Jadson de Souza Maciel

Desenvolvimento de protótipo para detecção de vazamento de gás liquefeito de petróleo (GLP)

Manaus - AM

2025

Jadson de Souza Maciel

Desenvolvimento de protótipo para detecção de vazamento de gás liquefeito de petróleo (GLP)

Trabalho de Conclusão de Curso apresentado à banca avaliadora do curso de Engenharia de Controle e Automação da Escola Superior de Tecnologia da Universidade do Estado do Amazonas como pré-requisito para obtenção do título de Engenheiro de Controle e Automação.

Universidade do Estado do Amazonas – UEA

Escola Superior de Tecnologia – EST

Engenharia de Controle e Automação

Orientador: Almir Kimura Junior, Dr.

Manaus - AM

2025

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

Sistema Integrado de Bibliotecas da Universidade do Estado do Amazonas.

M152d Maciel, Jadson de Souza

Desenvolvimento de protótipo para detecção de vazamento de gás liquefeito de petróleo (GLP) / Jadson de Souza Maciel . Manaus : [s.n], 2025.

108 f.: color.; 21,0 cm.

TCC - Graduação em Engenharia de Controle e Automação- Universidade do Estado do Amazonas, Manaus, 2025.

Inclui Bibliografia.

Inclui Apêndice.

Orientador: Kimura Junior, Almir.

1. Monitoramento de gás. 2. IoT. 3. Impressão 3D. 4. ESP32. 5. Segurança. I. Kimura Junior, Almir (Orient.) II. Universidade do Estado do Amazonas. III. Título

CDU(1997)681.5

Jadson de Souza Maciel

Desenvolvimento de protótipo para detecção de vazamento de gás liquefeito de petróleo (GLP)

Trabalho de Conclusão de Curso apresentado à banca avaliadora do curso de Engenharia de Controle e Automação da Escola Superior de Tecnologia da Universidade do Estado do Amazonas como pré-requisito para obtenção do título de Engenheiro de Controle e Automação.

Trabalho aprovado. Brasil, 13 de junho de 2025:

Luiz Alberto Queiroz Cordovil Júnior

Prof. Dr. Luiz Alberto Queiroz Cordovil Júnior
Sidia

Rodrigo Farias Araújo

Prof. Dr. Rodrigo Farias Araújo
Universidade do Estado do Amazonas (UEA)

Vitor Fernando de Souza Gadelha

Prof. Vitor Fernando de Souza Gadelha
Instituto Federal do Amazonas (IFAM)

Manaus - AM
2025

Agradecimentos

Gostaria de expressar meus agradecimentos a Deus, pela oportunidade de viver e pela força concedida para concluir esta jornada.

Agradeço à minha família, que está comigo desde o meu nascimento, sempre me proporcionando tudo de melhor que puderam. Em especial, aos meus pais, Jadilson Maciel e Nilcicleia de Souza; à minha avó, Luzia Cordeiro; e ao meu irmão, Leonardo de Souza. Todos foram fundamentais para que eu pudesse chegar até aqui.

Agradeço à Sofia Nascimento, que me deu forças e tornou os momentos finais desta caminhada muito mais leves e fáceis de suportar.

Também agradeço aos meus amigos mais próximos, Alexandre Salles e Paulo Randerson, que estiveram ao meu lado desde o início até o fim desta trajetória, sempre me apoiando e proporcionando momentos de alegria.

Sou grato ao meu orientador, Almir Kimura, que me deu todo o suporte necessário para a conclusão deste trabalho e que foi fundamental durante todo o curso, desde quando me deu a oportunidade de ingressar no Laboratório de Fabricação Digital do OCEAN, onde tive acesso a inúmeras oportunidades e desenvolvi muita experiência.

Agradeço ainda aos meus amigos da faculdade, que tornaram os dias muito mais agradáveis, especialmente: Kaue Martins, Airton Silva, Sandro Azevedo, Marcos Vinicius, Vinicius Santos, Yuri Macedo, Emanuel Thiago, Victor Silva e Marcus Eduardo. Agradeço também a todos os outros que, de forma direta ou indireta, contribuíram na convivência, no apoio emocional, nas atividades e nos estudos.

Por fim, gostaria de agradecer à Escola Superior de Tecnologia da Universidade do Estado do Amazonas (UEA), por proporcionar toda a estrutura e por reunir pessoas que foram muito importantes para o desenvolvimento desta jornada.

Epígrafe —

"Você está correndo o risco de viver uma vida tão confortável e fácil que vai morrer sem nunca ter alcançado seu pleno potencial." (David Goggins)

Resumo

O desenvolvimento de sistemas de monitoramento de vazamento de Gás Liquefeito de Petróleo (GLP) é fundamental para garantir a segurança de usuários e ambientes onde esse gás é manipulado. Este projeto propõe um dispositivo integrado, utilizando tecnologias de Internet das Coisas (IoT) para a detecção, notificação e monitoramento remoto de vazamentos de GLP. O sistema inclui a modelagem tridimensional da estrutura de proteção no *SolidWorks*, impressão 3D para a fabricação das peças, e o projeto de uma placa de circuito impresso confeccionada com máquina de fresagem CNC. O dispositivo conta com um sensor para identificar concentrações perigosas de GLP, um microcontrolador ESP32 para o processamento dos dados e um aplicativo móvel desenvolvido em *Swift*, que permite o monitoramento em tempo real e o envio de notificações de emergência. A validação foi realizada por meio de testes controlados, demonstrando a robustez, eficiência e potencial de aplicação do sistema em ambientes residenciais e comerciais.

Palavras-chave: monitoramento de gás, IoT, impressão 3D, ESP32, segurança.

Abstract

The development of monitoring systems for Liquefied Petroleum Gas (LPG) leakage is essential to ensure the safety of users and environments where this gas is handled. This project proposes an integrated device using Internet of Things (IoT) technologies for the detection, notification, and remote monitoring of LPG leaks. The system includes the three-dimensional modeling of the protective structure using *SolidWorks*, 3D printing for the fabrication of the parts, and the design of a printed circuit board manufactured with a CNC milling machine. The device features a sensor to identify hazardous LPG concentrations, an ESP32 microcontroller for data processing, and a mobile application developed in *Swift*, enabling real-time monitoring and the sending of emergency notifications. The validation was carried out through controlled tests, demonstrating the robustness, efficiency, and application potential of the system in residential and commercial environments.

Keywords: gas monitoring, IoT, 3D printing, ESP32, safety.

Lista de ilustrações

Figura 1 – Triângulo do Fogo. 16
Figura 2 – Proporção Ar-GLP Necessária para a Ocorrência de Fogo. 16
Figura 3 – Condições necessárias para a ocorrência de fogo. 17
Figura 4 – Arquitetura para IoT. 18
Figura 5 – Blocos básicos da IoT. 20
Figura 6 – Componentes de um Microcontrolador. 22
Figura 7 – Principais *softwares* de Modelagem 3D. 27
Figura 8 – Etapas da produção por manufatura aditiva. 28
Figura 9 – Sensor MQ-5 para detecção de Gás Liquefeito de Petróleo (GLP). . . . 29
Figura 10 – Características de sensibilidade do sensor MQ-5 para diferentes gases. . 30
Figura 11 – Placa ESP32 ESP-WROOM-32 DEVKit V1. 31
Figura 12 – *LEDs* para Alerta Visual. 33
Figura 13 – *buzzer* para Alerta Sonoro. 34
Figura 14 – Fonte ajustável. 35
Figura 15 – Medidas da tampa. 36
Figura 16 – Modelo 3D da tampa. 37
Figura 17 – Medidas da Carcaça. 38
Figura 18 – Modelo 3D da Carcaça. 39
Figura 19 – Montagem Final do Modelo 3D. 39
Figura 20 – Modelos 3Ds no *software* de fatiamento. 40
Figura 21 – Impressão em Processo. 40
Figura 22 – Peças Impressas em 3D. 41
Figura 23 – Montagem na Protoboard. 42
Figura 24 – Esquemático no Proteus. 42
Figura 25 – Prévia da PCI no Proteus. 43
Figura 26 – Placa Usinada na CNC de Precisão. 43
Figura 27 – Teste de Curto-circuito e Processo de Soldagem de Componentes. . . . 44
Figura 28 – Montagem Final da PCI. 44
Figura 29 – Código da Leitura. 45
Figura 30 – Código dos Estados da Leitura. 46
Figura 31 – Código para Tratamento de Dados - parte 1. 46
Figura 32 – Código para Tratamento de Dados - parte 2. 47
Figura 33 – Código para Conexão Wi-Fi e MQTT. 48
Figura 34 – UML - Parte 1. 49
Figura 35 – UML - Parte 2. 49
Figura 36 – Árvore de Códigos. 50

Figura 37 – Código para Inscrição no Tópico. 51
Figura 38 – Código Principal de Tratamento dos Dados Recebidos via MQTT. . . . 52
Figura 39 – Configurações do MQTT Explorer. 53
Figura 40 – Publicação e Inscrição de Tópicos no MQTT Explorer. 53
Figura 41 – Câmara de Gás. 55
Figura 42 – Válvula de Gás. 56
Figura 43 – Dispositivo Inserido na Câmara de Gás. 57
Figura 44 – Protótipo em Validação. 58
Figura 45 – Dispositivo Sensor de Gás Finalizado. 59
Figura 46 – Aplicativo Móvel 60
Figura 47 – Notificação de Alerta Amarelo 61
Figura 48 – Notificação de Alerta Vermelho 61
Figura 49 – Gráfico: Leitura x Tempo. 62
Figura 50 – Teste Final 63
Figura 51 – Sensor de Gás Comercial Disponível no Mercado. 64

Lista de abreviaturas e siglas

CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
FDM	Fused Deposition Modeling
GLP	Gás Liquefeito de Petróleo
IDE	Integrated Development Environment
IP	Internet Protocol
IOT	Internet of Things
MQTT	MQ Telemetry Transport
NFC	Near Field Communication
PCI	Placa de Circuito Impresso
RFID	Radio Frequency Identification
STL	Standard Triangle Language
UML	Unified Modeling Language

Sumário

1 INTRODUÇÃO . . . 13
1.1 Objetivos . . . 14
1.1.1 Objetivo Geral . . . 14
1.1.2 Objetivos Específicos . . . 14
1.2 Organização do Trabalho . . . 14

2 REFERENCIAL TEÓRICO . . . 15
2.1 Gás Liquefeito de Petróleo (GLP) . . . 15
2.1.1 Periculosidade do Gás LP . . . 16
2.2 Internet das Coisas (IoT) . . . 17
2.2.1 Arquitetura da IoT . . . 18
2.2.2 Blocos Básicos de Construção da IoT . . . 19
2.2.3 Benefícios da IoT . . . 20
2.2.4 IoT no Sistema Proposto . . . 20
2.3 Sensores . . . 21
2.3.1 Princípios Básicos de Funcionamento . . . 21
2.3.2 Características Fundamentais dos Sensores . . . 21
2.4 Microcontroladores . . . 21
2.5 Aplicativo Móvel . . . 23
2.5.1 Linguagem Swift para Aplicativos Móveis . . . 23
2.5.2 Ambiente de Desenvolvimento: Xcode, utilizando UIKit com abordagem ViewCode . . . 24
2.6 Fabricação Digital . . . 24
2.6.1 Processo de Fabricação de Placas de Circuito Impresso (PCI) . . . 25
2.6.2 Modelagem e Impressão 3D . . . 27

3 MATERIAIS E MÉTODOS . . . 29
3.1 Materiais Utilizados . . . 29
3.1.1 Sensor MQ-5 . . . 29
3.1.1.1 Integração com o Sistema Proposto . . . 30
3.1.2 Microcontrolador ESP32 . . . 30
3.1.2.1 Características Técnicas do ESP32 . . . 31
3.1.2.2 Vantagens do ESP32 em Relação a Outras Plataformas . . . 32
3.1.2.3 Integração com Sensores e IoT . . . 32
3.1.2.4 Ferramentas de Desenvolvimento . . . 32
3.1.2.5 Aplicação no Sistema Proposto . . . 32

3.1.3 *LEDs* . . . 33
3.1.4 *buzzer* . . . 34
3.1.5 Fonte Ajustável para *Protoboard* . . . 34
3.2 Metodologia . . . 35
3.2.1 Desenvolvimento Mecânico . . . 35
3.2.1.1 Modelagem 3D das Peças . . . 36
3.2.1.2 Impressão 3D das peças . . . 40
3.2.2 Desenvolvimento Elétrico . . . 41
3.2.2.1 Confecção da Placa de Circuito Impresso . . . 41
3.2.2.2 Programação do Microcontrolador ESP32 . . . 45
3.2.3 Desenvolvimento do Aplicativo Móvel . . . 48
3.2.3.1 Comunicação entre o Aplicativo e o *Broker* MQTT . . . 51
3.2.4 Desenvolvimento dos Testes . . . 53
3.2.4.1 Calibração do Sensor . . . 54
3.2.4.2 Testes com o Dispositivo Calibrado . . . 58

4 RESULTADOS E DISCUSSÃO . . . 59
4.1 Resultado do Dispositivo Sensor de Gás . . . 59
4.2 Resultado do Aplicativo Móvel . . . 60
4.3 Resultado dos Testes e Validação Final . . . 61
4.4 Lista de Materiais . . . 63

5 CONCLUSÃO . . . 65
5.1 Trabalhos Futuros . . . 65

REFERÊNCIAS . . . 67

APÊNDICES 70

APÊNDICE A – DESENHOS CAD 2D E 3D . . . 71

APÊNDICE B – PROJETOS DA PLACA DE CIRCUITO IMPRESSO 77

APÊNDICE C – PROGRAMAS DO APLICATIVO IOS . . . 81

APÊNDICE D – PROGRAMAS DO ESP32 . . . 103

1 Introdução

O gás liquefeito de petróleo (GLP) é um componente essencial no cotidiano residencial e industrial brasileiro, especialmente no preparo de alimentos. Entretanto, sua utilização apresenta riscos significativos à segurança quando ocorrem vazamentos, podendo resultar em incêndios, explosões e intoxicações graves.

De acordo com o Corpo de Bombeiros de São Paulo, nos primeiros oito meses de 2010, foram registrados 2.078 casos de vazamento de gás no estado, que resultaram em 417 pessoas feridas. Dentre esses casos, houve 79 ocorrências com gás encanado, 167 com GLP fora de edificações e, liderando os acidentes, 507 com GLP engarrafado. (PMESP, 2023)

Atualmente, os métodos convencionais para a detecção de vazamentos, como a identificação pelo odor característico adicionado ao gás ou a aplicação de espuma de sabão nas conexões, mostram-se limitados, pois dependem da intervenção humana e não oferecem monitoramento contínuo.

Em um estudo realizado em um hospital na China, foram admitidos 1.337 pacientes com queimaduras, dos quais 195 foram internados em decorrência de 169 acidentes relacionados ao GLP. Houve 11 ocorrências com múltiplas vítimas. As queimaduras relacionadas ao GLP ocorreram com maior frequência em pacientes com idade entre 21 e 60 anos (73,85%). A maioria dos acidentes aconteceu entre os meses de maio e agosto (56,41%), e o local mais comum foi o domicílio (83,08%, correspondendo a 162 pacientes). O vazamento de gás foi a principal causa das queimaduras (81,03%), seguido por operações inadequadas (7,69%) e negligência ao cozinhar (2,05%).(JIN et al., 2018)

Diante desse cenário, o sistema proposto neste trabalho oferece uma solução inovadora ao integrar tecnologias da Internet das Coisas (IoT) com um aplicativo móvel personalizado. Utilizando um sensor MQ-5 calibrado para a detecção de GLP, acoplado a um microcontrolador ESP32, o protótipo realiza o monitoramento contínuo da concentração de gás no ambiente. Os dados são transmitidos via protocolo MQTT para um aplicativo desenvolvido em *Swift*, que classifica o risco em três níveis (seguro, suspeita e alerta) e notifica o usuário instantaneamente em caso de perigo.

A estrutura física do dispositivo foi projetada em *software* de modelagem 3D e fabricada por meio de impressão 3D, garantindo baixo custo de produção e alto grau de personalização. Embora utilize sensores de baixo custo, o sistema é capaz de fornecer um monitoramento contínuo e confiável, atendendo às demandas de ambientes residenciais e pequenos estabelecimentos comerciais. Essa abordagem representa um equilíbrio entre funcionalidade, acessibilidade e simplicidade de implementação.

1.1 Objetivos

1.1.1 Objetivo Geral

Desenvolver um protótipo para detecção de vazamentos de GLP, com capacidade de monitoramento contínuo, comunicação sem fio e emissão de alertas em tempo real por meio de um aplicativo móvel, utilizando tecnologias acessíveis baseadas em Internet das Coisas (IoT).

1.1.2 Objetivos Específicos

1. Selecionar e calibrar os componentes eletrônicos (sensor MQ-5 e microcontrolador ESP32) para detecção adequada de GLP;
2. Projetar e fabricar uma placa de circuito impresso (PCI) otimizada para a aplicação proposta;
3. Desenvolver o *firmware* responsável pelo processamento de dados e comunicação via protocolo MQTT;
4. Implementar um aplicativo móvel, utilizando a linguagem *Swift*, com interface de usuário intuitiva;
5. Projetar e fabricar a estrutura física do dispositivo por meio de impressão 3D;
6. Validar o funcionamento do sistema em condições reais de operação.

1.2 Organização do Trabalho

A estrutura deste trabalho está organizada da seguinte forma: o Capítulo 2 apresenta o referencial teórico que fundamenta a pesquisa. O Capítulo 3 descreve em detalhes a metodologia adotada para o desenvolvimento do projeto. No Capítulo 4, são expostos os resultados obtidos. Por fim, o Capítulo 5 reúne as considerações finais e a conclusão do estudo.

2 Referencial Teórico

2.1 Gás Liquefeito de Petróleo (GLP)

O Gás Liquefeito de Petróleo (GLP), conhecido popularmente como "gás de cozinha", é uma mistura de hidrocarbonetos composta principalmente por propano e butano, derivada do refino de petróleo e gás natural. Este gás é amplamente utilizado como fonte de energia em ambientes residenciais, comerciais e industriais devido à sua alta eficiência energética, facilidade de armazenamento e transporte, além do seu custo relativamente acessível. No Brasil, o GLP desempenha um papel fundamental na matriz energética, sendo um recurso importante para setores que vão do residencial ao agronegócio e à indústria (Sindigás, 2024).

Contudo, o GLP apresenta riscos consideráveis à segurança devido à sua natureza altamente inflamável. Em caso de vazamento, o GLP, que possui uma densidade maior do que o ar, tende a se acumular em áreas baixas e confinadas, como porões, cozinhas e depósitos com ventilação limitada. Esse comportamento eleva o risco de explosão, pois o gás acumulado pode formar uma mistura inflamável que, ao entrar em contato com uma fonte de ignição, como uma faísca, chama aberta ou até mesmo certos aparelhos eletrônicos em funcionamento, pode causar acidentes graves. Esse potencial explosivo reforça a necessidade de sistemas de monitoramento contínuo e de respostas rápidas para prevenir incidentes em ambientes onde o GLP é utilizado (Sindigás, 2024; MOREIRA, 2015).

O uso seguro do GLP depende não apenas de um monitoramento eficaz, mas também de práticas rigorosas de segurança ao longo de todas as etapas do ciclo de vida do gás, desde o envasamento até o consumo final. Exemplos históricos, como o acidente ocorrido em San Juanico, México, em 1984, ilustram os riscos associados ao manuseio inadequado do gás. Esse incidente, causado por um vazamento seguido de explosão, resultou em inúmeras fatalidades e danos materiais, destacando a importância de medidas preventivas robustas e de tecnologias de monitoramento que possam detectar concentrações elevadas e emitir alertas para evitar consequências semelhantes (Sindigás, 2024; MOREIRA, 2015).

Além dos riscos à segurança, o GLP pode representar um perigo para a saúde em ambientes fechados. Embora o GLP não seja tóxico, ele pode deslocar o oxigênio presente no ar em locais confinados, levando à hipóxia e causando sintomas como tontura e desorientação em casos de inalação prolongada. Esse efeito reforça a necessidade de sistemas de monitoramento capazes de detectar níveis elevados de GLP e alertar os ocupantes sobre a presença do gás antes que ele atinja concentrações perigosas (Sindigás, 2024).

2.1.1 Periculosidade do Gás LP

Segundo (Escola do Gás, 2018), só é possível obter fogo a partir de três pilares: o comburente, o combustível e o calor, sendo eles, respectivamente, o oxigênio, o GLP e uma fonte de ignição, que pode ser, por exemplo, um isqueiro. Na Figura 1, é possível observar essa relação, conhecida como Triângulo do Fogo.

Figura 1 – Triângulo do Fogo.

Fonte: Escola do Gás (2018)

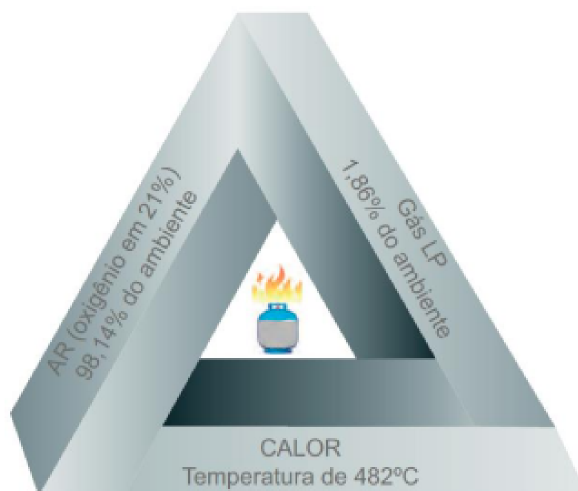
Entretanto, para que seja possível a combustão, é necessária mais uma condição: uma proporção específica entre o ar e o gás. Assim, ocorrem os chamados limites inferior e superior de inflamabilidade, conforme ilustrado na Figura 2. A combustão só ocorrerá quando a proporção ar-GLP estiver entre 1,86% e 9,50%.

Figura 2 – Proporção Ar-GLP Necessária para a Ocorrência de Fogo.

Fonte: Escola do Gás (2018).

Assim, caso o gás LP ocupe entre 1,86% e 9,50% do volume do ambiente, com a presença de uma fonte de calor a uma temperatura mínima de 482 °C, haverá fogo. A título de referência, uma chama de fósforo acesa emite calor a aproximadamente 1.649 °C. Na Figura 3, observa-se todas as condições necessárias para a ocorrência do fogo.

Figura 3 – Condições necessárias para a ocorrência de fogo.



Fonte: Escola do Gás (2018)

Com esses dados, é possível desenvolver um sistema para detecção de gás com maior precisão. Assim, para garantir a segurança em ambientes que utilizam GLP, um sistema de monitoramento eficiente, integrado a tecnologias de Internet das Coisas (IoT), torna-se indispensável. Esse tipo de sistema permite um monitoramento contínuo e em tempo real, aumentando a segurança dos ocupantes e a confiabilidade no uso do GLP. Essa abordagem será detalhada no capítulo seguinte, onde serão abordados os métodos e dispositivos específicos empregados para a detecção de vazamentos de GLP.

2.2 Internet das Coisas (IoT)

A Internet das Coisas (IoT) é definida como uma infraestrutura global que conecta objetos físicos e virtuais à internet, permitindo que troquem informações e realizem ações de maneira autônoma. Segundo (SANTOS et al., 2016), a IoT utiliza dispositivos inteligentes, como sensores, atuadores e microcontroladores, para coletar, processar e transmitir dados, criando um ecossistema interconectado. Essa tecnologia tem transformado setores como segurança, saúde, cidades inteligentes e automação residencial, tornando os ambientes mais eficientes e seguros.

A IoT se destaca também pela sua capacidade de análise preditiva e tomada de decisão com base nos dados coletados em tempo real. Segundo (SANTOS et al., 2016), os sistemas IoT podem integrar tecnologias como aprendizado de máquina e big data para interpretar os padrões dos dados capturados, permitindo não apenas a reação imediata a eventos críticos, mas também a previsão de possíveis falhas ou condições de risco. Essa característica é especialmente relevante para sistemas de monitoramento de segurança, como o proposto neste trabalho, pois aumenta a confiabilidade do sistema e possibilita a

antecipação de medidas preventivas, reduzindo significativamente os riscos de acidentes. Dessa forma, a IoT não apenas monitora e detecta vazamentos, mas também se torna uma ferramenta estratégica para aprimorar a segurança e a eficiência operacional.

No contexto do monitoramento de vazamento de GLP, a IoT é essencial para garantir que vazamentos sejam detectados em tempo real e que alertas sejam enviados de forma rápida e eficaz. Com a integração de sensores, como o MQ-5, e microcontroladores, como o ESP32, a IoT possibilita a coleta de dados do ambiente, a transmissão dessas informações por redes sem fio e a tomada de decisões automáticas, como o acionamento de alarmes locais ou o envio de notificações a um aplicativo móvel.

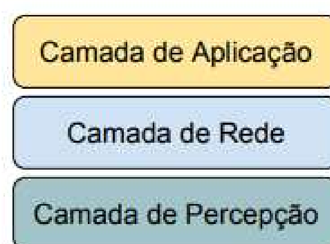
2.2.1 Arquitetura da IoT

De acordo com (SANTOS et al., 2016), a arquitetura da IoT pode ser dividida em três camadas principais:

- **Camada de Percepção:** Responsável pela interação direta com o ambiente por meio de sensores e atuadores. Por exemplo, no sistema proposto, o sensor MQ-5 monitora a concentração de gás, enquanto *LEDs* e *buzzers* atuam como indicadores visuais e sonoros de emergência.
- **Camada de Rede:** Encarregada de transmitir os dados coletados pelos sensores para sistemas centrais ou dispositivos conectados. Tecnologias como Wi-Fi são frequentemente utilizadas para garantir comunicação em tempo real, como no sistema do ESP32 utilizado neste trabalho.
- **Camada de Aplicação:** Relaciona-se diretamente com o usuário final, disponibilizando os dados coletados e analisados em interfaces como aplicativos móveis ou dashboards. No sistema proposto, o aplicativo mobile desempenha esse papel, notificando o usuário sobre o vazamento de GLP e oferecendo orientações de segurança.

Na Figura 4 são ilustradas as três camadas.

Figura 4 – Arquitetura para IoT.



Fonte: Santos et al. (2016)

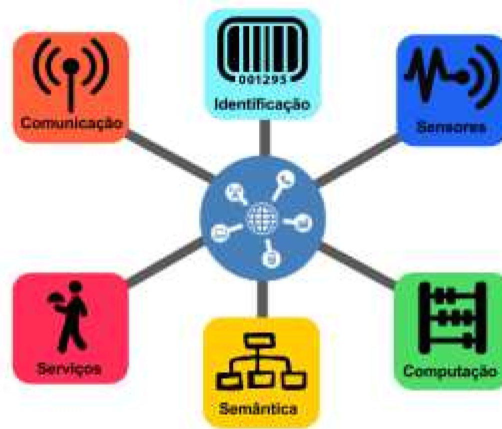
2.2.2 Blocos Básicos de Construção da IoT

Os sistemas IoT são construídos a partir de diversos blocos básicos que trabalham de forma integrada para viabilizar sua operação. De acordo com (SANTOS et al., 2016), esses blocos podem ser descritos como:

- **Identificação:** A identificação é fundamental para conectar objetos à internet de forma única e rastreável. Tecnologias como RFID *(Radio Frequency Identification)*, NFC*(Near Field Communication)* e endereçamento IP são amplamente utilizadas para atribuir identificadores únicos a dispositivos, garantindo seu correto funcionamento no sistema.
- **Sensores e Atuadores:** Os sensores capturam informações do ambiente, como temperatura, umidade ou concentração de gás, enquanto os atuadores reagem a essas condições, realizando ações como acionar alarmes ou ajustar variáveis no ambiente. No sistema proposto, o sensor MQ-5 detecta vazamentos de GLP e, em resposta, os atuadores (*LEDs* e *buzzer*) alertam os usuários.
- **Comunicação:** A comunicação é o elo que conecta os dispositivos IoT, permitindo que troquem informações entre si ou com servidores. Tecnologias como Wi-Fi, Bluetooth, ZigBee e protocolos como MQTT são frequentemente utilizadas para garantir a troca de dados em tempo real. No sistema proposto, a conectividade Wi-Fi desempenha esse papel, transmitindo os dados do ESP32 para o aplicativo móvel.
- **Computação:** A computação envolve o processamento dos dados capturados pelos sensores. Esse processamento pode ser local, utilizando *edge computing*, ou remoto, com a ajuda de *cloud computing*. No sistema de monitoramento de vazamento de GLP, o ESP32 processa os dados do sensor e toma decisões automáticas, como o acionamento de alarmes.
- **Serviços:** Os serviços disponibilizam informações processadas ao usuário final, seja por meio de aplicativos móveis ou dashboards interativos. O aplicativo desenvolvido neste trabalho cumpre essa função ao notificar os usuários sobre vazamentos e fornecer orientações de segurança em tempo real.
- **Semântica:** Este bloco trata da extração de conhecimento útil dos dados capturados, permitindo que o sistema atue de forma eficiente. Técnicas como *Resource Description Framework* (RDF) e *Web Ontology Language* (OWL) são frequentemente utilizadas para organizar e interpretar informações.

Na Figura 5, podemos visualizar os blocos básicos de construção da IoT.

Figura 5 – Blocos básicos da IoT.



Fonte: Santos et al. (2016)

2.2.3 Benefícios da IoT

A IoT oferece uma série de benefícios que tornam sua aplicação indispensável em sistemas de segurança. Entre os principais, destacam-se:

- **Automação e Eficiência:** Reduz a intervenção humana ao automatizar o processo de monitoramento e envio de alertas em caso de emergência.
- **Monitoramento Remoto:** Possibilita que o usuário acompanhe as condições do ambiente em tempo real, independentemente de sua localização.
- **Escalabilidade:** Permite a expansão do sistema com a adição de novos sensores e funcionalidades, sem a necessidade de alterações significativas na infraestrutura.
- **Conectividade:** Promove a integração contínua entre dispositivos, sensores e plataformas de visualização de dados.

2.2.4 IoT no Sistema Proposto

Neste trabalho, a IoT é utilizada como base para o desenvolvimento do sistema de monitoramento de GLP. A integração de sensores com o ESP32 permite a coleta de dados e a transmissão em tempo real para o aplicativo móvel, garantindo que os usuários sejam notificados rapidamente em caso de vazamento. Além disso, a IoT possibilita uma resposta automatizada e eficiente, como o acionamento de alarmes visuais e sonoros, tornando o sistema seguro e confiável.

2.3 Sensores

Os sensores são elementos indispensáveis em sistemas de automação e monitoramento, sendo responsáveis por medir grandezas físicas do ambiente e convertê-las em sinais elétricos processáveis. Como descrito por Wendling (WENDLING, 2010), esses dispositivos respondem a estímulos ambientais, como temperatura, pressão, luminosidade ou concentração de gases, fornecendo dados essenciais para a tomada de decisões em sistemas de controle. No contexto deste trabalho, sensores de gás desempenham um papel crucial na detecção de vazamentos de Gás Liquefeito de Petróleo (GLP), contribuindo para a segurança e eficiência no monitoramento.

2.3.1 Princípios Básicos de Funcionamento

Os sensores podem ser classificados de acordo com o tipo de saída gerada. Sensores analógicos fornecem um sinal contínuo, proporcional à grandeza medida, enquanto sensores digitais apresentam uma saída discreta, assumindo estados lógicos binários (0 ou 1). Essa saída pode ser diretamente interpretada pelo sistema de controle ou passar por circuitos de interface para ajustes necessários.

2.3.2 Características Fundamentais dos Sensores

A seleção de sensores para aplicações críticas, como a detecção de vazamentos de GLP, exige a análise de características fundamentais, incluindo:

- **Linearidade**: Refere-se à proporcionalidade entre a concentração do gás e a saída do sensor, o que simplifica a calibração e a interpretação dos dados.
- **Velocidade de Resposta**: A rapidez com que o sensor reage às variações de concentração é essencial para garantir ações corretivas em tempo hábil.
- **Faixa de Operação**: Representa a amplitude de concentrações que o sensor é capaz de medir com precisão, compatível com os níveis de risco do ambiente.

Essas características são indispensáveis para garantir a confiabilidade do sistema de monitoramento, especialmente em situações de emergência.

2.4 Microcontroladores

Microcontroladores são dispositivos eletrônicos programáveis compostos por um conjunto de componentes integrados em um único circuito, incluindo unidade central de processamento (CPU), memória (RAM, ROM, EEPROM) e periféricos de entrada e saída.

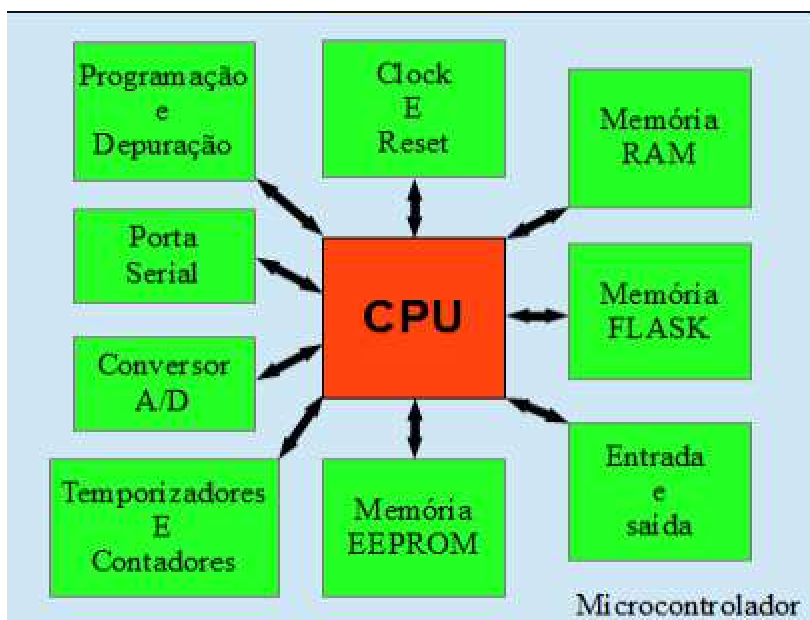
Esses dispositivos são amplamente utilizados em sistemas embarcados e aplicações que exigem controle automatizado e operação em tempo real.

Segundo (KERSCHBAUMER, 2013), os microcontroladores diferenciam-se dos microprocessadores por integrarem, em um único chip, todos os recursos necessários para o funcionamento de um sistema computacional embarcado, eliminando a necessidade de componentes externos adicionais, como memória e interfaces.

Os microcontroladores operam de forma eficiente em tarefas específicas, como o controle de sensores, atuadores, motores, displays, comunicação serial, entre outros. São comuns em dispositivos como eletrodomésticos, sistemas automotivos, equipamentos médicos e dispositivos industriais.

Ainda de acordo com o autor, a arquitetura interna de um microcontrolador inclui barramentos de dados, endereços e controle, timers, conversores analógico-digital (ADC), comunicação serial (UART, SPI, I²C) e, em muitos casos, recursos como interrupções e modos de economia de energia. A variedade de periféricos disponíveis torna os microcontroladores extremamente flexíveis e adaptáveis a diferentes tipos de aplicação.A Figura 6 ilustra os componentes de um microcontrolador.

Figura 6 – Componentes de um Microcontrolador.



Fonte: Kerschbaumer (2013)

Além disso, a programação de microcontroladores pode ser realizada em linguagens como C, *Assembly* ou linguagens interpretadas, dependendo da plataforma utilizada. Ferramentas como o Arduino IDE e o ESP-IDF (para microcontroladores ESP32) são exemplos comuns no ambiente educacional e profissional.

Dessa forma, os microcontroladores representam um elemento fundamental na

automação e no desenvolvimento de soluções tecnológicas embarcadas, sendo a base para a implementação do sistema proposto neste trabalho.

2.5 Aplicativo Móvel

O desenvolvimento de aplicativos móveis tornou-se essencial em diversos sistemas tecnológicos modernos, incluindo aplicações de monitoramento e automação baseadas em IoT. De acordo com a Microsoft Azure, aplicativos móveis consistem em *softwares* projetados especificamente para dispositivos portáteis, como smartphones e tablets, permitindo comunicação em tempo real, acesso remoto a dados e interação simplificada entre usuários e dispositivos (AZURE, 2024).

Esses aplicativos desempenham um papel crucial na conexão entre sistemas IoT e seus usuários finais. A comunicação em tempo real, viabilizada por protocolos eficientes como o MQTT, possibilita a rápida transferência de dados monitorados para os aplicativos, garantindo acessibilidade e uma experiência integrada. Além disso, a interface intuitiva e responsiva de aplicativos móveis é especialmente útil em situações que demandam decisões rápidas, como alertas de emergência e ações preventivas.

No contexto de sistemas de segurança, estudos destacam a relevância de aplicativos móveis na disseminação de informações críticas e na ampliação da interação com os dispositivos monitorados. Segundo Microsoft Azure, plataformas modernas permitem o desenvolvimento de aplicativos híbridos ou nativos, que otimizam tanto o desempenho quanto a compatibilidade com diferentes sistemas operacionais (AZURE, 2024). Essa flexibilidade tem sido fundamental para a popularização de aplicativos como ferramentas centrais em sistemas IoT.

Dessa forma, aplicativos móveis não apenas melhoram a usabilidade e acessibilidade de sistemas conectados, mas também expandem significativamente suas capacidades ao integrar monitoramento remoto e notificações automatizadas, alinhando-se aos objetivos de sistemas como o proposto neste trabalho.

2.5.1 Linguagem Swift para Aplicativos Móveis

Swift é uma linguagem de programação desenvolvida pela Apple Inc., projetada para o desenvolvimento de aplicativos nas plataformas iOS, iPadOS, macOS, watchOS e tvOS. Lançada em 2014, Swift substituiu progressivamente a linguagem Objective-C como principal ferramenta para o ecossistema Apple, destacando-se por sua sintaxe moderna, segurança e desempenho.

Segundo a Apple (Apple Inc., 2023a), *Swift* foi criada com foco em desempenho e facilidade de uso, permitindo que desenvolvedores escrevam código mais limpo, seguro

e eficiente. Além disso, sua integração nativa com o Xcode e as bibliotecas do sistema torna o desenvolvimento de aplicativos mais ágil, com suporte a recursos como interface gráfica baseada em SwiftUI, gerenciamento de memória automático e interoperabilidade com bibliotecas existentes em Objective-C.

No contexto deste trabalho, *Swift* foi utilizada no desenvolvimento de um aplicativo móvel para dispositivos iOS, responsável por exibir as leituras do sensor de GLP em tempo real e emitir alertas ao usuário. Sua escolha justifica-se pela necessidade de compatibilidade com o sistema operacional do dispositivo de destino, além da robustez e estabilidade proporcionadas pela linguagem.

Dessa forma, o uso de *Swift* contribui diretamente para a confiabilidade e usabilidade do sistema proposto, permitindo a construção de uma interface intuitiva, responsiva e integrada com os serviços de comunicação da arquitetura IoT implementada.

2.5.2 Ambiente de Desenvolvimento: Xcode, utilizando UIKit com abordagem ViewCode

O desenvolvimento do aplicativo móvel proposto foi realizado utilizando o Xcode, ambiente de desenvolvimento integrado (IDE) oficial da Apple para plataformas como iOS, macOS, watchOS e tvOS. O Xcode oferece um conjunto completo de ferramentas para codificação, design de interface, testes e distribuição de aplicativos. Segundo a Apple (Apple Inc., 2023b), o Xcode é otimizado para linguagens como *Swift* e Objective-C, além de oferecer simulação de dispositivos, gerenciamento de pacotes e integração com o sistema de publicação da App Store.

Para a construção da interface gráfica, optou-se pelo uso do *framework* UIKit, que é amplamente adotado no desenvolvimento de aplicativos iOS por meio da abordagem programática ou com Interface Builder.

Essa abordagem proporciona maior controle sobre o layout e facilita a reutilização de componentes, especialmente em projetos que exigem dinamismo e modularidade. Além disso, o uso do UIKit com ViewCode se alinha à prática de projetos mais escaláveis e testáveis, favorecendo a manutenção e organização do código.

Assim, o uso do Xcode em conjunto com UIKit e ViewCode contribuiu para a criação de uma interface eficiente, responsiva e alinhada com as boas práticas do desenvolvimento *mobile* moderno em plataformas Apple.

2.6 Fabricação Digital

A fabricação digital é um processo que permite a produção de objetos físicos a partir de modelos digitais, com dados enviados diretamente a máquinas controladas nume-

ricamente (CNC), eliminando etapas intermediárias de produção (BORGES; SILVA, 2016). Esse método revolucionou diversas áreas, como a indústria automobilística, aeronáutica e, mais recentemente, a construção civil e a produção de dispositivos eletrônicos. No contexto atual, a fabricação digital destaca-se pela precisão e eficiência, permitindo a criação de peças customizadas e a redução do desperdício de material.

No campo da prototipagem, a fabricação digital desempenha um papel essencial na transformação de projetos digitais em componentes físicos, sejam eles peças estruturais ou placas de circuito impresso (PCIs). Com o uso de *softwares* CAD (*Computer-Aided Design*) e CAM (*Computer-Aided Manufacturing*), é possível projetar e fabricar peças diretamente em equipamentos de CNC, imprimindo qualidade e precisão no resultado final (CELANI; ORCIUOLI, 2008). Esse processo é amplamente adotado por universidades e laboratórios de inovação, conhecidos como *Fab Labs*, que vêm explorando essas tecnologias para ensino e pesquisa (BORGES; SILVA, 2016).

No contexto do projeto em questão, a fabricação digital foi utilizada para duas finalidades principais: a prototipagem estrutural com impressão 3D e a produção da placa de circuito impresso (PCI) utilizando CNC. No primeiro caso, a modelagem tridimensional foi realizada com o *SolidWorks*, um *software* amplamente reconhecido por sua robustez e precisão na criação de peças para impressão 3D. A estrutura do dispositivo foi desenvolvida visando à eficiência e adequação às necessidades do sistema.

Para a produção de PCIs, o uso de uma fresadora CNC se destacou como uma alternativa eficiente e sustentável, eliminando a necessidade de processos químicos tradicionalmente empregados na fabricação de placas. Segundo (BORGES; SILVA, 2016), a fabricação digital aplicada à eletrônica promove maior controle sobre o design e fabricação, possibilitando ajustes em tempo real e garantindo maior compatibilidade entre o projeto e o produto final.

No âmbito deste trabalho, ela desempenhou um papel central para a criação do protótipo funcional, integrando as áreas mecânica, elétrica e de programação.

2.6.1 Processo de Fabricação de Placas de Circuito Impresso (PCI)

As Placas de Circuito Impresso (PCIs) constituem a base física de praticamente todos os sistemas eletrônicos modernos, sendo essenciais para a montagem e interconexão de componentes. Conforme (COOMBS, 2001), as PCIs evoluíram de simples conectores elétricos para plataformas altamente sofisticadas, que não apenas suportam componentes eletrônicos, mas também otimizam o desempenho elétrico e mecânico dos circuitos. Elas são compostas por uma base isolante revestida com cobre, onde trilhas condutoras são formadas para estabelecer conexões entre os componentes do circuito. A base das PCIs é composta por um material isolante, como fenolite ou fibra de vidro, revestido com uma fina

camada de cobre. Essa estrutura é utilizada para criar as trilhas condutoras que interligam os componentes do circuito.

O fenolite é amplamente utilizado devido ao seu baixo custo e facilidade de usinagem. Composto por papel impregnado com resina fenólica, o fenolite oferece propriedades adequadas para prototipagem e dispositivos de baixa complexidade. Em comparação, o FR-4, feito de fibra de vidro e resina epóxi, apresenta maior resistência térmica e mecânica, sendo indicado para aplicações mais exigentes.

A fabricação de PCIs pode ser realizada por diferentes métodos, cada um com vantagens e desvantagens. A Tabela 1 apresenta uma comparação dos principais métodos.

Tabela 1 – Comparação entre métodos de fabricação de PCIs.

Método	**Vantagens**	**Desvantagens**
Corrosão Química	Alta eficiência para produções em larga escala; Custo relativamente baixo.	Requer uso de produtos químicos; Impacto ambiental negativo.
Serigrafia	Boa precisão para trilhas simples; Adequado para grandes volumes.	Menos flexível para prototipagem; Menor precisão em trilhas densas.
Fresagem CNC	Não utiliza produtos químicos; Alta precisão para prototipagem e pequenas séries; Permite ajustes rápidos no design.	Menos eficiente para produções em larga escala; Tempo de corte elevado para circuitos complexos.

Fonte: Elaborado com base em (RAISA, 2024)

Entre os métodos, a fresagem CNC tem ganhado destaque devido à sua flexibilidade e sustentabilidade. O processo utiliza ferramentas de corte controladas por computador para remover o cobre seletivamente, formando as trilhas do circuito. Além disso, a CNC é capaz de realizar furos precisos para a montagem de componentes e interconexões entre as camadas, sendo especialmente útil para ajustes rápidos e prototipagem (RAISA, 2024).

Os métodos de fabricação de PCIs são auxiliados por *softwares* especializados, como Proteus, Eagle e KiCad, que permitem criar e simular o layout do circuito antes da fabricação. Esses *softwares* são ferramentas essenciais para assegurar que o design teórico esteja alinhado com a implementação prática, reduzindo erros e otimizando o tempo de desenvolvimento.

Compreender os materiais, métodos e ferramentas de fabricação de PCIs é fundamental para o desenvolvimento de dispositivos eletrônicos confiáveis e eficientes.

2.6.2 Modelagem e Impressão 3D

A modelagem e impressão 3D são processos fundamentais no desenvolvimento de protótipos e na fabricação de peças personalizadas, combinando flexibilidade de design e eficiência na produção. A modelagem 3D consiste na criação de representações digitais tridimensionais de objetos, utilizando *softwares* especializados, como Fusion 360, SolidWorks e Inventor (LAB, 2023).

Esses *softwares*, ilustrados na Figura 7, permitem o desenvolvimento de geometrias detalhadas, que podem ser ajustadas com precisão antes da fabricação, garantindo que os objetos atendam às necessidades específicas de cada projeto.

Figura 7 – Principais *softwares* de Modelagem 3D.

Fonte: Autor (2025)

Os modelos digitais gerados nesses programas são convertidos em formatos como STL (*Standard Triangle Language*) e preparados para impressão por meio de *softwares* de fatiamento. O fatiamento divide o modelo em camadas e gera um arquivo de instruções para a impressora 3D, detalhando parâmetros como a espessura das camadas, a densidade de preenchimento e a velocidade de deposição (VOLPATO, 2017).

A impressão 3D, ou manufatura aditiva, baseia-se na construção de objetos por adição sucessiva de material. Essa abordagem elimina a necessidade de moldes ou ferramentas específicas, reduzindo custos e prazos de fabricação em comparação com métodos tradicionais (RODRIGUES CLEBER E SILVA, 2017).

A tecnologia de modelagem por fusão e deposição (*Fused Deposition Modeling – FDM*) é a mais amplamente utilizada, empregando termoplásticos como PLA (Ácido Polilático), ABS (Acrilonitrila Butadieno Estireno) e PETG (Polietileno Tereftalato Glicol). Cada material oferece propriedades distintas: o PLA destaca-se pela facilidade de impressão

e custo acessível, enquanto o ABS é preferido em aplicações que exigem maior resistência mecânica e térmica (VOLPATO, 2017).

Além disso, a aplicação da impressão 3D em projetos de engenharia, como sistemas de monitoramento de GLP, possibilita a criação de estruturas sob medida para acomodar sensores, microcontroladores e demais componentes. Essas estruturas protegem os dispositivos contra impactos, poeira e interferências externas, além de contribuir para a organização dos circuitos.

A Figura 8 apresenta de forma linear os processos da manufatura aditiva.

Figura 8 – Etapas da produção por manufatura aditiva.

Sequência de passos para a manufatura aditiva

Fonte: Adaptado de (RODRIGUES CLEBER E SILVA, 2017)

3 Materiais e Métodos

3.1 Materiais Utilizados

Nesta seção, são descritos os materiais utilizados na construção e implementação do protótipo proposto.

3.1.1 Sensor MQ-5

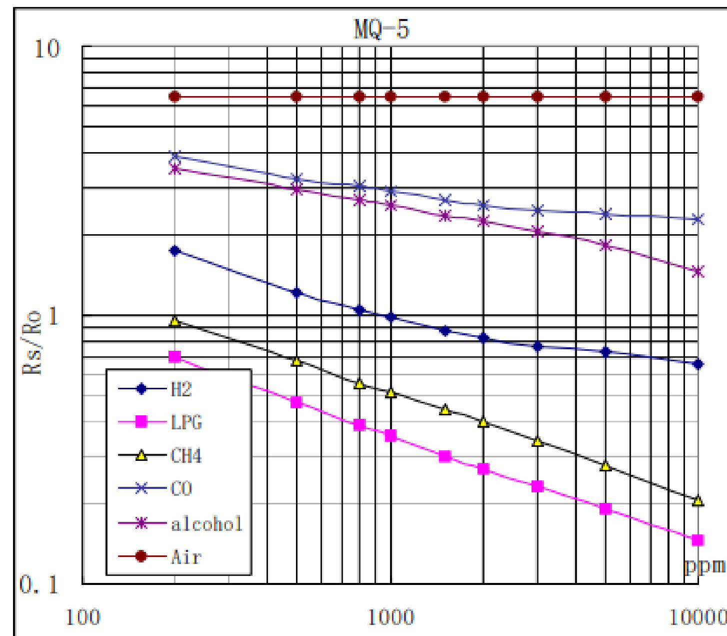
O sensor MQ-5 é amplamente utilizado na detecção de gases inflamáveis, como GLP, gás natural e gás de cidade. Sua camada sensível, composta por dióxido de estanho (SnO_2), altera sua resistência em resposta à concentração de gases no ambiente. Essa variação é traduzida em um sinal elétrico que pode ser interpretado por controladores, como o ESP32, integrando-se facilmente a sistemas de automação (HANWEI, 2024). O sensor pode ser visualizado na Figura 9.

Figura 9 – Sensor MQ-5 para detecção de Gás Liquefeito de Petróleo (GLP).

Fonte: Arducore (2025)

De acordo com o fabricante, o MQ-5 opera em uma faixa de detecção de 200 a 10.000 ppm, com resistência variável entre 10 kΩ e 60 kΩ . O sensor também possui um circuito de aquecimento interno que opera a 5V, garantindo as condições ideais para o funcionamento do material sensível. Sua construção robusta, com malha de aço inoxidável, proporciona resistência a interferências ambientais e proteção contra explosões (HANWEI, 2024).

Figura 10 – Características de sensibilidade do sensor MQ-5 para diferentes gases.



Fonte: Hanwei (2024)

A Figura 10 ilustra as características de sensibilidade do MQ-5 para diferentes gases. Observa-se que o sensor apresenta maior sensibilidade ao GLP e ao metano (CH_4), mostrando-se ideal para aplicações de detecção desses gases inflamáveis.

3.1.1.1 Integração com o Sistema Proposto

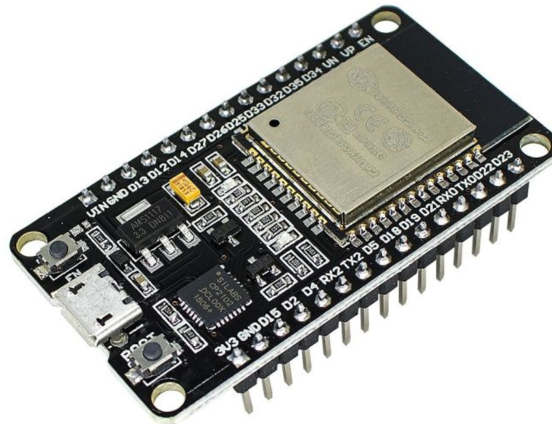
O MQ-5 é conectado ao microcontrolador ESP32, que converte o sinal analógico gerado pelo sensor em dados digitais processáveis. Essa integração permite a transmissão dos dados via comunicação sem fio para o aplicativo móvel, viabilizando notificações em tempo real sobre vazamentos de gás. A sensibilidade do sistema é calibrada conforme o datasheet do sensor, considerando variáveis ambientais, como temperatura e umidade, para evitar falsos positivos e garantir a confiabilidade na detecção.

3.1.2 Microcontrolador ESP32

O ESP32, desenvolvido pela Espressif Systems, é um microcontrolador altamente integrado e versátil, amplamente utilizado em aplicações de *Internet das Coisas* (IoT). Este dispositivo combina conectividade Wi-Fi e Bluetooth em uma única plataforma, tornando-o ideal para projetos que exigem comunicação sem fio de baixa potência e alto desempenho. Segundo (KOLBAN, 2017), o ESP32 destaca-se por suas capacidades de processamento, suporte a múltiplos protocolos de comunicação e uma ampla gama de interfaces periféricas.

Na Figura 11, podemos visualizar a placa de Desenvolvimento ESP-WROOM-32 Wi-Fi *Bluetooth* DEVKit V1.

Figura 11 – Placa ESP32 ESP-WROOM-32 DEVKit V1.



Fonte: Saravati Eletrônica (2025)

3.1.2.1 Características Técnicas do ESP32

O ESP32 possui um design robusto, com recursos que atendem às necessidades de aplicações complexas. Entre suas principais características estão:

- **Processador:** CPU dual-core Tensilica Xtensa LX6 com frequência de até 240 MHz.
- **Memória:** Memória RAM integrada de 520 KB, além de suporte a PSRAM externa.
- **Conectividade:** Compatibilidade com Wi-Fi 802.11 b/g/n e Bluetooth 4.2, incluindo BLE (*Bluetooth Low Energy*).
- **Interfaces Periféricas:** Suporte a GPIOs, SPI, I2C, UART, ADC, DAC, PWM, entre outros.
- **Segurança:** Mecanismos como criptografia AES, SHA-2, RSA e suporte a boot seguro.

Essas especificações tornam o ESP32 uma solução versátil para projetos que exigem alta conectividade e processamento eficiente. Em projetos de monitoramento de GLP, como o deste trabalho, a capacidade de integrar sensores e enviar dados para plataformas móveis em tempo real é um diferencial essencial.

3.1.2.2 Vantagens do ESP32 em Relação a Outras Plataformas

Em comparação com microcontroladores como o Arduino, o ESP32 oferece vantagens significativas, como:

- **Conectividade integrada:** O ESP32 possui Wi-Fi e Bluetooth nativos, eliminando a necessidade de módulos adicionais.
- **Desempenho superior:** Sua CPU dual-core e frequência mais alta permitem maior capacidade de processamento.
- **Baixo consumo de energia:** Modos de baixo consumo tornam o ESP32 ideal para aplicações alimentadas por bateria.
- **Custo-benefício:** Apesar de suas capacidades avançadas, o ESP32 é acessível, tornando-o viável para projetos educacionais e de pesquisa.

3.1.2.3 Integração com Sensores e IoT

O ESP32 é projetado para interagir de forma eficiente com sensores e módulos IoT. No contexto deste trabalho, o microcontrolador é integrado ao sensor MQ-5 para detectar vazamentos de GLP e enviar informações em tempo real a um aplicativo móvel. Sua compatibilidade com protocolos de comunicação, como MQTT e HTTP, facilita a troca de dados entre dispositivos.

3.1.2.4 Ferramentas de Desenvolvimento

O desenvolvimento para ESP32 é amplamente suportado por ferramentas como:

- **Arduino IDE:** Oferece uma interface simples e bibliotecas compatíveis para programação do ESP32.
- **Espressif IoT Development Framework (ESP-IDF):** Um ambiente de desenvolvimento oficial, mais avançado, com suporte a funcionalidades nativas do microcontrolador.
- **Linguagens de Programação:** Suporte a C, C++, MicroPython e Lua, permitindo flexibilidade na escolha da linguagem.

3.1.2.5 Aplicação no Sistema Proposto

No sistema de monitoramento de GLP proposto, o ESP32 atua como o núcleo do sistema, coletando dados do sensor MQ-5, processando essas informações e enviando notificações para o aplicativo móvel por meio de sua conectividade Wi-Fi. Além disso, o

microcontrolador é responsável por acionar alarmes locais, como *LEDs* e *buzzers*, em casos de emergência, reforçando a segurança do ambiente.

3.1.3 *LEDs*

Os *LEDs* (*Light Emitting Diodes*) são componentes eletrônicos que emitem luz visível quando percorridos por corrente elétrica. No sistema proposto, os *LEDs* desempenham a função de atuadores visuais, sinalizando ao usuário o estado atual da concentração de GLP detectada pelo sensor MQ-5.

Foram utilizados três *LEDs* de cores distintas (verde, amarelo e vermelho), como apresentado na Figura 12, sendo cada um deles responsável por representar um nível de risco:

- **LED Verde:** Indica que a concentração de gás está dentro dos parâmetros de segurança.
- **LED Amarelo:** Sinaliza uma situação de suspeita, onde a concentração de gás se aproxima do limite seguro.
- **LED Vermelho:** Alerta para um risco iminente, indicando a necessidade de ação imediata para evitar acidentes.

Figura 12 – *LEDs* para Alerta Visual.

Fonte: Mercado Livre (2025)

Os *LEDs* são conectados diretamente às portas digitais do ESP32, sendo acionados conforme a lógica implementada no *firmware*. A escolha por utilizar sinais visuais complementa as notificações enviadas pelo aplicativo móvel, proporcionando uma camada adicional de segurança, especialmente útil em situações em que o usuário esteja fisicamente próximo ao local monitorado.

3.1.4 *buzzer*

O *buzzer* piezoelétrico é um atuador sonoro empregado no sistema para fornecer alertas audíveis em situações de emergência. A Figura 13 ilustra o dispositivo. Sua principal função é complementar a sinalização visual realizada pelos *LEDs*, garantindo que a presença de vazamento de GLP seja percebida rapidamente, mesmo que o usuário não esteja visualizando o dispositivo.

Figura 13 – *buzzer* para Alerta Sonoro.

Fonte: Baú da Eletrônica (2025)

O acionamento do *buzzer* ocorre automaticamente sempre que o sistema detecta uma concentração de gás acima do limite seguro, configurado na lógica do *firmware*.

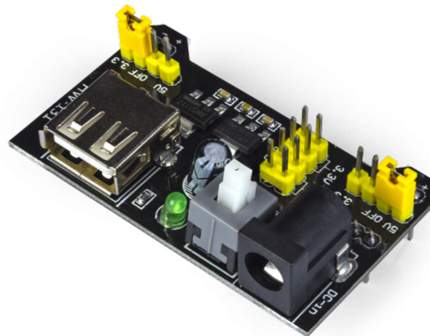
O *buzzer* é conectado a uma das portas digitais do ESP32, sendo acionado mediante comando lógico no *firmware*, em sincronia com as notificações enviadas ao aplicativo móvel e com a sinalização luminosa dos *LEDs*. Essa integração entre sinais sonoros e visuais reforça a eficiência do sistema de alerta, ampliando a segurança do ambiente monitorado.

3.1.5 Fonte Ajustável para *Protoboard*

Para a alimentação elétrica do sistema, foi utilizada uma fonte ajustável para *protoboard*, que proporciona praticidade e flexibilidade durante o desenvolvimento e os testes do protótipo. É possível visualizá-la na Figura 14. Esse tipo de fonte permite selecionar diferentes tensões de saída, normalmente 3,3 V ou 5 V, adequando-se às necessidades específicas dos componentes utilizados, como o sensor de detecção de gás liquefeito de petróleo MQ-5 e o microcontrolador ESP32.

A implementação dessa fonte no projeto exigiu a realização de algumas pequenas modificações físicas, devido ao seu propósito primário de ser utilizada diretamente em *protoboards*, possuindo, assim, pinos salientes na parte inferior da placa. Por esse motivo, foi realizada a remoção desses pinos, com o objetivo de economizar espaço e facilitar a modelagem 3D da carcaça.

Figura 14 – Fonte ajustável.



Fonte: RoboCore (2025)

A escolha pela fonte ajustável justifica-se pela facilidade de utilização, dispensando a necessidade de circuitos de alimentação externos mais complexos, além de permitir ajustes rápidos durante as etapas de montagem e validação do sistema. Essa solução também contribui para a organização e segurança dos testes, reduzindo o risco de danos aos componentes causados por falhas na alimentação elétrica.

3.2 Metodologia

O desenvolvimento do protótipo foi dividido em três etapas principais: o desenvolvimento mecânico, o desenvolvimento elétrico e, por fim, a implementação do aplicativo móvel.

3.2.1 Desenvolvimento Mecânico

Esta etapa envolveu o projeto e a fabricação da estrutura física do protótipo, utilizando técnicas de modelagem 3D e impressão em três dimensões (3D). O objetivo foi garantir um design funcional, compacto e de baixo custo, adequado para a acomodação dos componentes eletrônicos e para a instalação em ambientes residenciais ou comerciais.

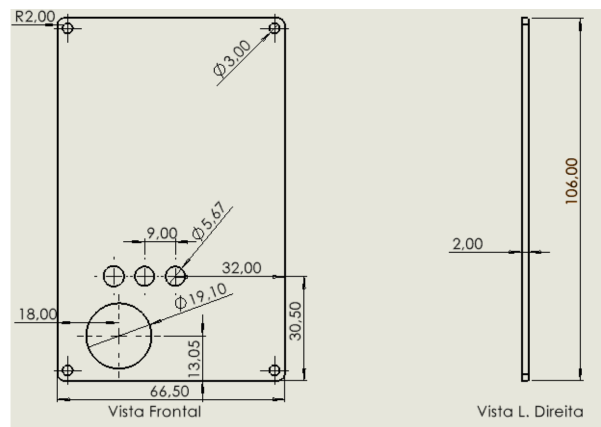
3.2.1.1 Modelagem 3D das Peças

Para este desenvolvimento, foi utilizado o *software* SolidWorks, responsável pela modelagem 3D da estrutura do protótipo. O principal desafio dessa etapa foi o processo criativo para conceber um design funcional, além da necessidade de obter medidas precisas, o que exigiu conhecimentos de metrologia e o uso de ferramentas adequadas, como o paquímetro, amplamente empregado neste projeto.

A seguir, são apresentadas as estruturas desenvolvidas para acomodar os componentes eletrônicos. Foram elaboradas duas peças: a tampa, cuja função é vedar toda a estrutura, além de possuir aberturas específicas para os *LEDs* e para o sensor de presença de gás; e a carcaça, destinada a abrigar internamente a placa eletrônica, a fonte de alimentação e o *buzzer*.

Na Figura 15, apresenta-se o desenho 2D da tampa, no qual é possível visualizar as medidas utilizadas para o desenvolvimento do modelo.

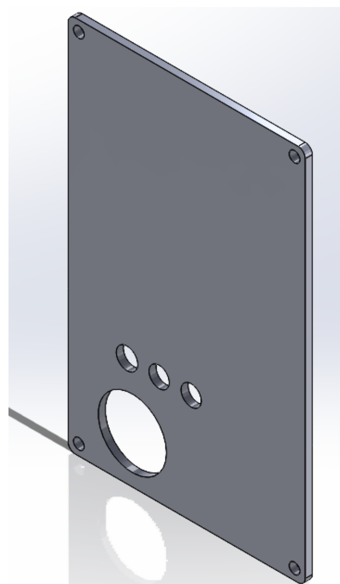
Figura 15 – Medidas da tampa.



Fonte: Autor (2025)

Na Figura 16, observa-se o modelo 3D da tampa, no qual se verificam as aberturas destinadas à fixação por meio de parafusos, além das aberturas para os *LEDs* e para o sensor, conforme mencionado anteriormente.

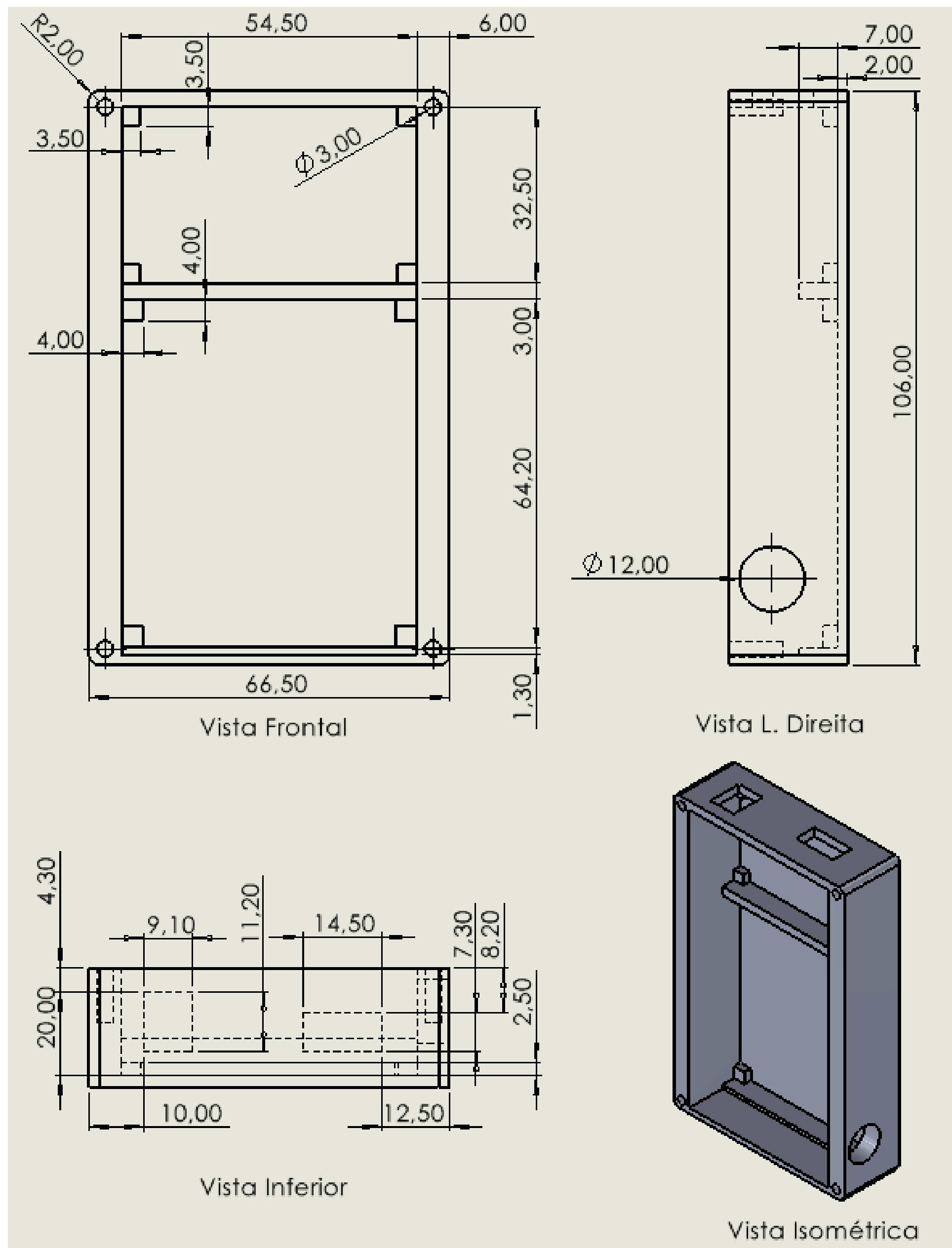
Figura 16 – Modelo 3D da tampa.



Fonte: Autor (2025)

Na Figura 17, são ilustrados os desenhos 2D da carcaça, nos quais é possível visualizar as medidas, assim como no desenho da tampa, facilitando a remodelagem ou manufatura, caso seja necessário.

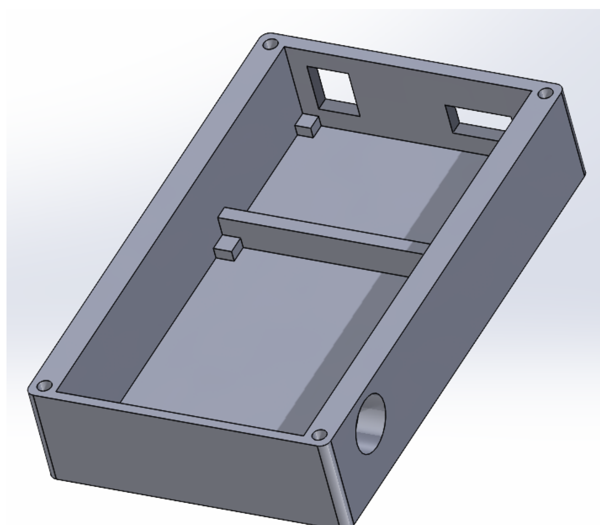
Figura 17 – Medidas da Carcaça.



Fonte: Autor (2025)

Na Figura 18, apresenta-se o modelo 3D da carcaça, no qual se observa o espaço necessário para o posicionamento dos componentes eletrônicos previamente estabelecidos. Foi necessária uma medição minimamente precisa, a fim de evitar folgas e garantir uma estrutura justa, aumentando a durabilidade e a resistência a movimentos durante o seu manejo. Com isso, a peça ficou pronta para ser fatiada no *software* de fatiamento e, posteriormente, impressa em 3D.

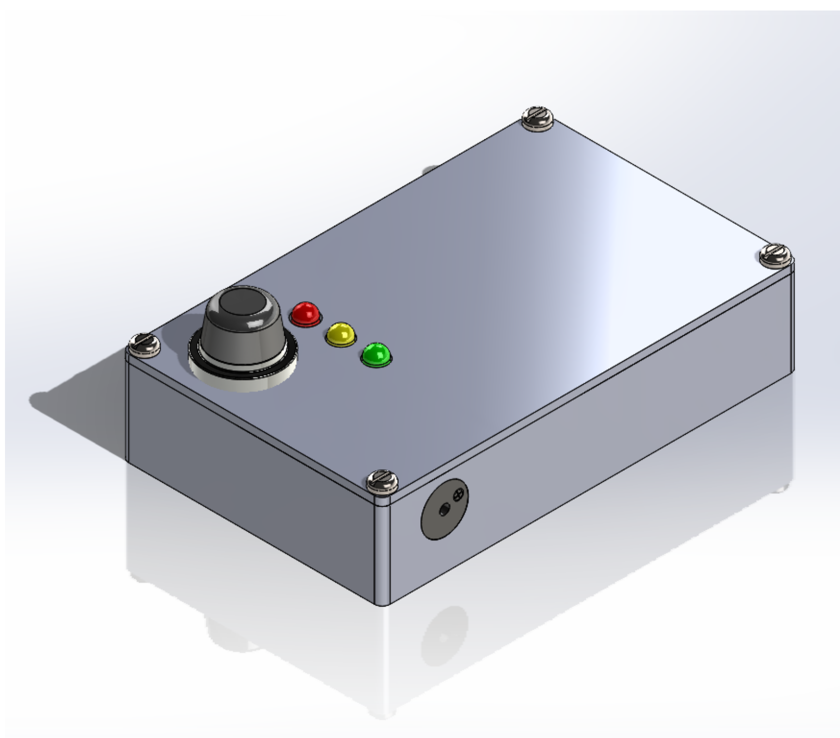
Figura 18 – Modelo 3D da Carcaça.



Fonte: Autor (2025)

Com o intuito de visualizar o protótipo final, na Figura 19 podemos observar uma montagem no *software* SolidWorks, na qual foi possível inserir e posicionar todas as peças e componentes do projeto, verificando, assim, a compatibilidade das medidas trabalhadas, além de ilustrar de forma fiel a representação final da parte física do sistema.

Figura 19 – Montagem Final do Modelo 3D.

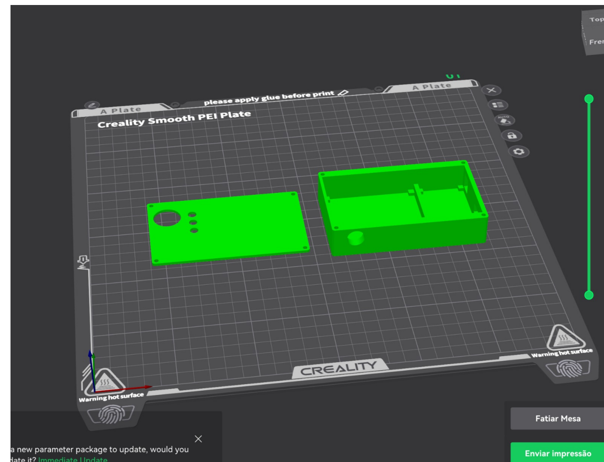


Fonte: Autor (2025)

3.2.1.2 Impressão 3D das peças

Partindo, para o *software* de fatiamento, etapa na qual é realizada toda a parte de CAM (Manufatura Assistida por Computador). O *software* utilizado foi o Creality Print, por ser o próprio da impressora utilizada. Na Figura 20, a seguir, é possível observar os modelos inseridos no ambiente de trabalho do fatiador.

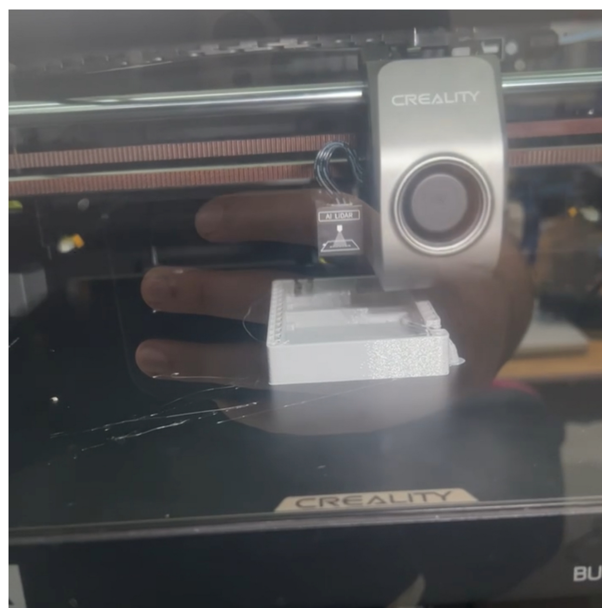
Figura 20 – Modelos 3Ds no *software* de fatiamento.



Fonte: Autor (2025)

Abaixo, pode-se observar a Figura 21, que demonstra a impressão em processo, a qual teve uma duração de aproximadamente 50 minutos. Vale ressaltar que o material utilizado foi o ABS, por oferecer maior resistência física em comparação ao PLA.

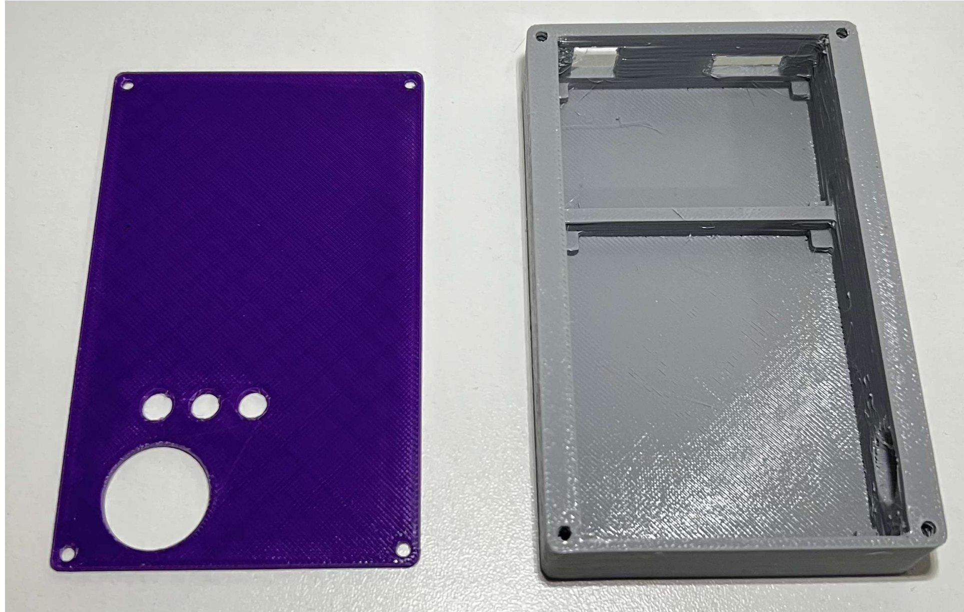
Figura 21 – Impressão em Processo.



Fonte: Autor (2025)

Com o término do processo de impressão, foram obtidas as seguintes peças, que, após serem retiradas da impressora, exigem um pós-processamento, utilizando lixas para melhorar o acabamento e ajustar as medidas de acordo com a necessidade real. A partir disso, é possível observar na Figura 22 o resultado final de todos esses passos.

Figura 22 – Peças Impressas em 3D.



Fonte: Autor (2025)

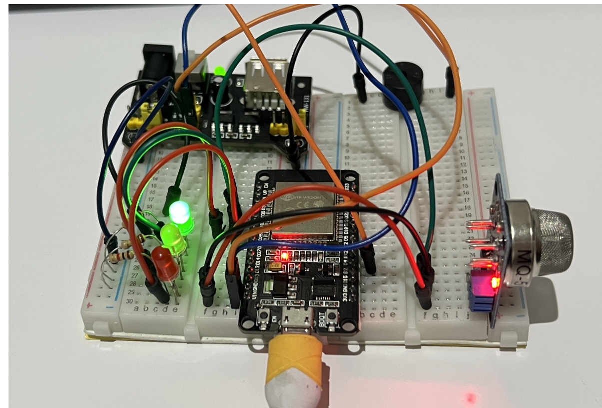
3.2.2 Desenvolvimento Elétrico

O desenvolvimento elétrico foi dividido em duas etapas: primeiramente, a confecção da Placa de Circuito Impresso (PCI), que envolveu todo o desenvolvimento do esquemático e, posteriormente, a sua manufatura. Na segunda etapa, foi implementada a programação do microcontrolador ESP32, responsável por receber dados do sensor, processá-los e enviá-los ao aplicativo, além de acionar os atuadores.

3.2.2.1 Confecção da Placa de Circuito Impresso

Nesta fase, conforme ilustrado na Figura 23, foram selecionados e integrados os componentes eletrônicos necessários para o funcionamento do sistema, como o sensor MQ-5, o microcontrolador ESP32, os *LEDs* e o *buzzer*. Inicialmente, foi realizada a montagem do circuito em protoboard, seguida da calibração do sensor e dos testes de funcionamento do sistema de detecção de GLP.

Figura 23 – Montagem na Protoboard.

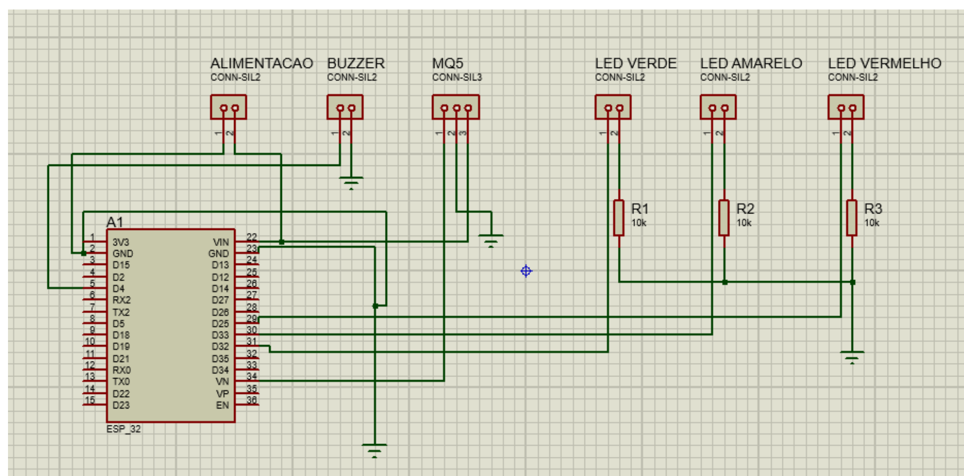


Fonte: Autor (2025)

A partir desses testes, foi projetada uma Placa de Circuito Impresso (PCI) utilizando o *software* Proteus.

A seguir, é possível observar, na Figura 24, o projeto da placa no *software* Proteus, no qual foram posicionados e conectados os principais componentes para a manufatura da PCI.

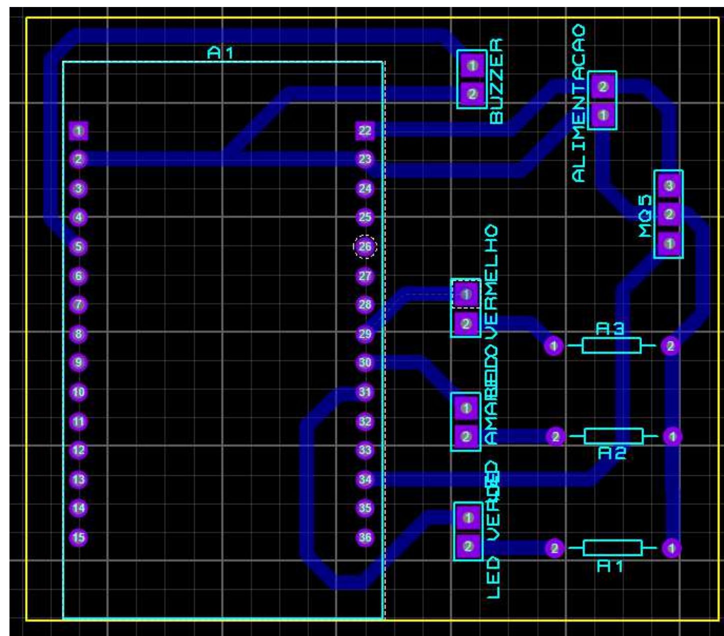
Figura 24 – Esquemático no Proteus.



Fonte: Autor (2025)

Após a conexão entre os dispositivos, foi possível definir o tamanho da placa, posicionar os componentes e implementar o roteamento das trilhas, ou seja, organizar as trilhas da melhor forma possível para ocupar o menor espaço viável, mantendo, contudo, o pleno funcionamento do circuito. Assim, é possível visualizar uma prévia do *layout* da placa na Figura 25.

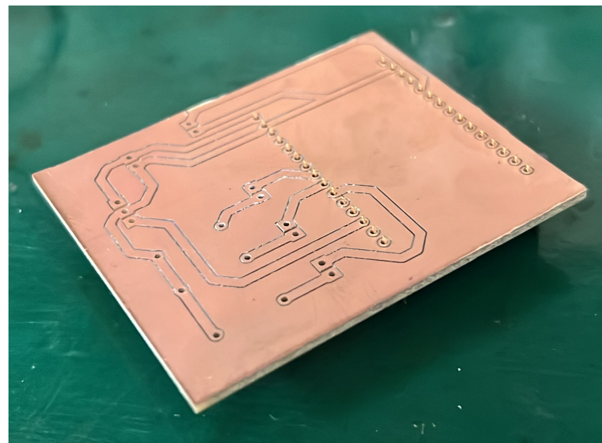
Figura 25 – Prévia da PCI no Proteus.



Fonte: Autor (2025)

Em seguida, na Figura 26 as trilhas foram confeccionadas em uma placa de fenolite com o auxílio de uma CNC de precisão.

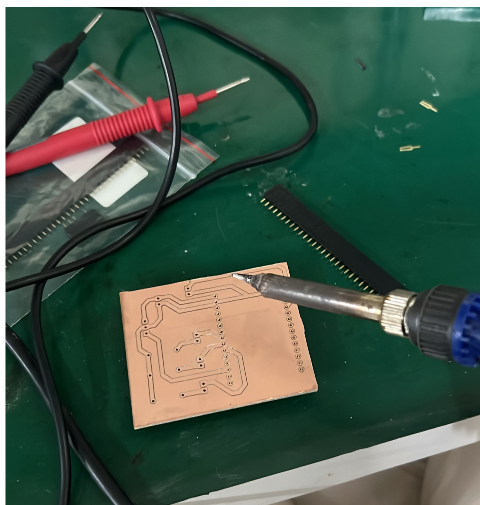
Figura 26 – Placa Usinada na CNC de Precisão.



Fonte: Autor (2025)

Finalizado o processo de usinagem, foi realizada a verificação da presença de curtos-circuitos entre as trilhas, utilizando um multímetro na função de continuidade. Caso fosse identificado algum curto, era realizado um pós-processamento para remover a camada residual que o estivesse provocando. Após essa etapa, iniciou-se o processo de soldagem dos componentes. A Figura 27 ilustra esse processo.

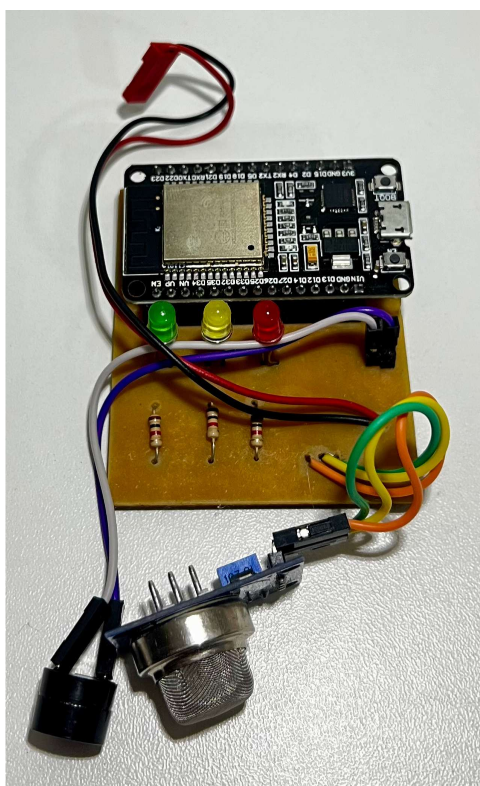
Figura 27 – Teste de Curto-circuito e Processo de Soldagem de Componentes.



Fonte: Autor (2025)

Por fim, os componentes eletrônicos foram soldados, resultando na montagem final da placa, conforme ilustrado na Figura 28.

Figura 28 – Montagem Final da PCI.



Fonte: Autor (2025)

3.2.2.2 Programação do Microcontrolador ESP32

Com a PCI finalizada, partiu-se para o desenvolvimento do código do microcontrolador, que consiste em um código de complexidade média, no qual são realizados o tratamento dos dados recebidos pelo sensor de gás MQ-5 e a comunicação com o *broker* MQTT.

O código foi desenvolvido no ambiente Arduino IDE, escolhido por ser um *software* amplamente utilizado, com excelente compatibilidade com o microcontrolador e bons resultados durante a sua utilização.

Assim, o código possui uma estrutura específica, na qual é necessário realizar a leitura do sensor, o tratamento dos dados recebidos, a conexão à rede Wi-Fi e a comunicação com o *broker* MQTT.

Para que o sistema funcionasse de forma adequada, foi necessário definir como realizar as leituras de maneira controlada. Dessa forma, foi montada uma câmara de gás, onde o sensor foi inserido internamente, permitindo a verificação das leituras.

Abaixo, na Figura 29, pode-se observar um trecho do código responsável pela leitura, utilizando a função `analogRead()` para extrair os dados do pino conectado ao sensor, além da conversão por meio de uma equação, transformando os valores em percentual, visando uma melhor visualização.

Figura 29 – Código da Leitura.

```
int lerPercentualSensor(int pinoSensor) {
  int leitura = analogRead(pinoSensor);
  int percentual = (int)((leitura / 4095.0) * 100);
  return percentual;
}
```

Fonte: Autor (2025)

Com o objetivo de gerar alertas e tratar os dados, na Figura 30 é possível visualizar o desenvolvimento que possibilitou a obtenção de três níveis de alerta, o acionamento de *LEDs* e do *buzzer*, a aplicação de um filtro às leituras e o envio dos dados para o *broker*, publicando no tópico escolhido.

Assim, primeiramente, a leitura do sensor é recebida em percentual e, a partir disso, na Figura 30, os valores são separados em estados, correspondentes aos níveis de gás monitorado.

Figura 30 – Código dos Estados da Leitura.

```
int percentualSensor = lerPercentualSensor(pinsensor);

Serial.print("Percentual do sensor: ");
Serial.println(percentualSensor);

int novoEstado;
if (percentualSensor <= 13) {
  novoEstado = 0;  // verde
} else if (percentualSensor <= 30) {
  novoEstado = 1;  // amarelo
} else {
  novoEstado = 2;  // vermelho
}
```

Fonte: Autor (2025)

Com o intuito de melhorar a confiabilidade da leitura obtida, foram implementadas algumas condições utilizando a função `millis()`, que gera um *clock*. Assim, quando é definido um valor para a variável `novoEstado`, este é comparado com a variável `estadoAtualSensor`, que inicialmente possui o valor `-1`. Caso o valor seja diferente, a variável `tempoUltimaMudanca` recebe o valor retornado pela função `millis()` e a variável `estadoAtualSensor` recebe o valor de `novoEstado`.

Abaixo na Figura 31, podemos verificar essas condições.

Figura 31 – Código para Tratamento de Dados - parte 1.

```
if (novoEstado != estadoAtualSensor) {
  tempoUltimaMudanca = millis();
  estadoAtualSensor = novoEstado;
}
```

Fonte: Autor (2025)

Em continuidade ao processo, é realizada mais uma verificação condicional, na qual se compara a diferença entre o valor atual retornado por `millis()` e o tempo registrado na variável `tempoUltimaMudanca`. Caso essa diferença seja maior que o valor de atraso definido (2 segundos), procede-se com uma nova verificação: se a variável `estadoAtualSensor` é diferente da variável `ultimoEstadoSensor`. Se for, `ultimoEstadoSensor` recebe o valor de `estadoAtualSensor`.

Em seguida, na Figura 32, executa-se uma estrutura `switch`, que, de acordo com o valor de `ultimoEstadoSensor`, define o acionamento dos *LEDs*, publica no tópico do *broker* e aciona o *buzzer*.

Figura 32 – Código para Tratamento de Dados - parte 2.

```
if ((millis() - tempoUltimaMudanca) > atrasoDebounce) {

  if (estadoAtualSensor != ultimoEstadoSensor) {

    ultimoEstadoSensor = estadoAtualSensor;

    switch (ultimoEstadoSensor) {
      case 0:
        digitalWrite(pinLEDGreen, HIGH);
        digitalWrite(pinLEDYellow, LOW);
        digitalWrite(pinLEDRed, LOW);
        digitalWrite(pinBuzzer, LOW);

        if (!messageSentG) {
          MQTT.publish(TOPIC_PUBLISH, "green");
          messageSentG = true;
          messageSentY = false;
          messageSentR = false;
        }
        break;

      case 1:
        digitalWrite(pinLEDGreen, LOW);
        digitalWrite(pinLEDYellow, HIGH);
        digitalWrite(pinLEDRed, LOW);
        digitalWrite(pinBuzzer, LOW);

        if (!messageSentY) {
          MQTT.publish(TOPIC_PUBLISH, "yellow");
          messageSentY = true;
          messageSentG = false;
          messageSentR = false;
        }
        break;

      case 2:
        digitalWrite(pinLEDGreen, LOW);
        digitalWrite(pinLEDYellow, LOW);
        digitalWrite(pinLEDRed, HIGH);
        digitalWrite(pinBuzzer, LOW);

        if (!messageSentR) {
          MQTT.publish(TOPIC_PUBLISH, "red");
          messageSentR = true;
          messageSentG = false;
          messageSentY = false;
        }
        break;
    }
  }
}
```

Fonte: Autor (2025)

Por fim, para possibilitar a comunicação com o *broker* MQTT e o envio dos dados ao aplicativo, foi desenvolvido um código, ilustrado na Figura 33, com o objetivo de manter o microcontrolador conectado à rede Wi-Fi e ao *broker* MQTT escolhido.

Figura 33 – Código para Conexão Wi-Fi e MQTT.

```
void conectaWiFi() {
  if (WiFi.status() == WL_CONNECTED) {
    return;
  }

  Serial.print("Conectando na rede: ");
  Serial.println(SSID);

  WiFi.begin(SSID, PASSWORD);

  while (WiFi.status() != WL_CONNECTED) {
    delay(100);
    Serial.print(".");
  }

  Serial.println();
  Serial.print("Conectado com IP: ");
  Serial.println(WiFi.localIP());
}

void conectaMQTT() {
  while (!MQTT.connected()) {
    Serial.print("Conectando no broker MQTT: ");
    Serial.println(BROKER_MQTT);

    if (MQTT.connect(ID_MQTT)) {
      Serial.println("Conectado ao broker MQTT!");
    } else {
      Serial.println("Falha na conexão. Tentando novamente em 10 segundos.");
      delay(10000);
    }
  }
}

void mantemConexoes() {
  if (!MQTT.connected()) {
    conectaMQTT();
  }
  conectaWiFi();
}
```

Fonte: Autor (2025)

Para um melhor entendimento, o código-fonte completo pode ser acessado no repositório do GitHub (MACIEL, 2025a).

3.2.3 Desenvolvimento do Aplicativo Móvel

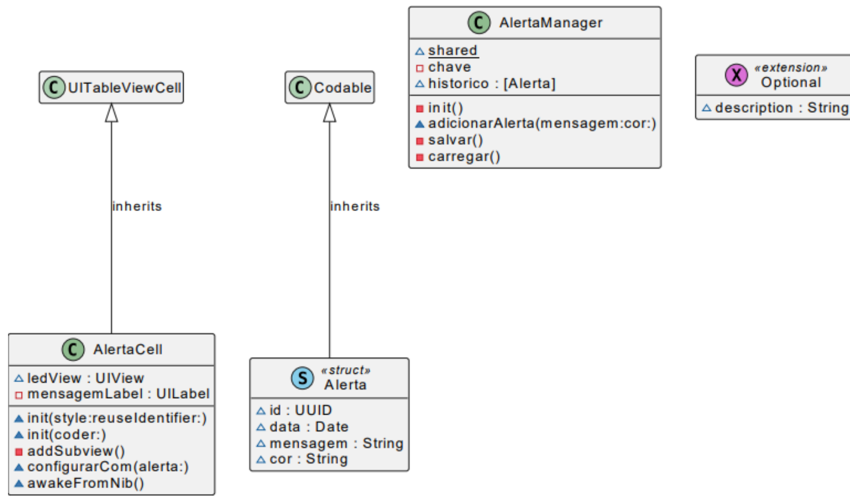
Por fim, foi desenvolvido um aplicativo móvel utilizando a linguagem *Swift*, com o objetivo de receber, processar e exibir as informações enviadas pelo sistema via protocolo MQTT. O aplicativo também foi responsável por alertar o usuário sobre possíveis vazamentos, por meio de notificações em tempo real e de uma interface intuitiva e responsiva.

O desenvolvimento foi realizado no ambiente Xcode, uma plataforma robusta e eficiente para a criação de *softwares* no ecossistema da Apple. Neste projeto, a interface foi construída utilizando a técnica conhecida como *ViewCode*, que consiste na criação manual dos elementos visuais da interface diretamente por meio de código *Swift*, sem a utilização de arquivos *Storyboard* ou *XIB*.

Para uma melhor visualização e especificação, as Figuras: Figura 34 e Figura 35 apresentam a modelagem de dados utilizando a abordagem UML, na qual é possível

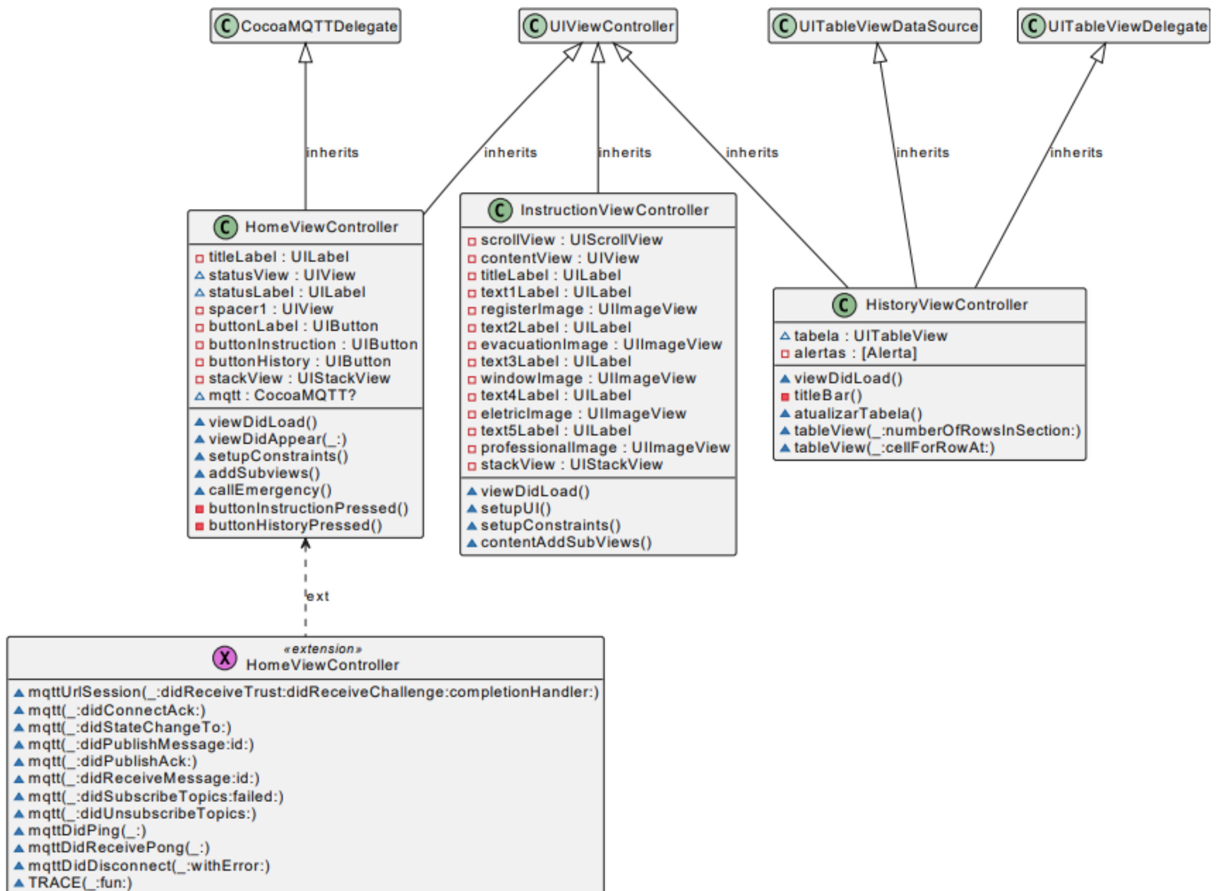
observar todas as classes, propriedades, métodos e suas respectivas relações.

Figura 34 – UML - Parte 1.



Fonte: Autor (2025)

Figura 35 – UML - Parte 2.

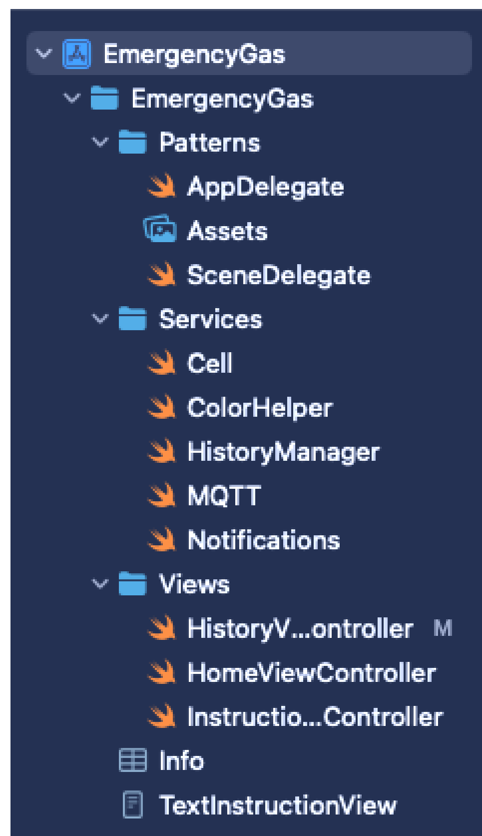


Fonte: Autor (2025)

Os códigos do aplicativo foram organizados em três pastas: uma pasta padrão, contendo todos os arquivos nativos do sistema; uma pasta destinada aos códigos de tratamento de dados; e, por fim, uma pasta com as *views*, onde são construídas todas as telas.

A seguir, apresenta-se a Figura 36, que ilustra a árvore de códigos desenvolvidos para o funcionamento do aplicativo.

Figura 36 – Árvore de Códigos.



Fonte: Autor (2025)

Para um melhor entendimento, descrevem-se, a seguir, as funções de cada código:

- **AppDelegate:** Arquivo responsável pelas configurações iniciais do ciclo de vida do aplicativo.
- **Assets:** Pasta que armazena os recursos gráficos e de mídia utilizados na interface.
- **SceneDelegate:** Responsável pela configuração das cenas e pela gestão das janelas do aplicativo.
- **Cell:** Arquivo destinado à criação de células reutilizáveis para a lista criada, com o objetivo de armazenar o histórico.

- **ColorHelper:** Classe auxiliar para converter os dados recebidos via MQTT, de `String` para `UIColor`, facilitando a troca de cores no status da tela principal. Também é responsável pela alteração das mensagens de status e pela definição das notificações enviadas ao usuário.
- **HistoryManager:** Gerenciador responsável pelo armazenamento e manipulação do histórico de dados coletados.
- **MQTT:** Módulo dedicado à implementação do protocolo MQTT, para envio e recebimento de mensagens.
- **Notifications:** Arquivo responsável pela configuração e envio de notificações ao usuário.
- **HistoryViewController:** Código responsável pela construção da tela de histórico, na qual são exibidos os dados de alertas armazenados localmente.
- **HomeViewController:** Arquivo responsável pela construção da tela principal, onde ocorre a primeira interação com o usuário, mostrando o status do sensor e os botões para ligar para a emergência, acessar instruções e visualizar o histórico de alertas, respectivamente.
- **InstructionViewController:** Responsável pela construção da tela de instruções em caso de emergência, orientando o usuário sobre as ações iniciais que devem ser tomadas.

O código-fonte completo do aplicativo está disponível no repositório do GitHub (MACIEL, 2025b).

3.2.3.1 Comunicação entre o Aplicativo e o *Broker* MQTT

Para que a comunicação seja realizada, é necessário que o aplicativo se inscreva no tópico especificado para receber os dados. Na Figura 37, observa-se um pequeno trecho do código MQTT mencionado anteriormente, no qual é realizada a inscrição no tópico.

Figura 37 – Código para Inscrição no Tópico.

```
func mqtt(_ mqtt: CocoaMQTT, didConnectAck ack: CocoaMQTTConnAck) {
    TRACE("ack: \(ack)")

    if ack == .accept {
        mqtt.subscribe("emergencygas")
    }
}
```

Fonte: Autor (2025)

Após a conexão, o aplicativo passa a receber os dados em formato de `String`, podendo, então, manipulá-los conforme necessário. A seguir, na Figura 38, apresenta-se um trecho do código MQTT, no qual está implementada a função responsável por todo o processamento relacionado aos dados recebidos via MQTT.

Figura 38 – Código Principal de Tratamento dos Dados Recebidos via MQTT.

```
func mqtt(_ mqtt: CocoaMQTT, didReceiveMessage message: CocoaMQTTMessage, id: UInt16 ) {
    TRACE("message: \(message.string.description), id: \(id)")
    let mensagemRecebida = message.string ?? "default"
    let novaCor = trocadeCor(mensagemRecebida)
    let novoStatus = trocadeStatus(mensagemRecebida)

    Notification(mensagemRecebida)
    DispatchQueue.main.async {
            self.statusView.backgroundColor = novaCor
            self.statusLabel.text = novoStatus


        }
    if mensagemRecebida != "green" {
        AlertaManager.shared.adicionarAlerta(mensagem: novoStatus, cor: mensagemRecebida)
        NotificationCenter.default.post(name: NSNotification.Name("NovoAlerta"), object: nil)
    }
```

Fonte: Autor (2025)

Inicialmente, é criada uma constante chamada `mensagemRecebida`, que armazena a `String` recebida. Em seguida, é criada outra constante, denominada `novaCor`, que armazena o dado tratado por meio da função `trocaDeCor()`. Além disso, há também a constante `novoStatus`, que recebe o dado processado a partir da função `trocaDeStatus()`.

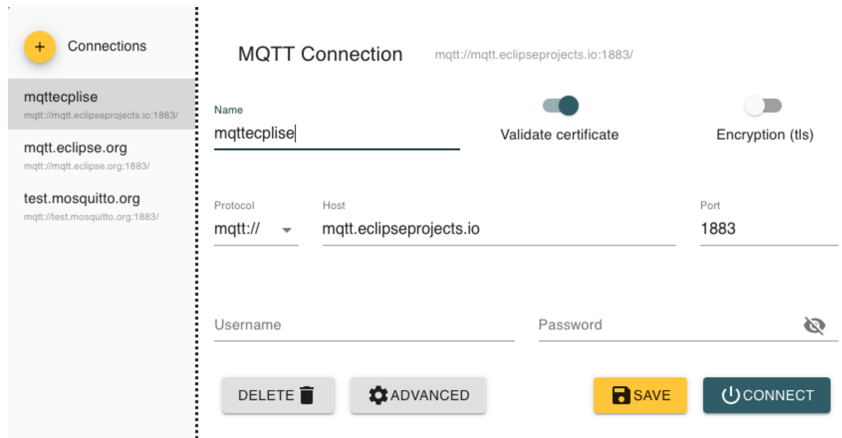
Com isso, por meio da função `notification()`, ocorre o tratamento da `String` recebida e o acionamento das notificações correspondentes.

Na sequência, com a utilização do `DispatchQueue.main.async`, os dados processados pelas funções anteriores — `novaCor` e `novoStatus` — são passados como parâmetros para realizar a alteração da cor e do texto na tela principal do aplicativo. Por fim, há uma condição que, quando satisfeita, gera um registro no log, resultando na adição de um item à tela de histórico.

Com o objetivo de realizar testes sem a necessidade de receber dados diretamente do ESP32, foi utilizado um *software* chamado MQTT Explorer, que funciona como cliente, sendo capaz de se inscrever e publicar em tópicos. Essa ferramenta facilitou significativamente a manipulação e a validação dos dados durante o desenvolvimento da comunicação.

A seguir, na Figura 39 são apresentadas as configurações do cliente MQTT para a conexão com o *broker* escolhido. Por questão de facilidade, foi selecionado o `mqtt.eclipseprojects.io`.

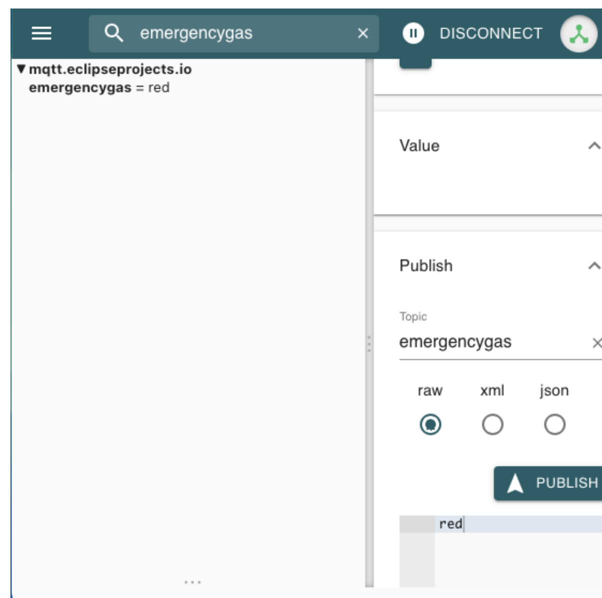
Figura 39 – Configurações do MQTT Explorer.



Fonte: Autor (2025)

Assim, é possível conectar-se ao *broker*, bem como publicar e se inscrever nos tópicos. Na Figura 40, visualiza-se esse processo.

Figura 40 – Publicação e Inscrição de Tópicos no MQTT Explorer.



Fonte: Autor (2025)

3.2.4 Desenvolvimento dos Testes

Nesta seção, são apresentados os procedimentos realizados para a validação do projeto, desde a calibração do sensor até os testes com o dispositivo já calibrado e pronto para uso.

3.2.4.1 Calibração do Sensor

Com o protótipo construído, foi necessário calibrar o sensor. Para um melhor aproveitamento, optou-se por realizar testes reais, com calibração empírica. Contudo, para garantir maior precisão nos resultados, observou-se a necessidade de um ambiente controlado.

Dessa forma, foi construída uma câmara de gás, conforme ilustrado na Figura 41, com volume calculado pelas expressões a seguir:

- Altura: 40 cm
- Largura: 26 cm
- Comprimento: 26 cm

O volume da câmara, em cm^3, é dado por:

$$V_{\mathrm{cm}} = \text{altura} \times \text{largura} \times \text{comprimento} \tag{3.1}$$

Substituindo os valores:

$$V_{\mathrm{cm}} = 40 \times 26 \times 26 = 27040\,\mathrm{cm}^3 \tag{3.2}$$

Convertendo para litros:

$$V_{\mathrm{L}} = \frac{27040}{1000} = 27,04\,\mathrm{L} \tag{3.3}$$

Figura 41 – Câmara de Gás.

Fonte: Autor (2025)

Como explicado no capítulo anterior, para que ocorra combustão do gás, é necessário que ao menos 1,8% do volume do ambiente seja preenchido por GLP. Podemos, então, calcular o volume necessário para que essa condição seja atendida:

$$V_{\text{GLP}} = V_{\text{câmara}} \times \frac{\text{concentração desejada}}{100} \tag{3.4}$$

Substituindo para 1,8%:

$$V_{\text{GLP}} = 27,04 \times \frac{1,8}{100} = 0,4867\,\text{L} \tag{3.5}$$

Convertendo para mililitros:

$$V_{\text{GLP}} = 0,4867 \times 1000 = 486,7\,\text{mL} \tag{3.6}$$

Assim, são necessários aproximadamente 487 mL de GLP para atingir 1,8% na câmara.

Para um teste mais abrangente, também foi considerado o valor de 0,5% de concentração. O cálculo é:

$$V_{\text{GLP}} = 27,04 \times \frac{0,5}{100} = 0,1352\,\text{L} = 135,2\,\text{mL} \tag{3.7}$$

Em seguida, foi necessário determinar a vazão da válvula da botija de gás para definir o tempo necessário de injeção do volume calculado. Na Figura 42, observa-se a especificação da válvula:

Figura 42 – Válvula de Gás.

Fonte: Autor (2025)

A válvula possui as seguintes especificações:

- Vazão máxima: $1\,\mathrm{kg\,h^{-1}}$
- Pressão de saída: 2,8 kPa

Sabendo que a densidade do GLP é aproximadamente:

$$\rho_{\mathrm{GLP}} \approx 2,5\,\mathrm{kg\,m^{-3}} = 0,0025\,\mathrm{kg\,L^{-1}} \tag{3.8}$$

A vazão volumétrica máxima da válvula é:

$$Q = \frac{1}{0,0025} = 400\,\mathrm{L\,h^{-1}} \tag{3.9}$$

Convertendo para $\mathrm{L\,s^{-1}}$:

$$Q = \frac{400}{3600} \approx 0,111\,\mathrm{L\,s^{-1}} \tag{3.10}$$

Tempo para injetar o volume necessário para 1,8% (487 mL)

$$t = \frac{V_{\text{GLP}}}{Q} = \frac{0,487}{0,111} \approx 4,39\,\text{s} \tag{3.11}$$

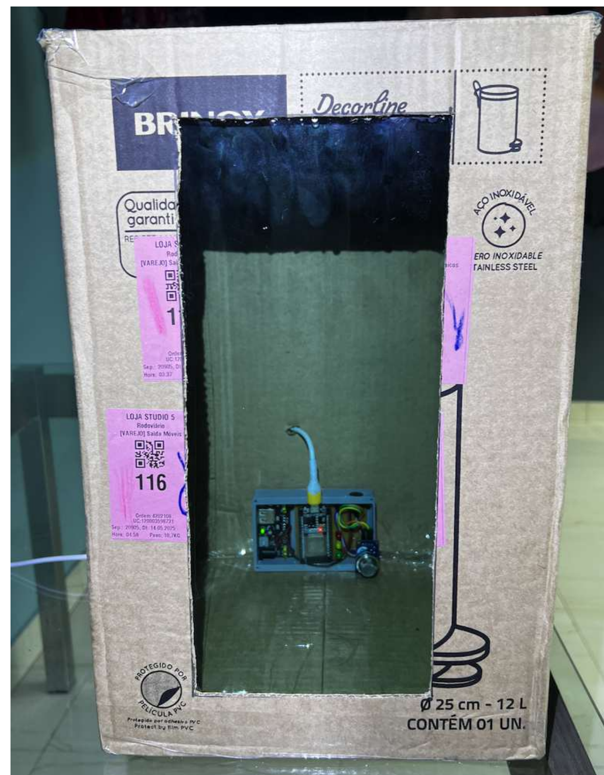
Tempo para injetar o volume necessário para 0,5% (135 mL)

$$t = \frac{0,135}{0,111} \approx 1,22\,\text{s} \tag{3.12}$$

Esses cálculos consideram a válvula com abertura total, conforme especificado pelo fabricante.

Em seguida, o protótipo foi inserido na câmara de gás, conectado via USB para monitoramento em tempo real das leituras do sensor. A Figura 43 ilustra essa configuração.

Figura 43 – Dispositivo Inserido na Câmara de Gás.



Fonte: Autor (2025)

O gás foi injetado durante o tempo previamente calculado, e, após o fechamento da válvula, foram coletadas amostras a cada minuto, durante um período de 20 minutos. As leituras foram feitas diretamente do sinal analógico de 12 bits, variando de 0 a 4095.

Os dados obtidos foram convertidos em percentuais para análise, utilizando a seguinte equação:

$$\text{Percentual} = \left(\frac{\text{Valor}_{\text{analógico}}}{4095} \right) \times 100 \tag{3.13}$$

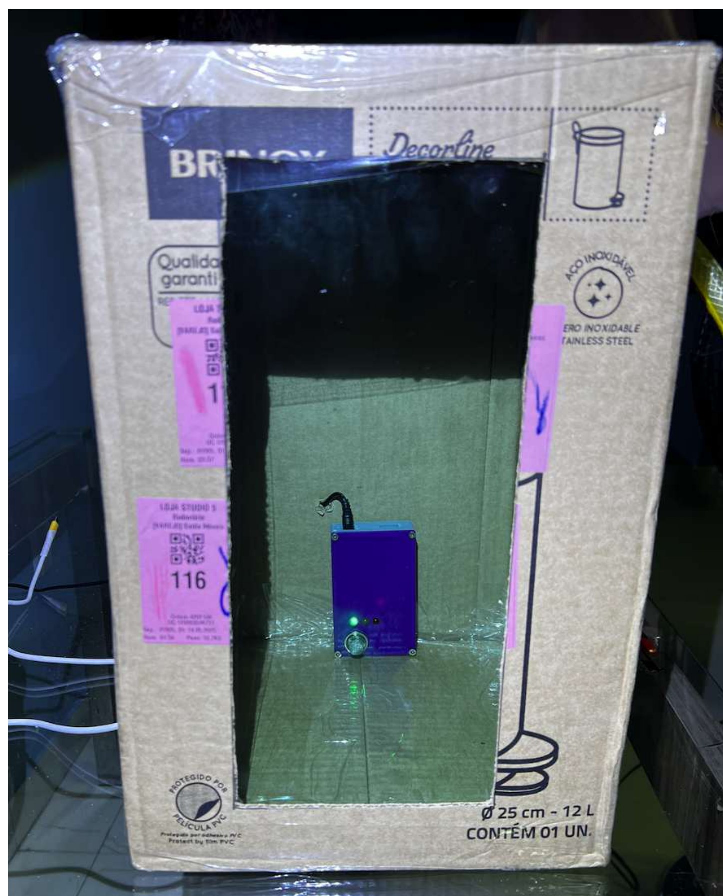
Esse procedimento foi realizado para as concentrações de 0,5% e 1,8%, permitindo a definição dos limites de alerta (verde, amarelo e vermelho). A conversão percentual foi utilizada na configuração das funções condicionais de calibração do sensor. O código correspondente encontra-se na Figura 29.

3.2.4.2 Testes com o Dispositivo Calibrado

Após a calibração, o monitoramento via USB foi encerrado, e o protótipo foi montado na sua configuração final, sendo alimentado por fonte própria e conectado à rede Wi-Fi. Em seguida, o dispositivo foi posicionado novamente na câmara de gás, realizando-se os testes finais de validação, incluindo o acionamento do aplicativo móvel e a verificação do funcionamento integral do sistema.

Na Figura 44, podemos observar o protótipo em processo de validação.

Figura 44 – Protótipo em Validação.



Fonte: Autor (2025)

4 Resultados e Discussão

Nesta seção, serão apresentados os resultados obtidos a partir dos processos descritos anteriormente. Primeiramente, será exposto o resultado do desenvolvimento físico do projeto, seguido da apresentação das telas finais do aplicativo. Por fim, será demonstrado o funcionamento do sistema como um todo, com os testes realizados e a análise do desempenho de acordo com as situações preparadas para a validação do protótipo.

4.1 Resultado do Dispositivo Sensor de Gás

Ao final dos processos, foi possível alcançar os resultados esperados com o desenvolvimento do protótipo, conforme pode ser visualizado na Figura 45. Destaca-se o baixo custo de produção, decorrente da utilização reduzida de material tanto para a impressão dos modelos 3D quanto para a manufatura da PCI. Além disso, o dispositivo se sobressai pela sua portabilidade e facilidade de instalação.

Figura 45 – Dispositivo Sensor de Gás Finalizado.

Fonte: Autor (2025)

Um desafio significativo foi o posicionamento do sensor MQ-5, fator crucial para o seu bom funcionamento, visto que o gás possui densidade maior do que o ar. Assim,

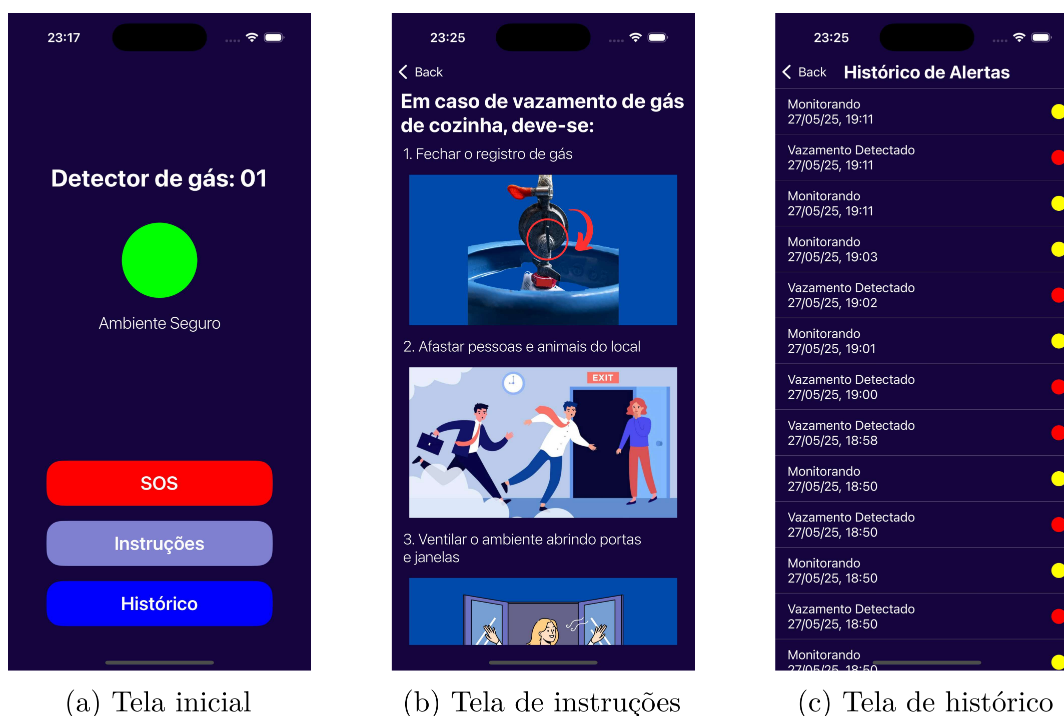
mesmo que o sensor apresentasse elevada sensibilidade, uma posição inadequada poderia impedir a detecção eficaz do gás.

Ao final deste projeto, os resultados foram satisfatórios em relação aos objetivos propostos, com o correto acionamento dos *LEDs* e do *buzzer*, além da comunicação eficiente com o aplicativo.

4.2 Resultado do Aplicativo Móvel

Ao final do desenvolvimento, foi possível obter uma versão funcional do aplicativo móvel, caracterizada pela simplicidade, facilidade de compreensão e pela execução eficiente de todas as funcionalidades propostas neste trabalho. O aplicativo conecta-se automaticamente ao dispositivo sensor via protocolo MQTT, oferece instruções sobre como agir em caso de um suposto vazamento, mantém um histórico de alertas, envia notificações e disponibiliza um botão de ligação direta para o número dos bombeiros. Além disso, apresenta uma interface principal de monitoramento, exibindo o status atual do dispositivo sensor.

Na Figura 46, são ilustradas as três principais telas do aplicativo.



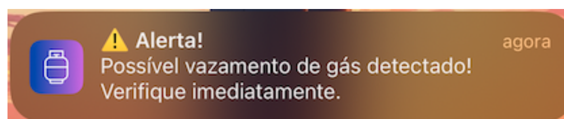
(a) Tela inicial (b) Tela de instruções (c) Tela de histórico

Figura 46 – Aplicativo Móvel

Fonte: Autor (2025).

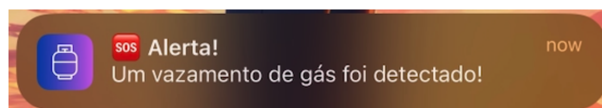
Caso o usuário não esteja com o aplicativo aberto, o sistema envia automaticamente uma notificação, de acordo com o nível de alerta. Esse comportamento pode ser observado nas Figuras: Figura 47 e Figura 48.

Figura 47 – Notificação de Alerta Amarelo



Fonte: Autor (2025).

Figura 48 – Notificação de Alerta Vermelho



Fonte: Autor (2025).

Os maiores desafios enfrentados foram o desenvolvimento da interface visual, buscando proporcionar uma melhor experiência ao usuário, e, principalmente, a implementação da conectividade com o protocolo MQTT. Para isso, foi necessário realizar uma pesquisa criteriosa por bibliotecas que não estivessem obsoletas, demandando ajustes e adaptações para viabilizar a conexão entre o cliente e o *broker*.

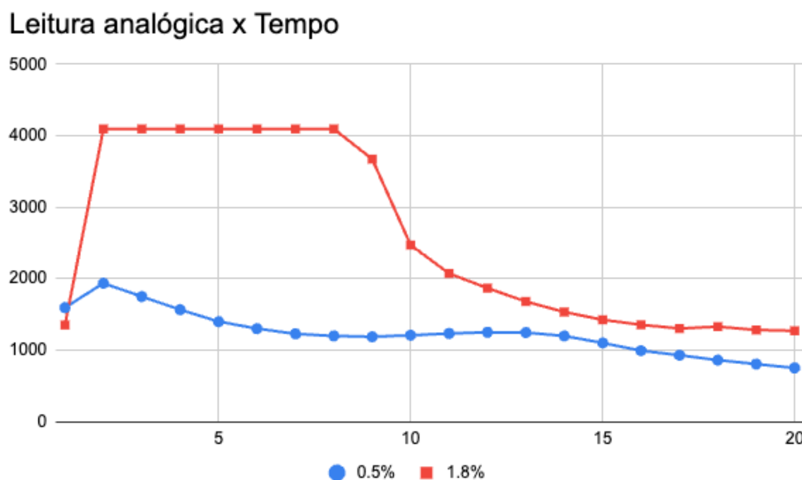
Outra dificuldade relevante foi o tratamento adequado dos dados, de modo que fossem facilmente interpretáveis e compatíveis com o protocolo. Para isso, foi necessário implementar estruturas condicionais e realizar conversões, uma vez que o MQTT opera com base no envio e recebimento de mensagens.

Uma tecnologia que contribuiu significativamente para o desenvolvimento foi o uso do GitHub, que possibilitou o versionamento do código, facilitando a visualização das alterações realizadas a cada ciclo de trabalho. Dessa forma, foi possível manter uma organização eficiente, promovendo maior controle e qualidade no desenvolvimento do *software*.

4.3 Resultado dos Testes e Validação Final

Para validar o projeto, conforme descrito no capítulo anterior, foi desenvolvida uma câmara de gás para a realização de testes em um ambiente controlado. O sensor foi inserido no interior da câmara e testado com diferentes concentrações de GLP. A amostragem de dados foi fundamental para avaliar o comportamento do sensor. Abaixo, na Figura 49, apresenta-se o resultado da leitura analógica do sensor em função do tempo. As leituras foram realizadas para as concentrações de 0,5% e 1,8% de GLP.

Figura 49 – Gráfico: Leitura x Tempo.



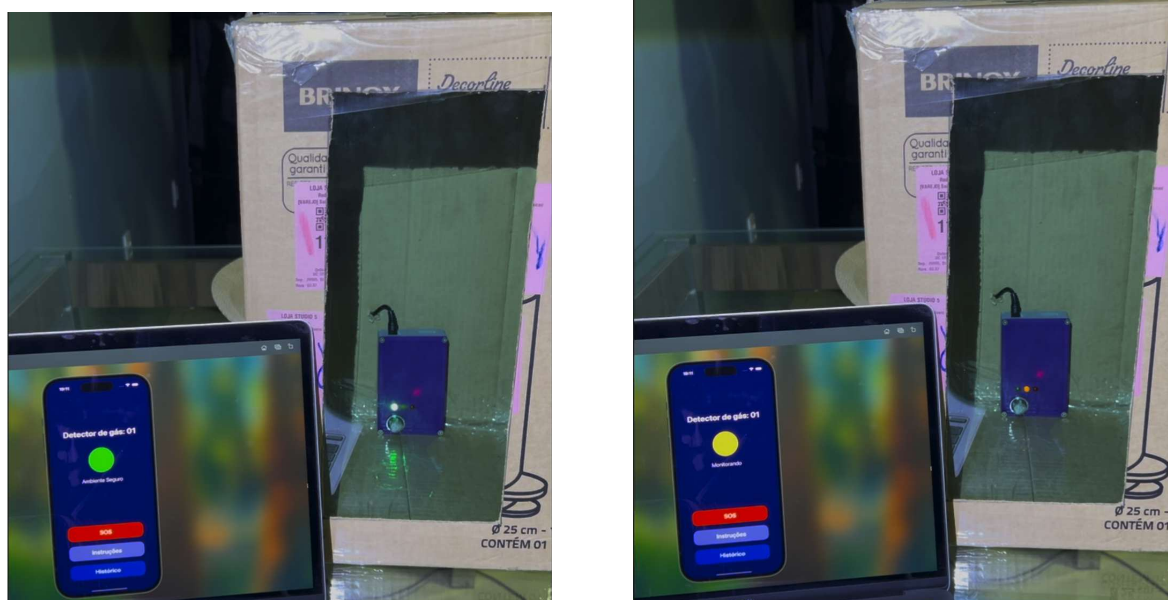
Fonte: Autor (2025)

O principal desafio foi realizar a calibração, sendo necessário o desenvolvimento de um ambiente controlado para minimizar as variáveis externas. Além disso, outro obstáculo esteve relacionado à válvula de gás: os cálculos realizados anteriormente consideraram a válvula totalmente aberta, pois apenas nessa posição há dados fornecidos pelo fabricante. Dessa forma, os testes foram realizados com essa abordagem; contudo, devido ao tamanho reduzido da câmara, ocorreu rapidamente a saturação do sensor, como pode ser observado no gráfico correspondente à concentração de 1,8%.

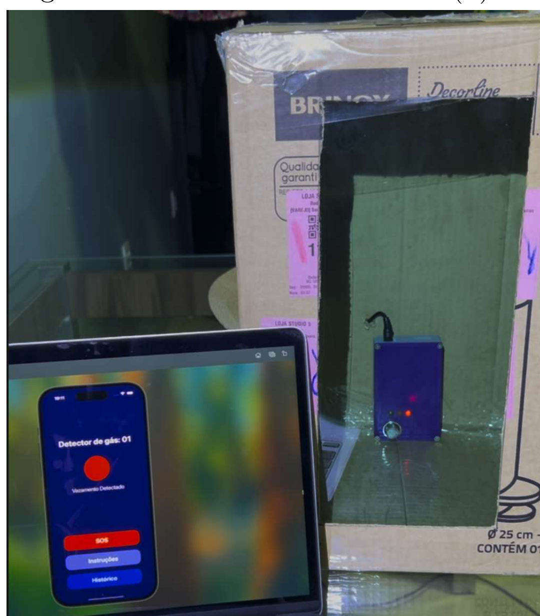
Apesar disso, tal fenômeno não comprometeu a validação do protótipo, visto que sua função principal é alertar sobre o vazamento de gás — objetivo plenamente atingido, com resposta rápida à inserção de GLP.

Para a calibração, foram utilizados os valores amostrados após a estabilização do gás dentro da câmara, exigindo um período de espera. Após essa estabilização, determinaram-se os valores de limiar para as funções responsáveis pela definição dos estados de alerta, conforme exemplificado na Figura 30, expressos em percentual.

Assim, com o dispositivo montado em sua configuração final, o aplicativo aberto e a inserção gradual de GLP, foi possível visualizar a transição entre os estados de alerta, conforme ilustrado na Figura 50.



(a) Ambiente Seguro (b) Vazamento Inicial



(c) Vazamento Crítico

Figura 50 – Teste Final

Fonte: Autor (2025)

4.4 Lista de Materiais

Com a construção do protótipo, é possível verificar o custo total dos componentes utilizados no seu desenvolvimento, com o objetivo de manter um baixo custo. Na tabela a seguir, apresenta-se a lista de materiais empregados.

Nome	**Quantidade**	**Preço (R$)**
Sensor MQ-5	1	12,73
ESP32-WROOM-32	1	40,00
Kit com 10 resistores	1	0,71
500g ABS Premium	1	27,51
Placa de fenolite cobreada (10x10 cm)	1	8,00
LEDs	3	0,40
Fonte para protoboard	1	7,12
Fonte 5V	1	16,06

Assim, o custo total dos materiais resulta em R$ 113,33, um valor acessível para um sensor de gás, alcançando, portanto, o objetivo de ser uma solução de baixo custo.

Para fins de comparação, apresenta-se, na Figura 51, um sensor de gás comercial, cujo valor é de R$ 279,99.

Figura 51 – Sensor de Gás Comercial Disponível no Mercado.

Fonte: Autor (2025)

Além de possuir um valor significativamente mais elevado, esse modelo comercial não oferece recursos como monitoramento remoto ou envio de notificações, o que o torna uma solução limitada localmente e com menor custo-benefício, quando comparada ao protótipo desenvolvido neste trabalho.

5 Conclusão

Neste trabalho, foi desenvolvido um protótipo utilizando conceitos de modelagem e impressão 3D para o desenvolvimento de sua estrutura, com o objetivo de garantir baixo custo e alta facilidade de prototipagem. Além disso, o projeto envolveu o desenvolvimento de uma placa eletrônica e de um aplicativo móvel, visando criar uma interface amigável e intuitiva para o usuário.

O protótipo demonstrou ser capaz de alertar o usuário em caso de vazamento de Gás Liquefeito de Petróleo (GLP), por meio de sinais visuais e sonoros locais, bem como notificações via aplicativo móvel, permitindo o monitoramento contínuo e orientando sobre as ações a serem tomadas diante da situação. A proposta visa promover educação e segurança, contribuindo para a redução dos riscos associados a vazamentos de gás.

Durante o desenvolvimento, constatou-se certa dificuldade na criação de um ambiente controlado para a calibração do sensor, o que resultou na saturação deste nos momentos iniciais dos testes. Contudo, foram adotadas as medidas necessárias para obter dados válidos, assegurando que esse fator não comprometesse a validação do protótipo. A saturação, inclusive, evidenciou a alta sensibilidade do sensor, uma característica importante para a detecção precoce de vazamentos.

Uma limitação deste projeto foi a necessidade de manter o sensor próximo à fonte de gás, a fim de obter uma resposta mais rápida, aspecto que pode ser aprimorado em futuras versões.

Apesar das dificuldades mencionadas, o projeto foi desenvolvido conforme o planejado, obtendo os resultados esperados e integrando diferentes áreas da engenharia. As etapas mecânica, elétrica e computacional foram abordadas de forma articulada, com a aplicação de conceitos de Internet das Coisas (IoT) e a implementação bem-sucedida do protocolo MQTT, garantindo a conectividade entre o dispositivo e o aplicativo. Assim, os objetivos propostos foram plenamente alcançados.

5.1 Trabalhos Futuros

Após a análise dos resultados obtidos, são apresentadas as seguintes sugestões de melhorias para trabalhos futuros:

- Aumentar a quantidade de sensores, tanto para a detecção de GLP quanto de outros gases, ampliando o campo de aplicabilidade do sistema;

- Implementar sensores mais precisos e robustos, aumentando a confiabilidade do monitoramento;
- Adicionar um sensor de temperatura para auxiliar na detecção de incêndios ou explosões;
- Explorar outros protocolos de comunicação visando maior eficiência e confiabilidade no sistema;
- Desenvolver uma versão do aplicativo para a plataforma Android;
- Projetar uma placa totalmente integrada, tornando o protótipo mais compacto e eficiente;
- Conectar o aplicativo e o dispositivo físico a um banco de dados para registro e geração de relatórios;
- Melhorar o design do projeto, visando maior ergonomia e estética;
- Aprimorar o aplicativo, adicionando funcionalidades extras, como o envio automático de dados diretamente para as autoridades competentes;
- Desenvolver uma página web para melhor visualização dos dados, ampliando o ecossistema do sistema;
- Integrar atuadores, como ventiladores para ventilar o ambiente, ou válvulas solenoides conectadas diretamente à botija, permitindo o fechamento rápido em caso de vazamento.

Referências

Apple Inc. *Swift — A powerful and intuitive programming language*. 2023. <https://developer.apple.com/swift/>. Acesso em: 20 maio 2025. Citado na página 23.

Apple Inc. *Xcode - Apple Developer*. 2023. <https://developer.apple.com/xcode/>. Acesso em: 20 maio 2025. Citado na página 24.

Arducore. *Sensor de Gás MQ-5 para Arduino - GLP (Gás de Cozinha) e Gás Natural*. 2025. <https://www.arducore.com.br/sensor-de-gas-mq-5-para-arduino-glp-gas-de-cozinha-e-gas-natural>. Acesso em: 20 maio 2025. Citado na página 29.

AZURE, M. *O que é o Desenvolvimento de Aplicativos Móveis?* 2024. Acessado em: 29 nov. 2024. Disponível em: <https://azure.microsoft.com/pt-br/resources/cloud-computing-dictionary/what-is-mobile-app-development/>. Citado na página 23.

Baú da Eletrônica. *Buzzer 5V / 12mm (com oscilador interno)*. 2025. <https://www.baudaeletronica.com.br/produto/buzzer-5v.html>. Acesso em: 24 maio 2025. Citado na página 34.

BORGES, M. F.; SILVA, M. M. da. Fabricação digital no brasil e as possibilidades de mudança de paradigma no setor da construção civil. *Ambiente Construído*, ANTAC, v. 16, n. 4, p. 79–91, 2016. Disponível em: <https://www.antac.org.br/revista/arquivos/artigos/747.pdf>. Citado na página 25.

CELANI, G.; ORCIUOLI, A. A tecnologia desvenda gaudí: a fabricação digital e o patrimônio histórico. *Revista AU*, v. 17, dezembro 2008. Disponível em: https://www.au.com.br. Citado na página 25.

COOMBS, C. F. *Printed Circuits Handbook*. 5th. ed. New York: McGraw-Hill, 2001. ISBN 0-07-135016-0. Citado na página 25.

Escola do Gás. *Análise dos Impactos do Enchimento Fracionado de Recipientes de GLP por Parte dos Distribuidores*. 2018. Disponível em: <https://www.sindigas.org.br/Download/TPC%20ANP%2007-2018/ANEXO%20VIII_AN%C3%81LISE%20DOS%20IMPACTOS%20DO%20ENCHIMENTO%20FRACIONADO%20DE%20RECIPIENTES%20DE%20GLP%20POR%20PARTE%20DOS%20DISTRIBUIDORES.pdf>. Citado 2 vezes nas páginas 16 e 17.

HANWEI. *MQ-5 Gas Sensor Technical Data*. [S.l.], 2024. Datasheet técnico do sensor MQ-5, incluindo características e especificações. Disponível em: <http://www.hwsensor.com>. Citado 2 vezes nas páginas 29 e 30.

JIN, R. et al. Five-year epidemiology of liquefied petroleum gas-related burns. *Burns*, Elsevier Ltd, v. 44, p. 210–217, 2 2018. ISSN 18791409. Citado na página 13.

KERSCHBAUMER, R. *Microcontroladores*. 2013. Apostila didática. Disponível em formato PDF, utilizada na disciplina de Microcontroladores do curso de Engenharia de Controle e Automação. Citado na página 22.

KOLBAN, N. *Kolban's Book on ESP32*. Texas, USA: Self-published, 2017. Available online at <https://leanpub.com/kolban-ESP32>. Citado na página 30.

LAB, D. *Conheça 13 dos melhores softwares de modelagem 3D!* 2023. Acesso em: 30 nov. 2024. Disponível em: <https://3dlab.com.br/10-softwares-de-modelagem-3d/>. Citado na página 27.

MACIEL, J. *EmergencyGasEsp32 - Repositório no GitHub*. 2025. <https://github.com/Jadsonsmxr/EmergencyGasEsp32/>. Acessado em: 20 maio 2025. Citado na página 48.

MACIEL, J. *Repositório do Projeto no GitHub*. 2025. <https://github.com/Jadsonsmxr/EmergencyGas>. Acessado em: 20 maio 2025. Citado na página 51.

Mercado Livre. *LED Amarelo Vermelho Verde 5mm Difuso Kit 50pcs de Cada*. 2025. <https://produto.mercadolivre.com.br/MLB-5111805914-led-amarelo-vermelho-verde-5mm-difuso-kit-50pcs-de-cada-_JM>. Acesso em: 20 maio 2025. Citado na página 33.

MOREIRA, A. M. *Segurança na Utilização de Gás Liquefeito de Petróleo*. 2015. Documento obtido a partir do arquivo PDF fornecido pelo autor. Disponível em: <https://ambiental.ufes.br/sites/ambiental.ufes.br/files/field/anexo/seguranca_na_utilizacao_de_gas_liquefeito_de_petroleo_-_alessandro_marcio_moreira.pdf>. Citado na página 15.

PMESP, C. de Bombeiros do Estado de S. P. *Corpo de Bombeiros dá dicas contra o vazamento de gás*. 2023. Acessado em: 15 de outubro de 2024. Disponível em: <https://www.saopaulo.sp.gov.br/spnoticias/ultimas-noticias/corpo-de-bombeiros-da-dicas-contra-o-vazamento-de-gas/>. Citado na página 13.

RAISA. *Comparação Entre Métodos de Fabricação de PCB: Fresagem CNC, Corrosão e Serigrafia*. 2024. Acessado em: 29 nov. 2024. Disponível em: <https://blog.raisa.com.br/comparacao-entre-metodos-de-fabricacao-de-pcb-fresagem-cnc-corrosao-e-serigrafia/>. Citado na página 26.

RoboCore. *Fonte Ajustável para Protoboard*. 2025. <https://www.robocore.net/regulador-de-tensao/fonte-ajustavel-para-protoboard>. Acesso em: 24 maio 2025. Citado na página 35.

RODRIGUES CLEBER E SILVA, B. Impressão 3d: Perspectivas e aplicações no setor industrial. *Revista de Tecnologia Industrial*, v. 10, n. 2, p. 100–120, 2017. Disponível em: <https://example.com/artigo-impressao3d>. Citado 2 vezes nas páginas 27 e 28.

SANTOS, B. P. et al. Internet das coisas: da teoriaa prática. *Minicursos SBRC-Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos*, 2016. Citado 4 vezes nas páginas 17, 18, 19 e 20.

Saravati Eletrônica. *Placa ESP32 WiFi Bluetooth DevKit V1 30 Pinos*. 2025. <https://www.saravati.com.br/placa-esp32-wifi-bluetooth-devkit-v1-30-pinos.html>. Acesso em: 20 maio 2025. Citado na página 31.

Sindigás. *O que é GLP*. 2024. Acessado em: 12 de novembro de 2024. Disponível em: <https://www.sindigas.org.br/?page_id=12>. Citado na página 15.

VOLPATO, N. *Manufatura Aditiva: Tecnologias e Aplicações da Impressão 3D*. São Paulo: Edgard Blücher, 2017. ISBN 9788521211501. Citado 2 vezes nas páginas 27 e 28.

WENDLING, M. *Sensores V2.0*. [S.l.], 2010. Material de apoio técnico sobre sensores. Citado na página 21.

Apêndices

APÊNDICE A – Desenhos CAD 2D e 3D

Este apêndice contém todos os desenhos CAD 2D e 3D desenvolvidos neste projeto.

R2,00
⌀3,00
9,00
⌀5,67
32,00
⌀19,10
18,00
30,50
13,05
66,50
Vista Frontal

106,00
2,00
Vista L. Direita

Vista Isométrica

Projetista: Jadson de souza maciel	
Data: 24/ 05/ 2025	Escala: 1 : 1
Projeto: Tampa	Diedro:
Universidade do estado do amazonas	Folha: 1/1

R2,00
54,50
6,00
3,50
3,50
$\varnothing$3,00
32,50
4,00
4,00
3,00
64,20
66,50
1,30

Vista Frontal

7,00
2,00
106,00
$\varnothing$12,00

Vista L. Direita

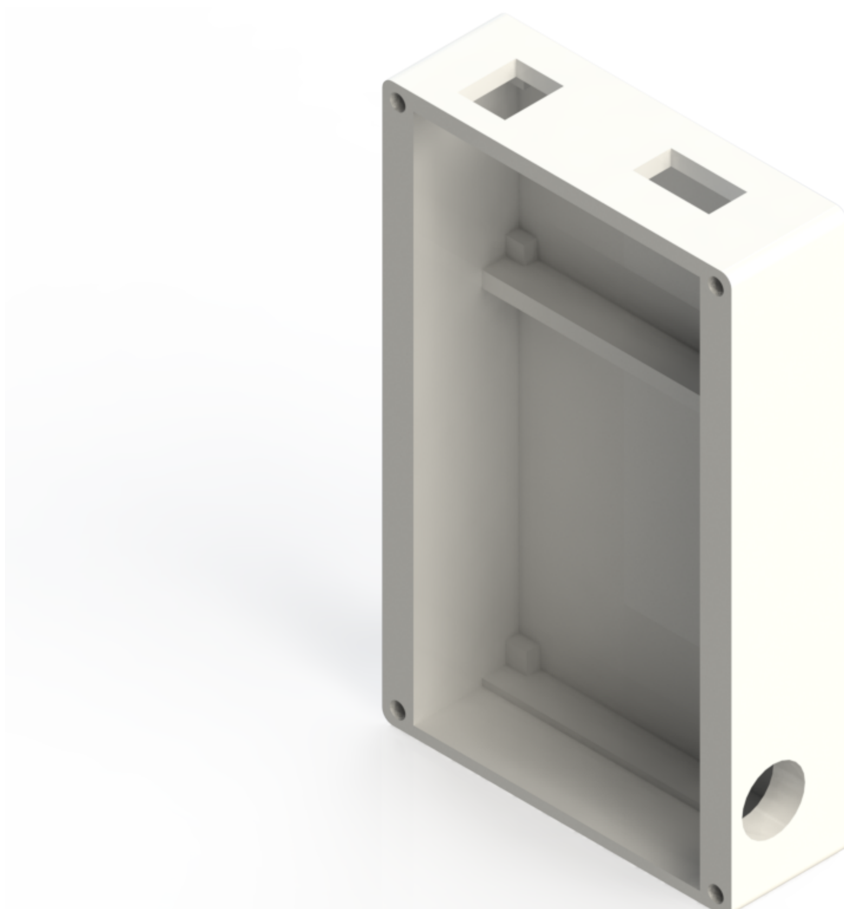
4,30
9,10
11,20
14,50
7,30
8,20
2,50
20,00
10,00
12,50

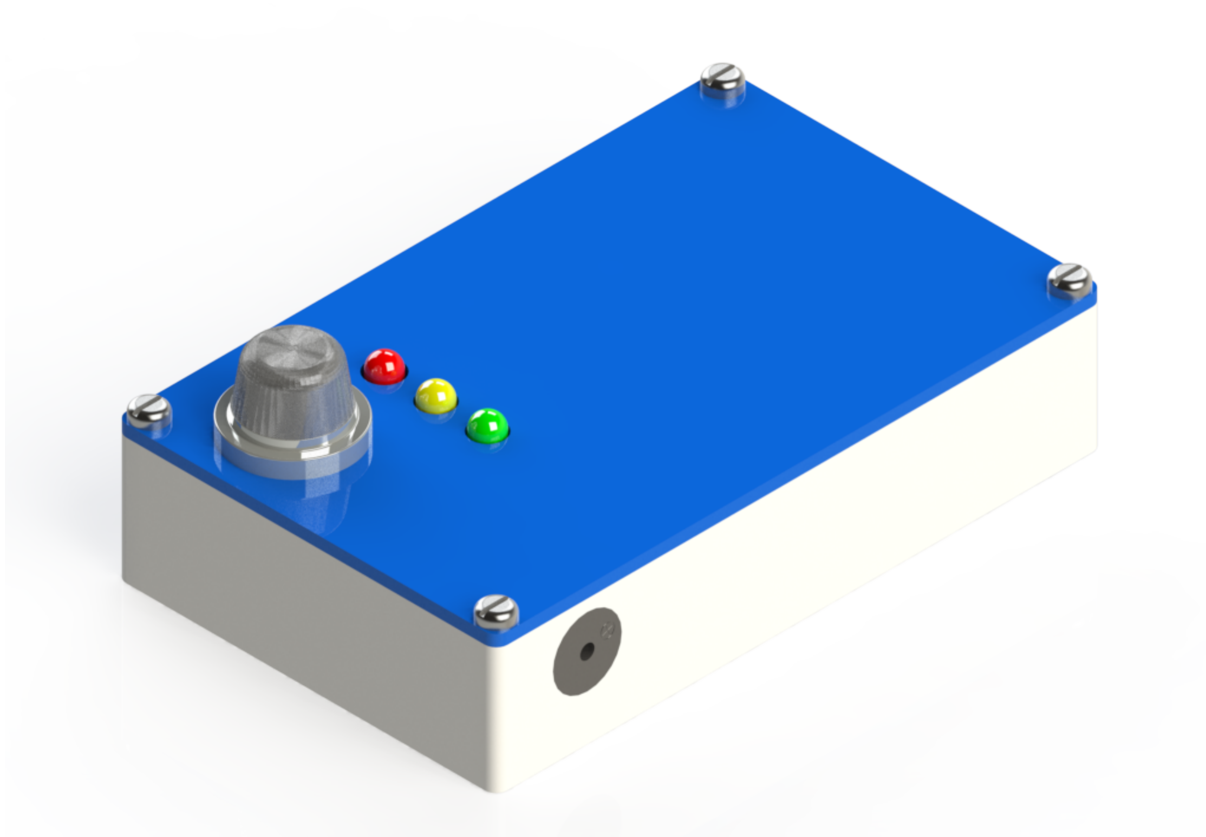
Vista Inferior

Vista Isométrica

Projetista: Jadson de souza maciel

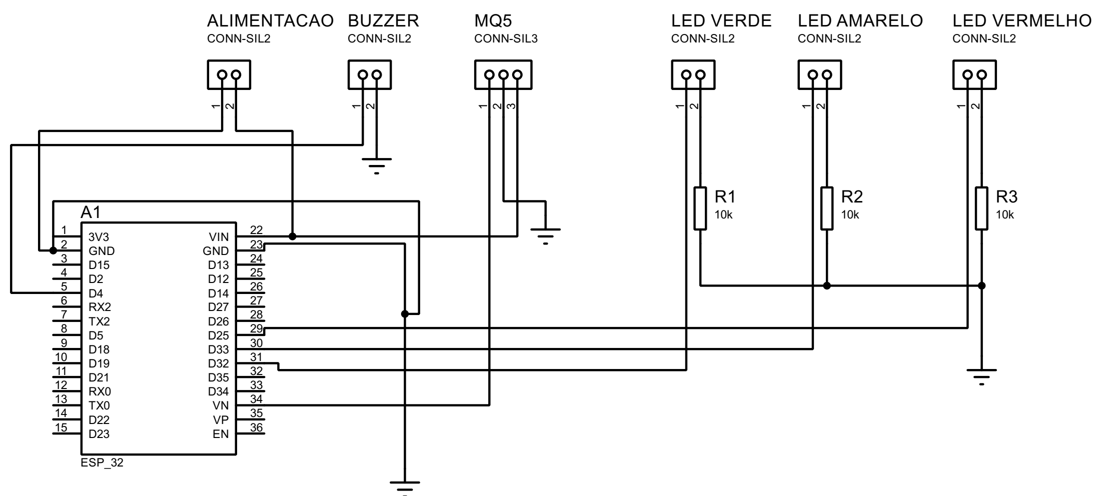
Data: 24/ 05/ 2025	Escala: 1 : 1
Projeto: Carcaça	Diedro:
Universidade do estado do amazonas	Folha: 1/1

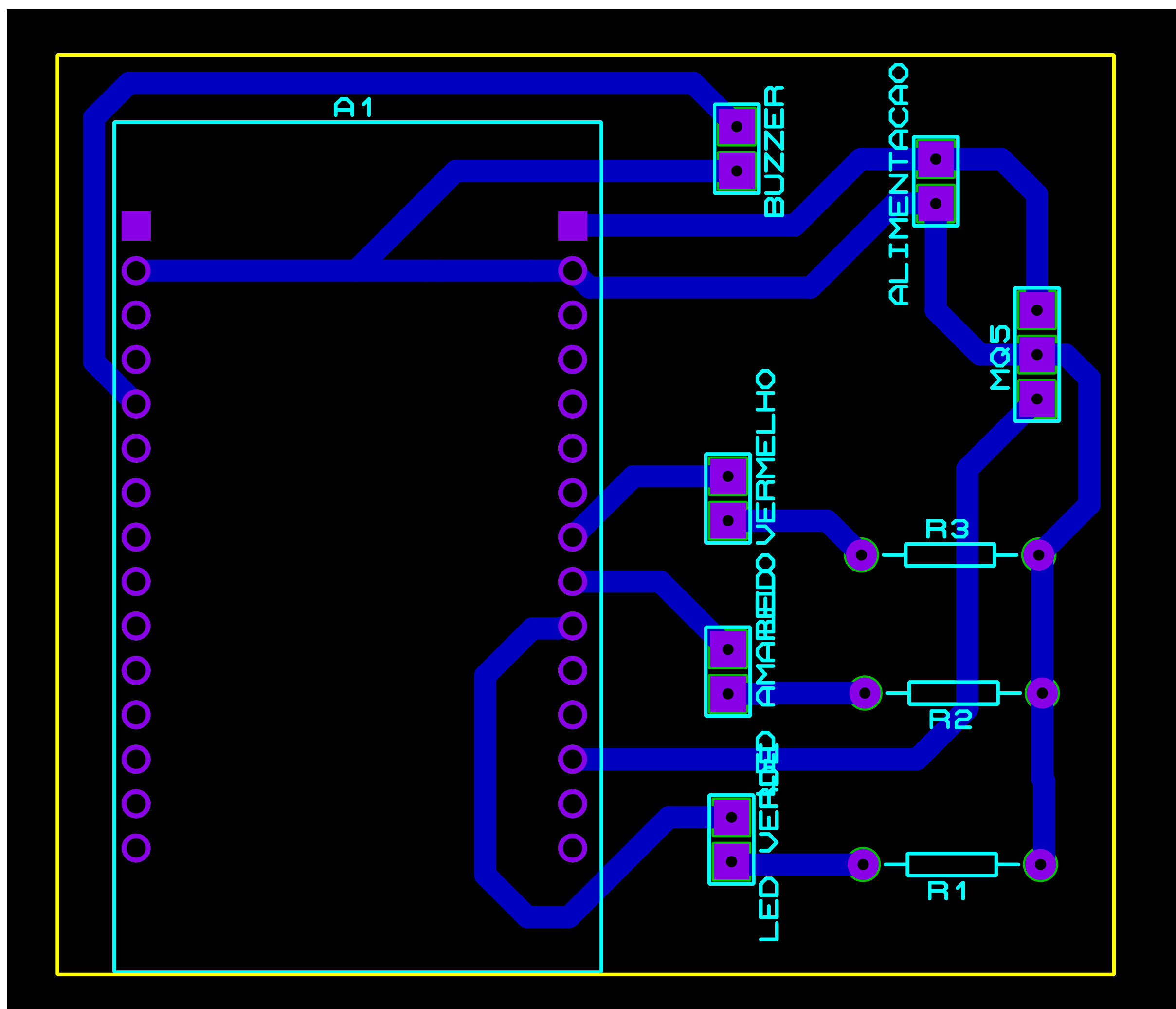


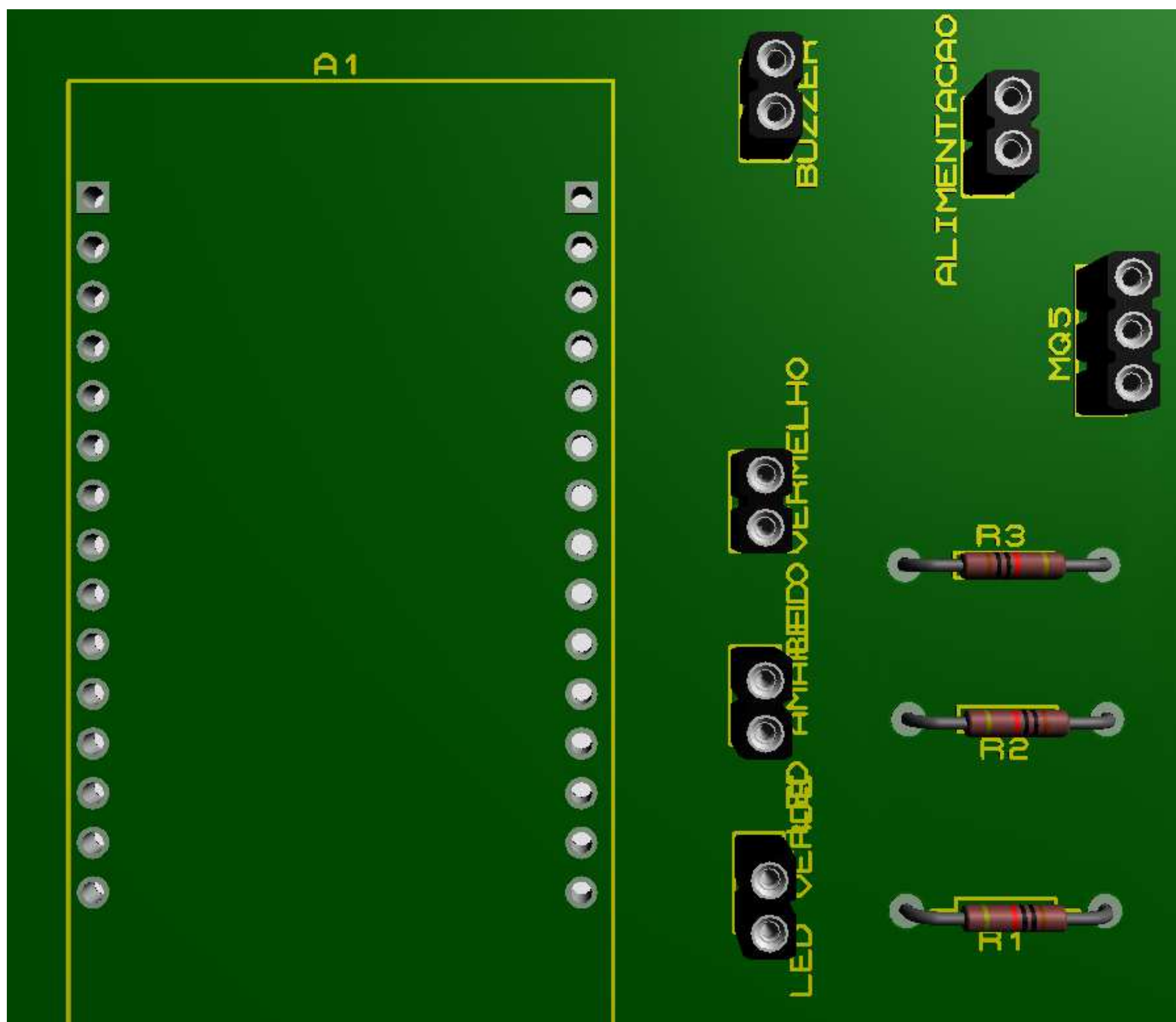


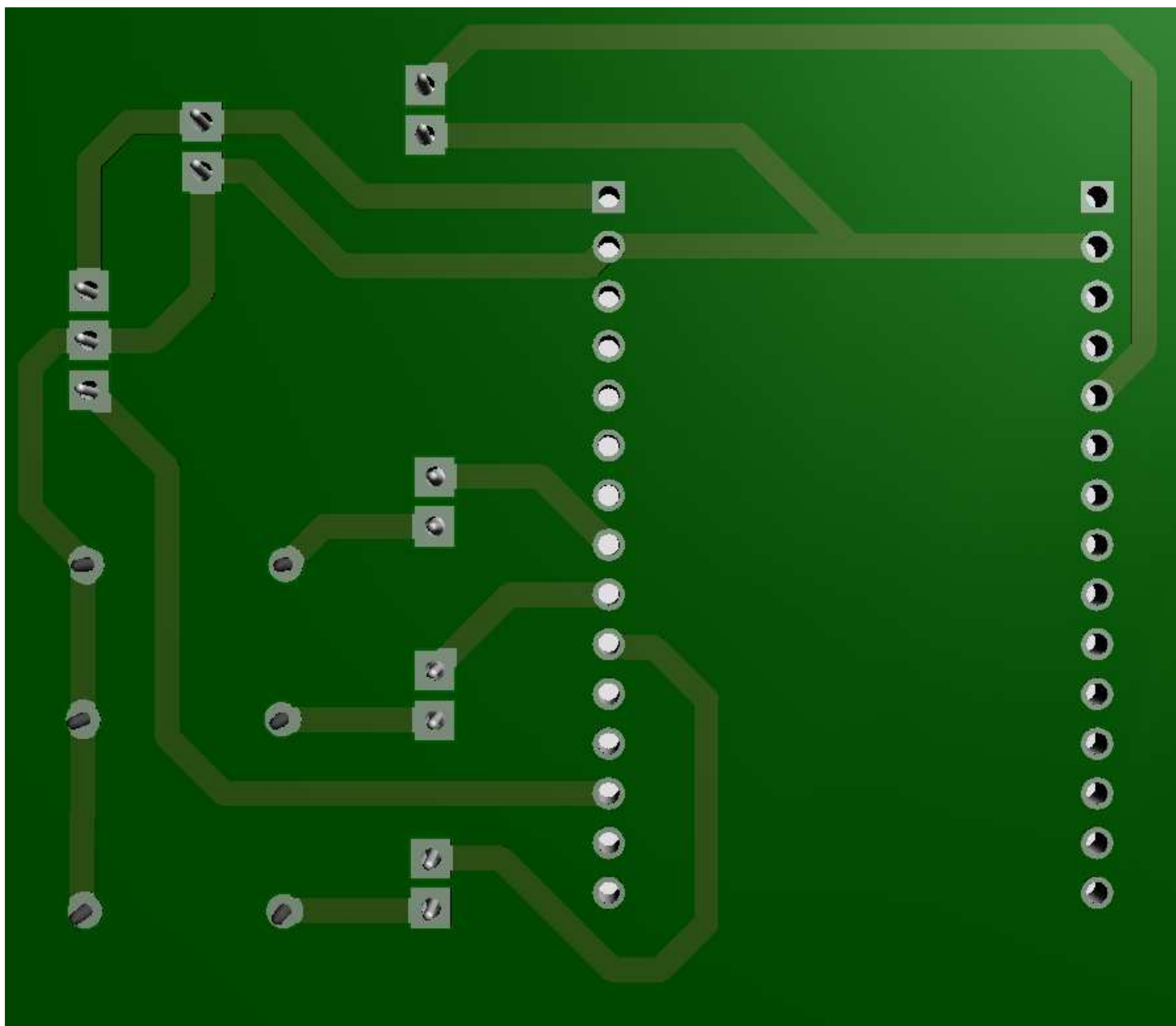
APÊNDICE B – Projetos da Placa de Circuito Impresso

Neste apêndice, são apresentados os projetos desenvolvidos no Proteus, incluindo o esquemático, o layout da PCI e os desenhos em 3D.




A1
BUZZER
ALIMENTACAO
MQ5
LED VERMELHO
LED AMARELO
LED VERDE
R3
R2
R1


A1
BUZZER
ALIMENTACAO
MQ5
R3
R2
R1



APÊNDICE C – Programas do aplicativo IOS

Os principais programas estão inseridos na seguinte ordem:

- **Cell**
- **ColorHelper**
- **HistoryManager**
- **MQTT**
- **Notifications**
- **HistoryViewController**
- **HomeViewController**
- **InstructionViewController**

```
1  //
2  //   Cell.swift
3  //   EmergencyGas
4  //
5  //   Created by Jadson Maciel on 24/04/25.
6  //
7
8  import UIKit
9
10 class AlertaCell: UITableViewCell {
11
12
13      lazy var ledView: UIView = {
14         let ledView = UIView()
15         ledView.backgroundColor = .green
16         ledView.translatesAutoresizingMaskIntoConstraints = false
17         ledView.layer.cornerRadius = 10
18         ledView.clipsToBounds = true
19         return ledView
20     }()
```

```
21
22     private lazy var mensagemLabel: UILabel = {
23         let mensagemLabel = UILabel()
24         // Label
25         mensagemLabel.translatesAutoresizingMaskIntoConstraints
              = false
26         mensagemLabel.textColor = .white
27         mensagemLabel.numberOfLines = 0
28
29         return mensagemLabel
30     }()
31
32
33     override init(style: UITableViewCell.CellStyle,
          reuseIdentifier: String?) {
34         super.init(style: style, reuseIdentifier:
              reuseIdentifier)
35
36         backgroundColor = .clear
37         selectionStyle = .none
38         //setupcellConstraints()
39         addSubview()
40
41         // Constraints
42         NSLayoutConstraint.activate([
43             ledView.widthAnchor.constraint(equalToConstant: 20),
44             ledView.heightAnchor.constraint(equalToConstant: 20),
45             ledView.trailingAnchor.constraint(equalTo:
                  contentView.trailingAnchor, constant: -16),
46             ledView.centerYAnchor.constraint(equalTo:
                  contentView.centerYAnchor),
47 //
48             mensagemLabel.centerYAnchor.constraint(equalTo:
                  contentView.centerYAnchor),
49             mensagemLabel.topAnchor.constraint(equalTo:
                  contentView.topAnchor, constant: 10),
50             mensagemLabel.bottomAnchor.constraint(equalTo:
                  contentView.bottomAnchor, constant: -10),
51             mensagemLabel.leadingAnchor.constraint(equalTo:
                  contentView.leadingAnchor, constant: 16)
52         ])
53
```

```
54
55      }
56
57      required init?(coder: NSCoder) {
58          fatalError("init(coder:) has not been implemented")
59      }
60
61
62      private func addSubview() {
63          contentView.addSubview(ledView)
64          contentView.addSubview(mensagemLabel)
65      }
66
67
68      func configurarCom(alerta: Alerta) {
69          let formatter = DateFormatter()
70          formatter.dateStyle = .short
71          formatter.timeStyle = .short
72
73          ledView.backgroundColor = trocadeCor(alerta.cor)
74          mensagemLabel.text =
              "\(alerta.mensagem)\n\(formatter.string(from:
              alerta.data))"
75
76
77      }
78      override func awakeFromNib() {
79          super.awakeFromNib()
80      }
81
82  }
```

```
1       //
2   //   ColorHelper.swift
3   //   EmergencyGas
4   //
5   //   Created by Jadson Maciel on 22/03/25.
6   //
7
8   import UIKit
9
10  func trocadeCor(_ message: String) -> UIColor {
```

```
11      switch message.lowercased() {
12      case "green":
13          return .green
14      case "red":
15          return .red
16      case "yellow":
17          return .yellow
18      default:
19          return .green
20      }
21  }
22
23  func trocadeStatus(_ message: String) -> String {
24      switch message.lowercased() {
25      case "green":
26          return "Ambiente Seguro"
27      case "red":
28          return "Vazamento Detectado"
29      case "yellow":
30          return "Monitorando"
31      default:
32          return "Ambiente Seguro"
33      }
34  }
35
36  func Notification(_ message: String) {
37      switch message.lowercased() {
38
39      case "yellow":
40          sendNotification(title: "Alerta!", body: "Possivel
               vazamento de gas detectado! Verifique imediatamente.")
41      case "red":
42          sendNotification(title: "Alerta!", body: "Um vazamento
               de gas foi detectado!")
43      default:
44          print ("Sem necessidade de notificacao")
45
46      }
47  }
```

```
1  //
2  //  HistoryManager.swift
```

```
//  EmergencyGas
//
//  Created by Jadson Maciel on 24/04/25.
//

import Foundation

struct Alerta: Codable {
    let id: UUID

    let data: Date
    let mensagem: String
    let cor: String
}


class AlertaManager {
    static let shared = AlertaManager()

    private let chave = "historico_alertas"
    private(set) var historico: [Alerta] = []

    private init() {
        carregar()
    }

    func adicionarAlerta( mensagem: String, cor: String) {
        let novo = Alerta(id: UUID(), data: Date(), mensagem:
           mensagem, cor: cor)
        historico.append(novo)
        salvar()
    }

    private func salvar() {
        if let dados = try? JSONEncoder().encode(historico) {
            UserDefaults.standard.set(dados, forKey: chave)
        }
    }

    private func carregar() {
        if let dados = UserDefaults.standard.data(forKey: chave),
           let alertas = try?
```

```
                JSONDecoder().decode([Alerta].self, from: dados) {
44              historico = alertas
45          }
46      }
47  }
```

```
1   //
2   //  MQTT.swift
3   //  EmergencyGas
4   //
5   //  Created by Jadson Maciel on 22/03/25.
6   //
7   
8   import CocoaMQTT
9   import UIKit
10  
11  let myCert = "myCert"
12  
13  extension HomeViewController: CocoaMQTTDelegate {
14  
15  
16      func mqttUrlSession(_ mqtt: CocoaMQTT, didReceiveTrust
            trust: SecTrust, didReceiveChallenge challenge:
            URLAuthenticationChallenge, completionHandler: @escaping
            (URLSession.AuthChallengeDisposition, URLCredential?) ->
            Void){
17          if (challenge.protectionSpace.authenticationMethod ==
                NSURLAuthenticationMethodServerTrust) {
18  
19              let certData = Data(base64Encoded: myCert as String)!
20  
21              if let trust = challenge.protectionSpace.serverTrust,
22                  let cert = SecCertificateCreateWithData(nil,
                        certData as CFData) {
23                  let certs = [cert]
24                  SecTrustSetAnchorCertificates(trust, certs as
                        CFArray)
25  
26          completionHandler(
27          URLSession.AuthChallengeDisposition.useCredential,
                URLCredential(trust: trust))
28                  return
```

```
29                  }
30              }
31
32  completionHandler(URLSession.AuthChallengeDisposition.cancelAuthentication(
        nil)
33
34          }
35
36
37          func mqtt(_ mqtt: CocoaMQTT, didConnectAck ack:
                CocoaMQTTConnAck) {
38              TRACE("ack: \(ack)")
39
40              if ack == .accept {
41                  mqtt.subscribe("emergencygas")
42              }
43          }
44
45          func mqtt(_ mqtt: CocoaMQTT, didStateChangeTo state:
                CocoaMQTTConnState) {
46              TRACE("new state: \(state)")
47          }
48
49          func mqtt(_ mqtt: CocoaMQTT, didPublishMessage message:
                CocoaMQTTMessage, id: UInt16) {
50              TRACE("message: \(message.string.description), id:
                    \(id)")
51          }
52
53          func mqtt(_ mqtt: CocoaMQTT, didPublishAck id: UInt16) {
54              TRACE("id: \(id)")
55          }
56
57          func mqtt(_ mqtt: CocoaMQTT, didReceiveMessage message:
                CocoaMQTTMessage, id: UInt16 ) {
58              TRACE("message: \(message.string.description), id:
                    \(id)")
59              let mensagemRecebida = message.string ?? "default"
60              let novaCor = trocadeCor(mensagemRecebida)
61              let novoStatus = trocadeStatus(mensagemRecebida)
62
63              Notification(mensagemRecebida)
```

```
64          DispatchQueue.main.async {
65                  self.statusView.backgroundColor = novaCor
66                  self.statusLabel.text = novoStatus
67
68
69              }
70          if mensagemRecebida != "green" {
71              AlertaManager.shared.adicionarAlerta(mensagem:
                    novoStatus, cor: mensagemRecebida)
72              NotificationCenter.default.post(name:
                    NSNotification.Name("NovoAlerta"), object: nil)
73          }
74
75      }
76
77      func mqtt(_ mqtt: CocoaMQTT, didSubscribeTopics success:
            NSDictionary, failed: [String]) {
78          TRACE("subscribed: \(success), failed: \(failed)")
79      }
80
81      func mqtt(_ mqtt: CocoaMQTT, didUnsubscribeTopics topics:
            [String]) {
82          TRACE("topic: \(topics)")
83      }
84
85      func mqttDidPing(_ mqtt: CocoaMQTT) {
86          TRACE()
87
88      }
89
90      func mqttDidReceivePong(_ mqtt: CocoaMQTT) {
91          TRACE()
92      }
93
94      func mqttDidDisconnect(_ mqtt: CocoaMQTT, withError err:
            Error?) {
95          TRACE("\(err.description)")
96      }
97
98
99      func TRACE(_ message: String = "", fun: String = #function) {
100         let names = fun.components(separatedBy: ":")
```

```
101         var prettyName: String
102         if names.count == 2 {
103             prettyName = names[0]
104         } else {
105             prettyName = names[1]
106         }
107 
108         if fun == "mqttDidDisconnect(_:withError:)" {
109             prettyName = "didDisconnect"
110         }
111 
112         print("[TRACE] [\(prettyName)]: \(message)")
113     }
114 }
115 
116 extension Optional {
117     // Unwrap optional value for printing log only
118     var description: String {
119         if let self = self {
120             return "\(self)"
121         }
122         return ""
123     }
124 }
```

```
1 //
2 //  Notifications.swift
3 //  EmergencyGas
4 //
5 //  Created by Jadson Maciel on 29/03/25.
6 //
7 
8 
9 import UIKit
10 
11 func sendNotification(title: String, body: String) {
12     let content = UNMutableNotificationContent()
13     content.title = title        // Titulo da notificacao
14     content.body = body          // Texto da notificacao
15     content.sound = .default     // Som padrao ao receber a
       notificacao
16 
```

```
17      // Define um tempo para exibir a notificacao (1 segundo
           depois)
18      let trigger =
           UNTimeIntervalNotificationTrigger(timeInterval: 1,
           repeats: false)
19
20      // Criando a notificacao
21      let request = UNNotificationRequest(identifier:
           UUID().uuidString, content: content, trigger: trigger)
22
23      // Adicionando a notificacao no sistema
24      UNUserNotificationCenter.current().add(request) { error in
25          if let error = error {
26              print("Erro ao agendar notificacao:
                   \(error.localizedDescription)")
27          } else {
28              print("Notificacao enviada com sucesso!")
29          }
30      }
31  }
```

```
1   //
2   //  HistoryViewController.swift
3   //  EmergencyGas
4   //
5   //  Created by Jadson Maciel on 24/04/25.
6   //
7
8   import UIKit
9
10  class HistoryViewController: UIViewController,
        UITableViewDataSource, UITableViewDelegate {
11      lazy var tabela: UITableView = {
12          let tabela = UITableView()
13          tabela.register(AlertaCell.self, forCellReuseIdentifier:
               "celula")
14
15
16          tabela.backgroundColor = .clear
17          view.backgroundColor = .background
18
19          tabela.separatorColor = .gray
```

```
20          tabela.separatorInset = .zero
21          tabela.dataSource = self
22          tabela.delegate = self
23          //tabela.register(UITableViewCell.self,
               forCellReuseIdentifier: "celula")
24          tabela.frame = view.bounds
25          return tabela
26      }()
27
28      private var alertas: [Alerta] = []
29
30
31
32      override func viewDidLoad() {
33          super.viewDidLoad()
34          NotificationCenter.default.addObserver(self, selector:
               #selector(atualizarTabela), name:
               NSNotification.Name("NovoAlerta"), object: nil)
35          alertas = AlertaManager.shared.historico.reversed()
36          titleBar()
37
38          view.addSubview(tabela)
39      }
40
41
42
43      private func titleBar () {
44          let titleLabel = UILabel()
45          titleLabel.text = "Historico de Alertas"
46          titleLabel.font = UIFont.systemFont(ofSize: 24, weight:
               .bold)
47          titleLabel.textColor = .white
48          navigationItem.titleView = titleLabel
49      }
50
51      @objc func atualizarTabela() {
52          alertas =
               Array(AlertaManager.shared.historico.reversed())
53          tabela.reloadData()
54      }
55      // MARK: - UITableViewDataSource
56      func tableView(_ tableView: UITableView,
```

```
        numberOfRowsInSection section: Int) -> Int {
57          return alertas.count
58      }
59
60      func tableView(_ tableView: UITableView, cellForRowAt
        indexPath: IndexPath) -> UITableViewCell {
61          let alerta = alertas[indexPath.row]
62
63          let cell = tableView.dequeueReusableCell(withIdentifier:
            "celula", for: indexPath) as! AlertaCell
64              cell.configurarCom(alerta: alerta)
65
66          return cell
67      }
68 }
```

```
1  //
2  //  ViewController.swift
3  //  EmergencyGas
4  //
5  //  Created by Jadson Maciel on 17/03/25.
6  //
7  import UIKit
8  import CocoaMQTT
9
10 class HomeViewController: UIViewController {
11
12     private lazy var titleLabel: UILabel = {
13         let title = UILabel()
14         title.text = "Detector de gas: 01"
15         title.font = .systemFont(ofSize: 32, weight: .bold)
16         title.textColor = .white
17         title.translatesAutoresizingMaskIntoConstraints = false
18         return title
19     }()
20
21     lazy var statusView: UIView = {
22         let view = UIView()
23         view.backgroundColor = .green
24         view.translatesAutoresizingMaskIntoConstraints = false
25         return view
26     }()
```

```
27
28      lazy var statusLabel: UILabel = {
29          let label = UILabel()
30          label.text = "Ambiente Seguro"
31          label.font = .systemFont(ofSize: 22, weight: .light)
32          label.textColor = .white
33          label.textAlignment = .center
34          label.translatesAutoresizingMaskIntoConstraints = false
35          return label
36      }()
37
38      private lazy var spacer1: UIView = {
39          let spacer1 = UIView()
40          spacer1.translatesAutoresizingMaskIntoConstraints = false
41
42          return spacer1
43      }()
44
45      private lazy var buttonLabel: UIButton = {
46          let button = UIButton()
47          button.setTitle("SOS", for: .normal)
48          button.backgroundColor = .red
49          button.setTitleColor(.white, for: .normal)
50          button.layer.cornerRadius = 20
51          button.titleLabel?.font = .boldSystemFont(ofSize: 25)
52          button.translatesAutoresizingMaskIntoConstraints = false
53          button.addTarget(self, action: #selector(callEmergency),
               for: .touchUpInside)
54          return button
55      }()
56
57      private lazy var buttonInstruction: UIButton = {
58          let buttonInstruction = UIButton()
59          buttonInstruction.setTitle("Instrucoes", for: .normal)
60          buttonInstruction.backgroundColor = .buttonBackground
61          buttonInstruction.setTitleColor(.white, for: .normal)
62          buttonInstruction.layer.cornerRadius = 20
63          buttonInstruction.titleLabel?.font =
               .boldSystemFont(ofSize: 25)
64      buttonInstruction.translatesAutoresizingMaskIntoConstraints
           = false
65      buttonInstruction.addTarget(self, action:
```

```
        #selector(buttonInstructionPressed), for: .touchUpInside)
66         return buttonInstruction
67     }()
68
69     private lazy var buttonHistory: UIButton = {
70         let buttonHistory = UIButton()
71         buttonHistory.setTitle("Historico", for: .normal)
72         buttonHistory.backgroundColor = .blue
73         buttonHistory.setTitleColor(.white, for: .normal)
74         buttonHistory.layer.cornerRadius = 20
75         buttonHistory.titleLabel?.font = .boldSystemFont(ofSize:
            25)
76     buttonHistory.translatesAutoresizingMaskIntoConstraints =
        false
77     buttonHistory.addTarget(self, action:
        #selector(buttonHistoryPressed), for: .touchUpInside)
78         return buttonHistory
79     }()
80
81
82     private lazy var stackView: UIStackView = {
83         let stackView = UIStackView(arrangedSubviews:
            [statusView, statusLabel, spacer1 ,buttonLabel,
            buttonInstruction, buttonHistory])
84         stackView.axis = .vertical
85         stackView.spacing = 20
86         stackView.alignment = .center
87         stackView.translatesAutoresizingMaskIntoConstraints =
            false
88         return stackView
89     }()
90
91     var mqtt: CocoaMQTT?
92
93     override func viewDidLoad() {
94         super.viewDidLoad()
95         view.backgroundColor = .background
96
97         let clientID = "CocoaMQTT-" +
            String(ProcessInfo().processIdentifier)
98         mqtt = CocoaMQTT(clientID: clientID, host:
            "mqtt.eclipseprojects.io", port: 1883)
```

```
99
100         guard let mqtt = mqtt else {
101             print("Erro ao inicializar MQTT")
102             return
103         }
104 
105         mqtt.keepAlive = 60
106         mqtt.delegate = self
107         _ = mqtt.connect()
108 
109         addSubviews()
110         setupConstraints()
111 
112 
113 
114     }
115 
116     override func viewDidAppear(_ animated: Bool) {
117         super.viewDidAppear(animated)
118         statusView.layer.cornerRadius = statusView.frame.width /
               2
119         statusView.layer.masksToBounds = true
120     }
121 
122     func setupConstraints() {
123         NSLayoutConstraint.activate([
124             titleLabel.topAnchor.constraint(equalTo:
                   view.safeAreaLayoutGuide.topAnchor, constant: 100),
125             titleLabel.centerXAnchor.constraint(equalTo:
                   view.centerXAnchor),
126 
127             stackView.topAnchor.constraint(equalTo:
                   titleLabel.bottomAnchor, constant: 40),
128             stackView.centerXAnchor.constraint(equalTo:
                   view.centerXAnchor),
129 
130             statusView.widthAnchor.constraint(equalToConstant:
                   100),
131             statusView.heightAnchor.constraint(equalTo:
                   statusView.widthAnchor),
132 
133             buttonLabel.widthAnchor.constraint(equalToConstant:
```

```
                300),
134             buttonLabel.heightAnchor.constraint(equalToConstant:
                    60),
135
136             spacer1.heightAnchor.constraint(equalToConstant:
                    130),
137         buttonInstruction.widthAnchor.constraint(equalToConstant:
                300),
138         buttonInstruction.heightAnchor.constraint(equalToConstant:
                60),
139
140         buttonHistory.widthAnchor.constraint(equalToConstant:
                300),
141         buttonHistory.heightAnchor.constraint(equalToConstant:
                60)
142         ])
143     }
144
145     func addSubviews() {
146         view.addSubview(stackView)
147         view.addSubview(titleLabel)
148     }
149
150     @objc func callEmergency() {
151         if let phoneURL = URL(string: "tel://193"),
152            UIApplication.shared.canOpenURL(phoneURL) {
153             UIApplication.shared.open(phoneURL)
154         } else {
155             print("Nao e possivel fazer a ligacao.")
156         }
157     }
158
159     @objc private func buttonInstructionPressed () {
160     navigationController?.pushViewController(
161     InstructionViewController(), animated: true)
162     }
163
164     @objc private func buttonHistoryPressed () {
165     navigationController?.pushViewController(
166     HistoryViewController(), animated: true)
167     }
168 }
```

```
//
//  InstructionViewController.swift
//  EmergencyGas
//
//  Created by Jadson Maciel on 24/03/25.
//

import UIKit

class InstructionViewController: UIViewController {

    private let scrollView: UIScrollView = {
        let scrollView = UIScrollView()
        scrollView.translatesAutoresizingMaskIntoConstraints =
           false
        return scrollView
    }()

    private let contentView: UIView = {
        let contentView = UIView()
        contentView.translatesAutoresizingMaskIntoConstraints =
           false
        return contentView
    }()

    private lazy var titleLabel: UILabel = {
        let title = UILabel()
        title.text = "Em caso de vazamento de gas de cozinha,
           deve-se:"
        title.font = .systemFont(ofSize: 27, weight: .bold)
        title.textColor = .white
        title.translatesAutoresizingMaskIntoConstraints = false
        title.numberOfLines = 0
        return title
    }()

    private lazy var text1Label: UILabel = {
        let body = UILabel()
        body.text = "1. Fechar o registro de gas"
        body.font = .systemFont(ofSize: 20, weight: .light)
```

```
38          body.textColor = .white
39          body.translatesAutoresizingMaskIntoConstraints = false
40
41          body.numberOfLines = 0
42          return body
43      }()
44
45      private lazy var registerImage: UIImageView = {
46          let registerImage = UIImageView(image: .registergas)
47          registerImage.translatesAutoresizingMaskIntoConstraints
               = false
48          registerImage.contentMode = .scaleAspectFit
49          return registerImage
50      }()
51
52      private lazy var text2Label: UILabel = {
53          let body1 = UILabel()
54          body1.text = "2. Afastar pessoas e animais do local"
55          body1.font = .systemFont(ofSize: 20, weight: .light)
56          body1.textColor = .white
57          body1.translatesAutoresizingMaskIntoConstraints = false
58          body1.numberOfLines = 0
59          return body1
60      }()
61
62      private lazy var evacuationImage: UIImageView = {
63          let evacuationImage = UIImageView(image: .evacuation)
64      evacuationImage.translatesAutoresizingMaskIntoConstraints =
           false
65          evacuationImage.contentMode = .scaleAspectFit
66          return evacuationImage
67      }()
68
69      private lazy var text3Label: UILabel = {
70          let body2 = UILabel()
71          body2.text = "3. Ventilar o ambiente abrindo portas e
               janelas"
72          body2.font = .systemFont(ofSize: 20, weight: .light)
73          body2.textColor = .white
74          body2.translatesAutoresizingMaskIntoConstraints = false
75          body2.numberOfLines = 0
76          return body2
```

```
77      }()
78
79      private lazy var windowImage: UIImageView = {
80          let windowImage = UIImageView(image: .openingWindow)
81          windowImage.translatesAutoresizingMaskIntoConstraints =
               false
82          windowImage.contentMode = .scaleAspectFit
83          return windowImage
84      }()
85
86      private lazy var text4Label: UILabel = {
87          let body3 = UILabel()
88          body3.text = "4. Nao acender luzes ou usar aparelhos
               eletricos"
89          body3.font = .systemFont(ofSize: 20, weight: .light)
90          body3.textColor = .white
91          body3.translatesAutoresizingMaskIntoConstraints = false
92          body3.numberOfLines = 0
93          return body3
94      }()
95
96      private lazy var eletricImage: UIImageView = {
97          let eletricImage = UIImageView(image: .eletric)
98          eletricImage.translatesAutoresizingMaskIntoConstraints =
               false
99          eletricImage.contentMode = .scaleAspectFit
100         return eletricImage
101     }()
102
103     private lazy var text5Label: UILabel = {
104         let body4 = UILabel()
105         body4.text = "5. Chamar um profissional especializado"
106         body4.font = .systemFont(ofSize: 20, weight: .light)
107         body4.textColor = .white
108         body4.translatesAutoresizingMaskIntoConstraints = false
109         body4.numberOfLines = 0
110         return body4
111     }()
112
113     private lazy var professionalImage: UIImageView = {
114         let professionalImage = UIImageView(image:
               .callprofissional)
```

```
115         professionalImage.translatesAutoresizingMaskIntoConstraints
            = false
116         professionalImage.contentMode = .scaleAspectFit
117         return professionalImage
118     }()
119
120     private lazy var stackView: UIStackView = {
121         let stackView = UIStackView(arrangedSubviews:
               [text1Label, registerImage, text2Label,
               evacuationImage, text3Label, windowImage, text4Label,
               eletricImage, text5Label, professionalImage])
122         stackView.translatesAutoresizingMaskIntoConstraints =
               false
123
124         stackView.axis = .vertical
125         stackView.spacing = 16
126
127
128         return stackView
129     }()
130
131     override func viewDidLoad() {
132         super.viewDidLoad()
133         view.backgroundColor = .background
134         setupUI()
135         contentAddSubViews()
136         setupConstraints()
137
138         // Do any additional setup after loading the view.
139     }
140
141
142
143
144
145     func setupUI() {
146         view.addSubview(scrollView)
147         scrollView.addSubview(contentView)
148     }
149
150
151
```

```
152
153
154     func setupConstraints() {
155         let viewSF = view.safeAreaLayoutGuide
156
157         NSLayoutConstraint.activate([
158             scrollView.topAnchor.constraint(equalTo:
                   viewSF.topAnchor),
159             scrollView.trailingAnchor.constraint(equalTo:
                   view.trailingAnchor),
160             scrollView.leadingAnchor.constraint(equalTo:
                   view.leadingAnchor),
161             scrollView.bottomAnchor.constraint(equalTo:
                   viewSF.bottomAnchor)
162         ])
163
164         let hConst =
               contentView.heightAnchor.constraint(equalTo:
               scrollView.heightAnchor)
165         hConst.isActive = true
166         hConst.priority = UILayoutPriority(50)
167         NSLayoutConstraint.activate([
168             contentView.topAnchor.constraint(equalTo:
                   scrollView.topAnchor),
169             contentView.trailingAnchor.constraint(equalTo:
                   scrollView.trailingAnchor),
170             contentView.leadingAnchor.constraint(equalTo:
                   scrollView.leadingAnchor),
171             contentView.bottomAnchor.constraint(equalTo:
                   scrollView.bottomAnchor),
172             contentView.widthAnchor.constraint(equalTo:
                   scrollView.widthAnchor)
173
174         ])
175
176
177         NSLayoutConstraint.activate([
178             titleLabel.topAnchor.constraint(equalTo:
                   contentView.topAnchor),
179             titleLabel.centerXAnchor.constraint(equalTo:
                   contentView.centerXAnchor),
180             titleLabel.trailingAnchor.constraint(equalTo:
```

```
                contentView.trailingAnchor, constant: -12),
181             titleLabel.leadingAnchor.constraint(equalTo:
                contentView.leadingAnchor, constant: 12),
182
183             stackView.topAnchor.constraint(equalTo:
                titleLabel.bottomAnchor, constant: 10),
184             stackView.leadingAnchor.constraint(equalTo:
                contentView.leadingAnchor, constant: 15),
185             stackView.trailingAnchor.constraint(equalTo:
                contentView.trailingAnchor, constant: -15),
186             stackView.bottomAnchor.constraint(equalTo:
                contentView.bottomAnchor, constant: -42),
187
188         registerImage.heightAnchor.constraint(equalToConstant:
            200),
189
190         evacuationImage.heightAnchor.constraint(equalToConstant:
            200),
191
192         windowImage.heightAnchor.constraint(equalToConstant:
            200),
193
194         eletricImage.heightAnchor.constraint(equalToConstant:
            200),
195
196     professionalImage.heightAnchor.constraint(equalToConstant:
        200)
197
198
199         ])
200     }
201
202
203     func contentAddSubViews() {
204
205         contentView.addSubview(titleLabel)
206         contentView.addSubview(stackView)
207
208     }
209
210 }
```

APÊNDICE D – Programas do ESP32

Os programas a seguir serão definidos na ordem seguinte:

- **EmergencyGas.ino**
- **sensor.cpp**
- **sensor.h**
- **wifi_mqtt.cpp**
- **wifi_mqtt.h**

```
1  #include <WiFi.h>
2  #include <PubSubClient.h>
3  #include "wifi_mqtt.h"
4  #include "sensor.h"
5
6
7
8
9
10 const char* SSID = "iPhone";
11 const char* PASSWORD = "12345678*";
12
13 const char* BROKER_MQTT = "mqtt.eclipseprojects.io";
14 const char* ID_MQTT = "emergencygasesp";
15 #define TOPIC_PUBLISH "emergencygas"
16 const int BROKER_PORT = 1883;
17
18 WiFiClient wifiClient;
19 PubSubClient MQTT(wifiClient);
20
21 // Pinos
22 int pinsensor = 39;
23 int pinLEDRed = 25;
24 int pinLEDYellow = 33;
25 int pinLEDGreen = 32;
26 int pinbuzzer = 4;
27
```

```
28
29 bool messageSentG = false;
30 bool messageSentY = false;
31 bool messageSentR = false;
32
33
34 unsigned long tempoUltimaMudanca = 0;
35 const unsigned long atrasoDebounce = 2000;
36 int ultimoEstadoSensor = -1;
37 int estadoAtualSensor = -1;
38
39
40 void conectaWiFi();
41 void conectaMQTT();
42 void mantemConexoes();
43 int lerPercentualSensor(int pinoSensor);
44
45 void setup() {
46   Serial.begin(115200);
47
48   pinMode(pinsensor, INPUT);
49   pinMode(pinLEDRed, OUTPUT);
50   pinMode(pinLEDYellow, OUTPUT);
51   pinMode(pinLEDGreen, OUTPUT);
52   pinMode(pinbuzzer, OUTPUT);
53
54   conectaWiFi();
55   MQTT.setServer(BROKER_MQTT, BROKER_PORT);
56 }
57
58 void loop() {
59   mantemConexoes();
60   MQTT.loop();
61
62   int percentualSensor = lerPercentualSensor(pinsensor);
63
64   Serial.print("Percentual do sensor: ");
65   Serial.println(percentualSensor);
66
67   int novoEstado;
68   if (percentualSensor <= 13) {
69     novoEstado = 0;  // verde
```

```
70      } else if (percentualSensor <= 37) {
71        novoEstado = 1;   // amarelo
72      } else {
73        novoEstado = 2;   // vermelho
74      }
75
76      if (novoEstado != estadoAtualSensor) {
77        tempoUltimaMudanca = millis();
78        estadoAtualSensor = novoEstado;
79      }
80
81      if ((millis() - tempoUltimaMudanca) > atrasoDebounce) {
82
83        if (estadoAtualSensor != ultimoEstadoSensor) {
84
85          ultimoEstadoSensor = estadoAtualSensor;
86
87          switch (ultimoEstadoSensor) {
88            case 0:
89              digitalWrite(pinLEDGreen, HIGH);
90              digitalWrite(pinLEDYellow, LOW);
91              digitalWrite(pinLEDRed, LOW);
92              digitalWrite(pinbuzzer, LOW);
93
94              if (!messageSentG) {
95                MQTT.publish(TOPIC_PUBLISH, "green");
96                messageSentG = true;
97                messageSentY = false;
98                messageSentR = false;
99              }
100             break;
101
102           case 1:
103             digitalWrite(pinLEDGreen, LOW);
104             digitalWrite(pinLEDYellow, HIGH);
105             digitalWrite(pinLEDRed, LOW);
106             digitalWrite(pinbuzzer, LOW);
107
108             if (!messageSentY) {
109               MQTT.publish(TOPIC_PUBLISH, "yellow");
110               messageSentY = true;
111               messageSentG = false;
```

```
112             messageSentR = false;
113           }
114           break;
115 
116         case 2:
117           digitalWrite(pinLEDGreen, LOW);
118           digitalWrite(pinLEDYellow, LOW);
119           digitalWrite(pinLEDRed, HIGH);
120           digitalWrite(pinbuzzer, LOW);
121 
122           if (!messageSentR) {
123             MQTT.publish(TOPIC_PUBLISH, "red");
124             messageSentR = true;
125             messageSentG = false;
126             messageSentY = false;
127           }
128           break;
129       }
130     }
131   }
132 
133   delay(100);
134 }
```

```
1 
2 #include "sensor.h"
3 #include <Arduino.h>
4 
5 int lerPercentualSensor(int pinoSensor) {
6   int leitura = analogRead(pinoSensor);
7   int percentual = (int)((leitura / 4095.0) * 100);
8   return percentual;
9 }
```

```
1 #ifndef SENSOR_H
2 #define SENSOR_H
3 
4 int lerPercentualSensor(int pinoSensor);
5 
6 #endif
```

```
1 #include <WiFi.h>
```

```
#include <PubSubClient.h>
#include "wifi_mqtt.h"

extern const char* SSID;
extern const char* PASSWORD;
extern const char* BROKER_MQTT;
extern WiFiClient wifiClient;
extern PubSubClient MQTT;
extern const int BROKER_PORT;
extern const char* ID_MQTT;

void conectaWiFi() {
  if (WiFi.status() == WL_CONNECTED) {
    return;
  }

  Serial.print("Conectando na rede: ");
  Serial.println(SSID);

  WiFi.begin(SSID, PASSWORD);

  while (WiFi.status() != WL_CONNECTED) {
    delay(100);
    Serial.print(".");
  }

  Serial.println();
  Serial.print("Conectado com IP: ");
  Serial.println(WiFi.localIP());
}

void conectaMQTT() {
  while (!MQTT.connected()) {
    Serial.print("Conectando no broker MQTT: ");
    Serial.println(BROKER_MQTT);

    if (MQTT.connect(ID_MQTT)) {
      Serial.println("Conectado ao broker MQTT!");
    } else {
      Serial.println("Falha na conexao. Tentando novamente em 10 segundos.");
      delay(10000);
```

```
43      }
44    }
45  }
46
47  void mantemConexoes() {
48    if (!MQTT.connected()) {
49      conectaMQTT();
50    }
51    conectaWiFi();
52  }
```

```
1 #ifndef WIFI_MQTT_H
2 #define WIFI_MQTT_H
3
4 void conectaWiFi();
5 void conectaMQTT();
6 void mantemConexoes();
7
8 #endif
```