count;
    const avgRms = snapshot.reduce(
      (sum, r) => sum + r.currentRms, 0) / count;
```

```
62
63      const reading = new CurrentReading(
64        this.idGenerationService.newId(),
65        Math.round(avgTime),
66        avgRms,
67        new Date(),
68        new Date(),
69      );
70
71      await this.currentReadingRepository.save(reading);
72    }
73  }
```

APÊNDICE F – MIGRAÇÃO DE CRIAÇÃO DA TABELA CURRENT_READINGS

Código-fonte 6 – Migração de criação da tabela current_readings

```
export class CreateCurrentReadingsTable1774297559483
  implements MigrationInterface
{
  public async up(queryRunner: QueryRunner): Promise<void>
     {
    await queryRunner.query(`
      CREATE TABLE "public"."current_readings" (
        "id"           uuid          NOT NULL,
        "time_ms"      integer       NOT NULL,
        "current_rms"  numeric(10,3) NOT NULL,
        "createdAt"    timestamptz   NOT NULL
                       DEFAULT CURRENT_TIMESTAMP,
        "updatedAt"    timestamptz   NOT NULL
                       DEFAULT CURRENT_TIMESTAMP,
        CONSTRAINT "PK_current_readings"
                       PRIMARY KEY ("id")
      )
    `);
  }

  public async down(queryRunner: QueryRunner): Promise<void
     > {
    await queryRunner.query(
      `DROP TABLE "public"."current_readings"`
    );
  }
}
```

APÊNDICE G – ESTRUTURA DA RESPOSTA DO *ENDPOINT* `GET /CURRENT-READINGS`

Código-fonte 7 – Estrutura da resposta do *endpoint* `GET /current-readings`

```
{
  "data": [
    {
      "id": "a1b2c3d4-...",
      "timeMs": 1234567890,
      "currentRms": 2.345,
      "createdAt": "2026-03-20T14:30:00.000Z"
    }
  ],
  "total": 8640,
  "limit": 100,
  "offset": 0
}
```

APÊNDICE H – IMPLEMENTAÇÃO DO *STREAM* SSE NO CONTROLADOR DE LEITURAS

Código-fonte 8 – Implementação do *stream* SSE no controlador de leituras

```
@Controller('current-readings')
export class CurrentReadingController {
  constructor(
    private readonly currentReadingService:
      CurrentReadingService,
  ) {}

  @Sse('stream')
  stream(): Observable<MessageEvent> {
    return this.currentReadingService.readings$.pipe(
      map((reading) =>
        ({ data: reading }) satisfies MessageEvent,
      ),
    );
  }
}
```