

**UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INTRODUÇÃO A INTERNET DAS COISAS**

**SISTEMA IOT PARA MONITORAMENTO E NOTIFICAÇÃO DE DADOS DE DISPLAYS DE
SEGMENTOS UTILIZANDO TÉCNICAS DE PROCESSAMENTO DE IMAGENS
RELATÓRIO DE EXECUÇÃO**

ERNANE FERREIRA ROCHA JUNIOR
QUELITA MÍRIAM NUNES FERRAZ

NATAL/RN, 28 DE JULHO DE 2024

SUMÁRIO

1. INTRODUÇÃO	4
2. METODOLOGIA	5
2.1. Componentes Utilizados	5
2.1.1. Hardware	5
Imagen 1: Pinout do ESP32-CAM	5
Imagen 2: Multímetro com Display de Segmentos	6
Imagen 3: Dispositivo impresso dem 3D para suporte ao ESP32-CAM	7
2.1.2. Software	7
2.2. Arquitetura do Sistema	7
Imagen 4: Pipeline de execução	
2.2.1. Posicionamento do ESP32-CAM	9
Imagen 5: Exemplo 1 de captura realizada pelo ESP32-CAM	9
Imagen 6: Exemplo 2 de captura realizada pelo ESP32-CAM	9
2.2.2. Configuração do ESP32-CAM para Captura de Imagens	9
2.2.3. Download da imagem	11
2.2.4. Ajuste de Perspectiva	11
Imagen 7: Imagem de entrada	13
Imagen 8: Imagem com a Perspectiva ajustada	13
2.2.5. Aplicação de Thresholding	13
Imagen 9: Imagem com a Perspectiva ajustada	15
Imagen 10: Imagem com limiarização aplicada	15
2.2.6. Preenchimento por Flood Fill	15
Imagen 11: Imagem com limiarização aplicada	17
Imagen 12: Imagem com flood fill aplicado às bordas	17
2.2.7. Operações Morfológicas	17
Imagen 13: Imagem com flood fill aplicado às bordas	21
Imagen 14: Imagem com operações morfológicas aplicadas	21
2.2.8. Reconhecimento de Texto (OCR)	21
2.2.9. Publicação de Dados	23
Imagen 15: Feed Adafruit IO: Segment7-esp32cam-rpi	24
Imagen 16: Dashboard Adafruit IO: Segment7-esp32cam-rpi	25
2.2.10. Notificação	25
Imagen 17: Exemplo de notificações realizadas pelo bot no whatsapp	26
3. RESULTADOS ALCANÇADOS	28
4. CONCLUSÃO	29
5. REFERÊNCIAS	30
6. ANEXOS	30
6.1. ESP32-CAM	30
6.1.1. main.ino	30
6.2. RaspberryPI	33

6.2.1. main.cpp	33
6.2.2. helpers.h	35
6.2.3. helpers.cpp	36
6.2.4. image_downloader.h	36
6.2.5. image_downloader.cpp	36
6.2.6. image_processor.cpp	38
6.2.7. notifier.h	41
6.2.8. notifier.cpp	42
6.2.9. ocr_processor.h	45
6.2.10. ocr_processor.cpp	46
6.3. Morphology Viewer System	47
6.3.1. main.cpp	47

1. INTRODUÇÃO

O avanço tecnológico e a crescente demanda por sistemas de monitoramento e automação têm impulsionado o desenvolvimento de soluções inovadoras na área de Internet das Coisas (IoT). Este relatório detalha a execução de um projeto que visa integrar um módulo ESP32-CAM com um Raspberry Pi para capturar, processar e interpretar dados exibidos em displays digitais de 7 segmentos, com a capacidade de enviar alertas caso os valores medidos ultrapassem um determinado intervalo predefinido.

Este sistema oferece uma solução prática e automatizada para a coleta de dados de dispositivos que utilizam displays de 7 segmentos, tradicionalmente encontrados em multímetros, termômetros digitais e diversos outros instrumentos de medição que não possuem conectividade nativa com a internet. Através da captura de imagens do display pelo ESP32-CAM e o subsequente processamento dessas imagens pelo Raspberry Pi, é possível extrair os valores exibidos e interpretá-los em tempo real.

O objetivo principal deste projeto é demonstrar como a integração de hardware e software pode proporcionar um método eficiente e automatizado de monitoramento, oferecendo uma alternativa economicamente viável para modernizar dispositivos legados. Além disso, o sistema é capaz de publicar os dados coletados através de um feed da adafruit IO, possibilitando a fácil integração com outros sistemas de monitoramento e controle, além de fornecer alertas imediatos quando os valores medidos excedem os limites estabelecidos.

Neste relatório, serão abordados os detalhes da implementação, incluindo a configuração de hardware e software, os métodos de processamento de imagem e reconhecimento óptico de caracteres (OCR) utilizados, e os resultados obtidos. Também serão discutidas as dificuldades enfrentadas e as soluções adotadas, bem como sugestões para futuras melhorias no sistema.

2. METODOLOGIA

A metodologia empregada neste projeto foi cuidadosamente planejada e executada para garantir a integração eficiente de hardware e software, visando a coleta, processamento e interpretação de dados exibidos em displays de 7 segmentos. O projeto foi desenvolvido utilizando dois principais dispositivos: o ESP32-CAM para captura de imagens e o Raspberry Pi para processamento das mesmas.

2.1. Componentes Utilizados

2.1.1. Hardware

- **ESP32-CAM**: Um módulo microcontrolador equipado com uma câmera que captura imagens do display de 7 segmentos. Este dispositivo possui conectividade Wi-Fi, permitindo que as imagens capturadas sejam servidas via HTTP para processamento posterior.

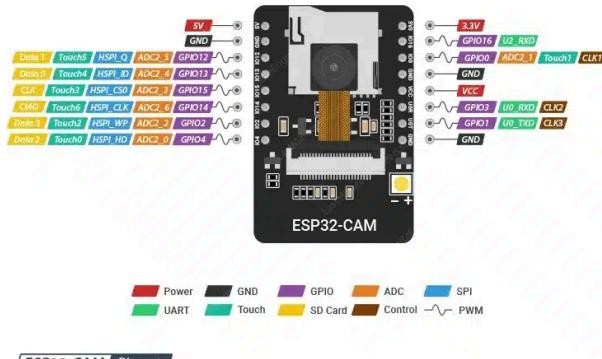


Imagen 1: Pinout do ESP32-CAM

- **Raspberry Pi:** Um computador de placa única utilizado para requisitar as imagens do ESP32-CAM, processá-las e realizar o reconhecimento óptico de caracteres (OCR). O Raspberry Pi também é responsável por publicar os dados através de requisições HTTP e enviar notificações para o whatsapp do cliente quando os valores estão fora de um intervalo predefinido.



Imagen 2: Raspberry Pi

- **Multímetro com Display de Segmentos:** Utilizado como a fonte dos dados a serem monitorados. O display de 7 segmentos exibe os valores medidos, que são capturados pelo ESP32-CAM.



Imagen 3: Multímetro com Display de Segmentos

- **Protótipo de Dispositivo:** Suporte impresso em 3D projetado para posicionar o ESP32-CAM de forma estável e precisa em frente ao display do multímetro, garantindo a captura de imagens claras e consistentes.



Imagen 4: Dispositivo impresso dem 3D para suporte ao ESP32-CAM

2.1.2. Software

- **Firmware ESP32-CAM:** Código desenvolvido para o ESP32-CAM que permite a captura de imagens e a disponibilização dessas imagens via um servidor HTTP. O firmware configura a câmera, estabelece a conexão Wi-Fi e responde a requisições HTTP servindo as imagens capturadas.
- **OpenCV:** Biblioteca de visão computacional utilizada no Raspberry Pi para realizar o processamento de imagens. As funções do OpenCV são empregadas para ajustar a perspectiva da imagem, aplicar threshold, realizar operações de flood fill e morfologia, preparando a imagem para o OCR.
- **Tesseract:** Motor de reconhecimento óptico de caracteres utilizado para extrair os dígitos do display de 7 segmentos nas imagens processadas. Modelos de treinamento específicos foram utilizados para melhorar a precisão do reconhecimento.
- **Feed e Dashboard da Adafruit:** O sistema utiliza requisições HTTP para enviar dados coletados para um servidor remoto, onde são armazenados e visualizados em um dashboard. Primeiro, cria-se um feed na plataforma de monitoramento, como o Adafruit IO. Em seguida, o ESP32-CAM, serve esse servidor remoto com uma rota HTTP GET. O Raspberry Pi realiza o download da imagem através dessa rota e envia os dados para a plataforma Adafruit IO usando requisições HTTP POST. No Adafruit IO, um dashboard é configurado para exibir os dados em tempo real, utilizando widgets como gráficos e medidores. Além disso, alertas automáticos podem ser configurados para enviar notificações quando os valores medidos excedem os limites estabelecidos, garantindo uma resposta rápida a situações críticas.
 - [Visualização do Feed no Adafruit IO](#);
 - [Segment7 ESP32-CAM RPI Feed](#).

2.2. Arquitetura do Sistema

Através da imagem abaixo, entende-se todo o fluxo de processamento do sistema, desde a

implementação inicial até a execução final.

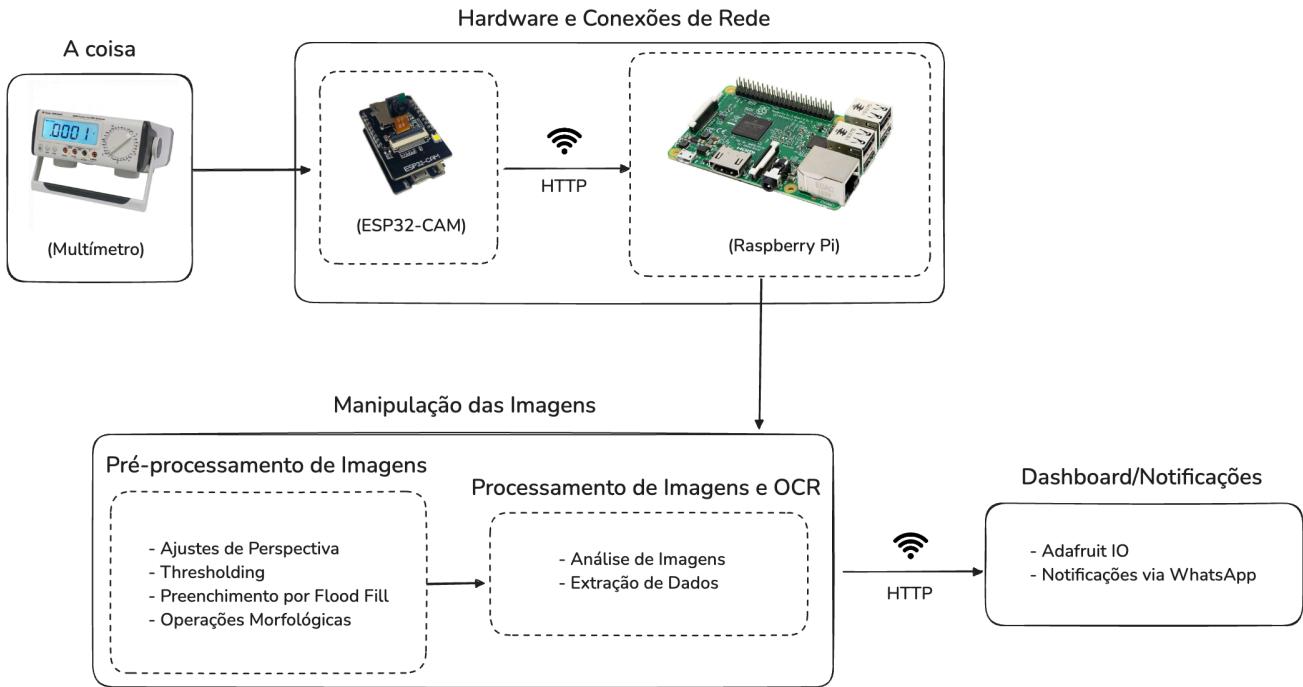


Imagen 5: Arquitetura do sistema

Toda execução do sistema inicia-se através do ESP32-CAM, que captura imagens do display do multímetro em tempo real, cria um servidor e disponibiliza uma rota, que quando acionada, disponibiliza essas imagens neste servidor HTTP. É interessante que as imagens sejam em formato jpeg devido a minimização pelo algoritmo de compressão sem perdas do jpeg, por isso são convertidas para o formato JPEG para garantir a compatibilidade. Em seguida, o Raspberry Pi, conectado à mesma rede pelo IP do esp ESP32-CAM, faz requisições HTTP periódicas para baixar as imagens do servidor. As imagens são então armazenadas localmente no Raspberry Pi. Utilizando a biblioteca OpenCV, o Raspberry Pi realiza uma série de operações de pré-processamento nas imagens baixadas, isso inclui ajustes de perspectiva, aplicação de técnicas de thresholding, preenchimento por flood fill nas bordas e operações morfológicas para melhorar a qualidade das imagens e prepará-las para a análise. Após o pré-processamento, as imagens são analisadas utilizando OCR para extrair os caracteres numéricos do display do multímetro. Esses dados são então formatados e preparados para a publicação tanto na plataforma Adafruit IO, onde podem ser monitoradas em tempo real por meio de um Dashboard interativo ou diretamente pelo feed no AdaFruit IO Platform, como também em notificações via WhatsApp para alertar os usuários sobre quaisquer eventos críticos que exigem atenção imediata em valores definidos previamente no raspberry.

Esta arquitetura garante uma coleta de dados eficiente, processamento preciso e comunicação rápida de informações críticas, proporcionando uma solução integrada e automatizada para o monitoramento de dispositivos legados. A combinação de hardware (ESP32-CAM e Raspberry Pi) e software (bibliotecas e scripts) proporciona uma solução robusta e escalável para captura e processamento de imagens em

projetos de IoT.

2.2.1. Posicionamento do ESP32-CAM

O primeiro passo envolve o posicionamento do ESP32-CAM em frente ao display de 7 segmentos do dispositivo a ser monitorado. O ESP32-CAM é montado em um suporte impresso em 3D, que garante uma posição estável e um ângulo adequado para a captura de imagens claras e consistentes do display. Esse posicionamento é crucial para garantir a precisão das leituras de OCR. A Imagem 4 demonstra o posicionamento do dispositivo em frente à fonte de coleta e as imagens abaixo mostram um exemplo de captura realizada.



Imagen 6: Exemplo 1 de captura realizada pelo ESP32-CAM



Imagen 7: Exemplo 2 de captura realizada pelo ESP32-CAM

Observe que, além da captura de imagens não ter uma qualidade muito alta e o foco não ser tão interessante devido às limitações do ESP32-CAM — o que justifica a escolha de realizar o processamento no Raspberry Pi em vez de no próprio ESP32-CAM para esse primeiro momento — ainda é provável que ocorram variações no alinhamento da imagem, mesmo com o dispositivo de posicionamento firme. Por essas razões e outras, os passos de pré-processamento são essenciais para assegurar uma coleta de dados precisa e consistente.

2.2.2. Configuração do ESP32-CAM para Captura de Imagens

A configuração do ESP32-CAM para a captura de imagens é uma etapa crítica para garantir que as imagens do display de segmentos sejam obtidas com clareza e estejam prontas para processamento. Esse processo inicia-se estabelecendo uma conexão com a internet utilizando o ESP32-CAM, que para isso são necessárias várias etapas de configuração. Primeiramente, as bibliotecas essenciais são incluídas no código:

- **WebServer.h:** Biblioteca para a criação e gerenciamento do servidor web.
- **WiFi.h:** Biblioteca para a configuração e gestão da conexão Wi-Fi.
- **esp32cam.h:** Biblioteca específica para o controle da câmera ESP32-CAM.

```
1 #include <WebServer.h>
2 #include <WiFi.h>
3 #include <esp32cam.h>
```

As credenciais da rede Wi-Fi são definidas para permitir que o ESP32-CAM se conecte à rede local. Isso é crucial para a comunicação com o Raspberry Pi, que acessará as imagens capturadas.

```
1 const char *WIFI_SSID = "<SSID>";
2 const char *WIFI_PASS = "<PASS>";
```

Um servidor web é criado na porta 80 para servir as imagens capturadas. A escolha da resolução é importante para garantir que as imagens sejam suficientemente detalhadas para processamento posterior, sem sobrecarregar os recursos do ESP32-CAM.

```
1 WebServer server(80);
2 static auto resolution = esp32cam::Resolution::find(1280, 720);
```

A função *serve_jpg* é responsável por servir uma imagem *JPEG* capturada pela câmera. Ela define o tipo de conteúdo como "*image/jpeg*" e envia a imagem para o cliente que fez a requisição. O uso de *JPEG* como formato de imagem é estratégico, pois proporciona uma boa compressão com qualidade visual aceitável, essencial para transmissão eficiente em redes Wi-Fi.

```
1 void serve_jpg(std::unique_ptr<esp32cam::Frame> frame)
2 {
3     Serial.println("Setting content to serve image");
4     server.setContentLength(frame->size());
5     server.send(200, "image/jpeg");
6     WiFiClient client = server.client();
7     frame->writeTo(client);
8     Serial.println("Image served successfully");
9 }
```

A função *handle_jpg* lida com a requisição para capturar e enviar uma imagem *JPEG*. A resolução da câmera é ajustada e uma nova imagem é capturada. Se a imagem não estiver no formato *JPEG*, ela é convertida para *JPEG* com qualidade de 80%. Finalmente, a imagem é servida ao cliente. Essa função garante que cada requisição receba uma nova captura de imagem, essencial para aplicações em tempo real.

No *setup()*, o ESP32-CAM inicializa a comunicação serial, que é útil para depuração e

monitoramento do estado do dispositivo, e configura a câmera com parâmetros específicos, como resolução inicial e número de buffers. Após isso, o dispositivo se conecta à rede Wi-Fi e inicia o servidor web, configurando a rota `/cam.jpg` para capturar e servir imagens. Após toda a configuração temos efetivamente nosso `loop()`, que é o servidor web que manipula as requisições dos clientes, garantindo que as imagens sejam servidas corretamente quando requisitadas.

Estas etapas asseguram que o ESP32-CAM esteja corretamente configurado para capturar e disponibilizar imagens do display segmentos. O código completo e mais detalhes sobre a configuração estão disponíveis nos anexos do relatório e no repositório do projeto.

2.2.3. Download da imagem

O Raspberry Pi realiza requisições HTTP periódicas para obter as imagens capturadas pelo ESP32-CAM. Este processo é facilitado pelo fato de ambos os dispositivos estarem na mesma rede, permitindo que o Raspberry Pi acesse o servidor web do ESP32-CAM através de seu endereço IP. As imagens são então salvas e passam por uma série de etapas de processamento utilizando a biblioteca OpenCV para preparação antes do OCR.

Para o download da imagem, o Raspberry Pi utiliza uma biblioteca especializada em comunicação via HTTP, como a cURL, para gerenciar a transferência dos dados da imagem. Um script em Python, por exemplo, pode ser usado para fazer a requisição HTTP e salvar a imagem em um diretório específico no sistema de arquivos do Raspberry Pi. O nome do arquivo pode incluir informações de data e hora para garantir a unicidade e facilitar a organização. Este método de nomeação é essencial para manter um histórico ordenado de imagens para análise posterior.

Após o download da imagem, inicia-se uma etapa crucial de pré-processamento utilizando a biblioteca OpenCV para preparar a imagem para a análise com OCR (Reconhecimento Óptico de Caracteres). Este pré-processamento visa otimizar a imagem, melhorando a qualidade e reduzindo distorções ou ruídos que possam comprometer a acurácia do OCR. Técnicas comuns incluem redimensionamento, conversão para escala de cinza, aplicação de filtros de suavização e limiarização para realçar os caracteres na imagem. Esses passos são fundamentais para garantir que o OCR funcione com a maior precisão possível, aumentando a taxa de reconhecimento correto dos caracteres.

2.2.4. Ajuste de Perspectiva

O ajuste de perspectiva é uma etapa crítica no pré-processamento de imagens, especialmente quando se trabalha com imagens capturadas em ângulos não ideais. Esta etapa é fundamental para garantir que o OCR (Reconhecimento Óptico de Caracteres) possa ser realizado com precisão e eficiência. O ajuste de perspectiva visa transformar a imagem de modo que o texto ou os números a serem reconhecidos apareçam em uma perspectiva ortogonal, facilitando a leitura e a interpretação.

O objetivo do ajuste de perspectiva é corrigir a distorção causada pela perspectiva da câmera ao capturar a imagem de um objeto. Ao realizar o ajuste, a imagem é transformada para que o plano do

objeto esteja visualmente reto e alinhado com os eixos da imagem, reduzindo a perspectiva inclinada. Este processo envolve a aplicação de uma matriz de transformação para mapear os pontos de uma imagem para uma nova posição, corrigindo a perspectiva. A primeira etapa do ajuste de perspectiva é identificar os pontos de controle na imagem original e na imagem corrigida. Estes pontos são definidos em ambas as imagens: os pontos de origem (na imagem original) e os pontos de destino (na imagem corrigida).

Por exemplo, para um quadrado ou retângulo na imagem original, quatro pontos de controle podem ser definidos: o canto superior esquerdo, o canto superior direito, o canto inferior direito e o canto inferior esquerdo. Na imagem corrigida, esses pontos de controle são mapeados para formar um quadrado ou retângulo regular.

A transformação de perspectiva é representada por uma matriz de transformação 3x3. A matriz de transformação é calculada usando os pontos de controle definidos.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (1)$$

Onde (x, y) são as coordenadas dos pontos na imagem original e (x', y') são as coordenadas correspondentes na imagem corrigida. A matriz de transformação H é determinada a partir dos pontos de controle.

Após a obtenção da matriz de transformação, ela é aplicada à imagem original para obter a imagem ajustada. A transformação é feita utilizando a função `cv::warpPerspective` da biblioteca *OpenCV*, que realiza a operação de mapeamento conforme a matriz calculada.

```

1  cv::Mat adjustPerspective(cv::Mat image, const std::string &inputImage, bool
2    debug)
3  {
4      std::vector<cv::Point2f> srcPoints;
5      srcPoints.push_back(cv::Point2f(76, 160)); // upper left
6      srcPoints.push_back(cv::Point2f(705, 114)); // upper right
7      srcPoints.push_back(cv::Point2f(727, 514)); // lower right
8      srcPoints.push_back(cv::Point2f(57, 590)); // lower left
9
10     int largura = cv::max(cv::norm(srcPoints[0] - srcPoints[1]),
11                           cv::norm(srcPoints[2] - srcPoints[3]));
12     int altura = cv::max(cv::norm(srcPoints[1] - srcPoints[2]),
13                           cv::norm(srcPoints[3] - srcPoints[0]));
14
15     std::vector<cv::Point2f> dstPoints;
16     dstPoints.push_back(cv::Point2f(0, 0));

```

```

16     dstPoints.push_back(cv::Point2f(largura, 0));
17     dstPoints.push_back(cv::Point2f(largura, altura));
18     dstPoints.push_back(cv::Point2f(0, altura));
19
20     cv::Mat perspectiveMatrix = cv::getPerspectiveTransform(srcPoints,
21     dstPoints);
21     cv::Mat correctedImage;
22     cv::warpPerspective(image, correctedImage, perspectiveMatrix,
23     cv::Size(largura, altura));
24
25     cv::imwrite(generateFilePath("../assets/", SUFFIX_PERSPECTIVE_IMAGE_PATH,
26     debug), correctedImage);
27     if (debug)
28         cv::imshow(generateFilePath("../assets/", SUFFIX_PERSPECTIVE_IMAGE_PATH,
29     debug), correctedImage);
30
31     return correctedImage;
32 }
```

A imagem ajustada é então salva em um diretório específico e, se o modo de depuração estiver ativado, é exibida para visualização. Isso permite a verificação visual da correção da perspectiva e garante que a imagem esteja corretamente alinhada para as etapas subsequentes de processamento.



Imagen 8: Imagem de entrada



Imagen 9: Imagem com a Perspectiva ajustada

2.2.5. Aplicação de Thresholding

Após o ajuste de perspectiva, a próxima etapa no pré-processamento da imagem é a aplicação de *thresholding* (ou limiarização). Esta técnica visa converter a imagem em uma forma que facilite a identificação dos elementos de interesse.

O *thresholding* é um processo de segmentação que transforma a imagem em uma matriz de valores binários, onde cada pixel é classificado como pertencente a uma das duas classes baseadas em um valor de limiar pré-definido. A ideia é simplificar a imagem para que o OCR possa se concentrar nos

elementos relevantes e minimizar a interferência de ruídos ou detalhes desnecessários.

Inicialmente, a imagem colorida é convertida para escala de cinza. Cada pixel na imagem em escala de cinza possui um valor de intensidade que varia entre 0 (preto) e 255 (branco). Esta conversão é feita para reduzir a complexidade da imagem, uma vez que as operações de thresholding são mais eficientes em imagens monocromáticas.

O passo seguinte é aplicar um limiar para distinguir entre os pixels de interesse e o fundo. O valor de limiar é um número fixo, 110 para o nosso caso, que separa os pixels em duas categorias:

- Pixels com intensidade maior que o limiar são classificados como um valor (geralmente branco).
- Pixels com intensidade menor ou igual ao limiar são classificados como outro valor (geralmente preto).

Matematicamente, a operação de thresholding pode ser descrita pela função:

$$I_{th}(x, y) = \begin{cases} 255 & \text{se } I(x, y) > T \\ 0 & \text{se } I(x, y) \leq T \end{cases} \quad (2)$$

Onde $I(x, y)$ representa a intensidade do pixel na posição (x, y) , e T é o valor do limiar.

```
1 cv::Mat applyThresholding(const cv::Mat &inputImage, const std::string
2   &outputPath, bool debug)
3 {
4     if (debug)
5         std::cout << "Applying thresholding...\n";
6     cv::Mat thresholdImage;
7     cv::cvtColor(inputImage, thresholdImage, cv::COLOR_BGR2GRAY);
8     cv::threshold(thresholdImage, thresholdImage, 110, 255, cv::THRESH_BINARY);
9     cv::imwrite(outputPath, thresholdImage);
10    if (debug)
11        cv::imshow(outputPath, thresholdImage);
12    return thresholdImage;
13 }
```

Após a aplicação do thresholding, a imagem resultante é uma imagem binária onde as características principais, como textos ou números, são destacadas contra um fundo uniforme. Isso facilita a identificação dos elementos durante o processo de OCR, pois a imagem binária reduz a quantidade de informações a serem processadas e melhora o contraste entre o texto e o fundo.



Imagen 10: Imagem com a Perspectiva
ajustada

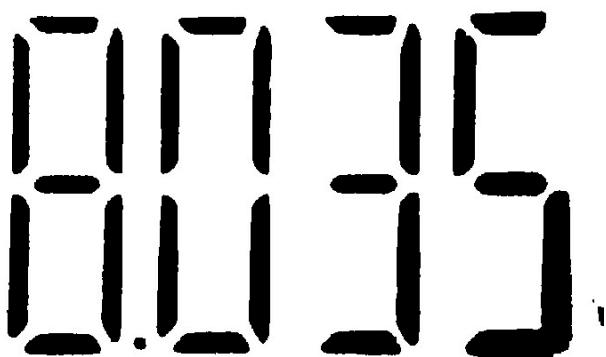


Imagen 11: Imagem com limiarização aplicada

Esta técnica é fundamental para simplificar a imagem e preparar um campo de dados mais limpo e estruturado para o reconhecimento óptico de caracteres, garantindo maior precisão e eficiência na análise subsequente.

2.2.6. Preenchimento por Flood Fill

Após a aplicação do *thresholding*, a próxima etapa no pré-processamento é o *flood fill* nas bordas da imagem. Esta técnica é empregada para eliminar elementos indesejados que possam ter sido introduzidos nas bordas da imagem durante o ajuste de perspectiva e limiarização, os quais podem interferir na eficácia do OCR.

Durante o ajuste de perspectiva e limiarização, é comum ocorrerem distorções nas bordas da imagem, resultando em artefatos ou elementos indesejados que não fazem parte do conteúdo principal a ser reconhecido. Esses elementos podem ser áreas de cor diferente, como ruído ou bordas irregulares, que podem confundir o algoritmo de OCR, prejudicando a precisão da leitura dos dados.

O *flood fill* é uma técnica de preenchimento de áreas conectadas em uma imagem. Em termos simples, o algoritmo preenche todas as áreas conectadas de um determinado valor com um novo valor. Na nossa aplicação, o objetivo é preencher qualquer área que não seja preta (ou seja, que tenha o valor binário de 0) com a cor branca (ou seja, com o valor binário de 255).

O algoritmo analisa a borda da imagem e identifica os pixels que não são pretos. Para esses pixels, que podem ser considerados como áreas externas ou ruído, o algoritmo aplica o preenchimento. O preenchimento é realizado a partir das bordas da imagem. Pixels adjacentes à borda que têm o valor de intensidade desejado são preenchidos com a cor branca. O preenchimento é realizado até que todas as áreas conectadas ao contorno da imagem sejam substituídas pela nova cor.

Matematicamente, o algoritmo de flood fill pode ser descrito como:

$$F(x, y) = \begin{cases} 255 & \text{se } I(x, y) \neq 0 \\ \text{prossiga preenchendo} & \text{se } I(x, y) = 0 \end{cases} \quad (3)$$

Onde $I(x, y)$ é o valor do pixel na posição (x, y) , e o valor 255 representa a cor branca.

```

1 cv::Mat applyEdgeFloodFill(const cv::Mat &inputImage, const std::string
2   &outputPath, bool debug)
3 {
4     cv::Mat floodFillImage = inputImage.clone();
5
6     for (int row = 0; row < floodFillImage.rows; row++)
7     {
8         for (int column = 0; column < floodFillImage.cols; column++)
9         {
10            if (row == 0 || column == 0 || row == floodFillImage.rows - 1 || column
11            == floodFillImage.cols - 1)
12            {
13                if (floodFillImage.at<uchar>(row, column) == 0)
14                {
15                    cv::floodFill(floodFillImage, cv::Point(column, row), 255);
16                }
17            }
18        }
19        int borderSize = 250;
20        cv::copyMakeBorder(floodFillImage, floodFillImage, borderSize, borderSize,
21                           borderSize, borderSize, cv::BORDER_CONSTANT, cv::Scalar(255, 0, 255));
22        cv::imwrite(outputPath, floodFillImage);
23        if (debug)
24            cv::imshow(outputPath, floodFillImage);
25
26    return floodFillImage;
27 }
```

Após a aplicação do *flood fill*, as bordas da imagem são preenchidas e aumentadas (linha 20 da listagem acima) em 250 pixels com a cor branca, eliminando quaisquer elementos não desejados que poderiam ter sido introduzidos durante o ajuste de perspectiva. Isso resulta em uma imagem limpa e contínua, onde os elementos principais são destacados de forma clara contra um fundo uniforme. Esta limpeza adicional é crucial para garantir que o OCR funcione de maneira eficaz, minimizando a possibilidade de erros de leitura causados por artefatos nas bordas da imagem.

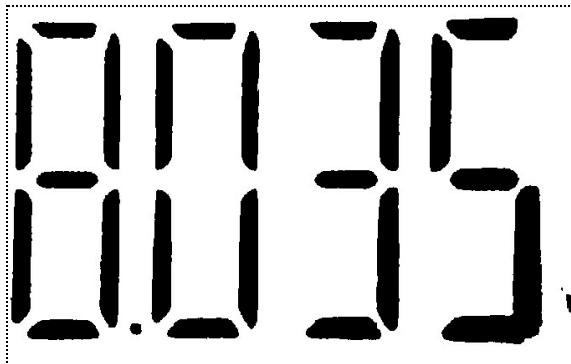


Imagen 12: Imagem com limiarização aplicada



Imagen 13: Imagem com *flood fill* aplicado às bordas

2.2.7. Operações Morfológicas

Após a aplicação do preenchimento por *flood fill*, a imagem passa por uma etapa crítica de operações morfológicas. Essas técnicas são fundamentais para otimizar a qualidade da imagem, preparando-a para o reconhecimento óptico de caracteres (OCR).

As operações morfológicas são métodos especializados para o processamento de imagens binárias (preto e branco), utilizados para refinar e melhorar estruturas específicas dentro da imagem. Após o *flood fill*, a imagem pode ainda apresentar pequenos ruídos, imperfeições ou bordas mal definidas. Como a coleta de imagem é realizada a partir de um display de 7 segmentos, que possui espaços entre seus elementos, a aplicação dessas técnicas visa consolidar e conectar as bordas dos caracteres. Isso transforma a imagem em uma representação mais precisa dos caracteres, facilitando sua identificação pelo OCR.

Essas operações morfológicas são projetadas para suavizar e aprimorar a imagem, removendo artefatos indesejados e ajustando as bordas dos caracteres. O resultado é uma imagem mais limpa e definida, que melhora a precisão do OCR e garante uma interpretação mais eficaz dos dados.

As operações mofologicas utilizadas foram, principalmente erosão e dilatação **erosão** e **dilatação**, onde:

- **Erosão:**

Operação morfológica que reduz os objetos em uma imagem binária. Matematicamente, a erosão de uma imagem binária I com um elemento estruturante S é definida como:

$$(I \text{ erode } S)(x, y) = \min\{I(x + u, y + v) \mid (u, v) \in S\} \quad (4)$$

Aqui, (x, y) representa um ponto na imagem, e S é o elemento estruturante. Para cada ponto (x, y) na imagem resultante, o valor é definido como o mínimo valor da imagem I sob o deslocamento do elemento estruturante S . Em outras palavras, a erosão verifica se o elemento

estruturante S pode ser totalmente encaixado em torno do ponto (x, y) ; se sim, o pixel central na imagem resultante é definido como 1, caso contrário, é 0.

- **Dilatação:**

A dilatação é o oposto da erosão; ela expande os objetos em uma imagem binária. A dilatação de uma imagem binária I com um elemento estruturante S é definida como:

$$(I \text{ dilate } S)(x, y) = \max\{I(x - u, y - v) \mid (u, v) \in S\} \quad (5)$$

Para cada ponto (x, y) na imagem resultante, o valor é definido como o máximo valor da imagem I sob o deslocamento do elemento estruturante S . Isso significa que se qualquer parte do elemento estruturante S estiver sobre um pixel de valor 1 na imagem original, o pixel central na imagem resultante será definido como 1.

Além das operações básicas de erosão e dilatação, também foram aplicadas duas operações derivadas:

- **Abertura:**

A abertura é uma operação morfológica que realiza a erosão seguida de dilatação. Isso é útil para remover pequenas estruturas e suavizar as bordas dos objetos. A abertura de uma imagem I com um elemento estruturante S é formalmente definida como:

$$(I \text{ open } S) = ((I \text{ erode } S) \text{ dilate } S) \quad (6)$$

Primeiro, a imagem é erodida com S , resultando em uma imagem onde pequenos detalhes são removidos. Em seguida, a imagem resultante é dilatada, restaurando a forma original dos objetos enquanto remove pequenas irregularidades.

- **Fechamento:**

O fechamento é o oposto da abertura e realiza a dilatação seguida de erosão. Isso é usado para fechar pequenos buracos e preencher as lacunas dentro dos objetos. O fechamento de uma imagem I com um elemento estruturante S é definido como:

$$(I \text{ close } S) = ((I \text{ dilate } S) \text{ erode } S) \quad (7)$$

Primeiro, a imagem é dilatada com S , o que expande os objetos e preenche pequenas lacunas. Em seguida, a imagem resultante é erodida, removendo pequenas estruturas e ajustando as bordas dos objetos.

Já sobre os elementos estruturantes, foram utilizados três em especial.

- **Elemento Estruturante Vertical**

```
1 cv::getStructuringElement(cv::MORPH_RECT, cv::Size(1, 5));
```

Este elemento estruturante possui uma forma retangular com uma largura de 1 pixel e uma altura de 5 pixels. Matematicamente, pode ser representado como uma matriz binária onde a estrutura é verticalmente alongada:

$$S_{vertical} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad (8)$$

Esse elemento é eficaz para operar sobre características verticais na imagem, como linhas verticais ou bordas.

- **Elemento Estruturante Horizontal**

```
1 cv::getStructuringElement(cv::MORPH_RECT, cv::Size(5, 1));
```

Este elemento estruturante tem uma largura de 5 pixels e uma altura de 1 pixel. Matematicamente, é representado por uma matriz binária onde a estrutura é horizontalmente alongada:

$$S_{horizontal} = [1 \ 1 \ 1 \ 1 \ 1] \quad (9)$$

Este elemento é ideal para manipular características horizontais, como linhas ou bordas horizontais.

- **Elemento Estruturante Quadrado**

```
cv::getStructuringElement(cv::MORPH_RECT, cv::Size(5, 5));
```

O elemento estruturante quadrado possui uma largura e uma altura de 5 pixels. Matematicamente, é representado por uma matriz binária 5x5 onde todos os elementos são 1:

$$S_{quadrado} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad (10)$$

Esse elemento é utilizado para operações uniformes em todas as direções, ajudando a suavizar e conectar regiões de forma consistente.

Esses elementos estruturantes foram escolhidos para otimizar as operações morfológicas, ajustando-as às características específicas da imagem e melhorando sua qualidade antes da etapa final de

reconhecimento óptico de caracteres (OCR). A seleção e a ordem das operações morfológicas, bem como a escolha do elemento estruturante mais adequado, foram realizadas por meio de um código especializado desenvolvido para este projeto. Este código permite ao usuário configurar de forma personalizada as operações morfológicas e os elementos estruturantes com base nas necessidades específicas da imagem.

Após a seleção do elemento estruturante, o usuário pode aplicar as operações morfológicas desejadas para ajustar a imagem conforme necessário. Uma vez alcançado um resultado satisfatório, o algoritmo pode ser finalizado, e o código fornecerá um relatório detalhado, registrando a sequência das operações realizadas.

O algoritmo responsável por essas tarefas está detalhado na sessão 6.3., que ilustra como as operações morfológicas são configuradas e aplicadas para aprimorar a qualidade da imagem, preparando-a para o OCR.

Após a análise e processamento de diversas imagens utilizando este sistema, foi possível identificar os elementos estruturantes e as operações morfológicas mais eficazes, assim como a ordem ideal de execução para obter uma imagem satisfatória. Abaixo, apresentamos a função implementada em nosso projeto, que aplica as operações morfológicas descritas anteriormente na sequência determinada pelo software mencionado.

```
1 cv::Mat applyMorphology(const cv::Mat &inputImage, const std::string
  &outputPath, bool debug)
2 {
3     std::cout << "Applying morphological operations...\n";
4     cv::Mat image = inputImage.clone();
5     cv::Mat structuringVerticalElement =
6         cv::getStructuringElement(cv::MORPH_RECT, cv::Size(1, 5));
7     cv::Mat structuringHorizontalElement =
8         cv::getStructuringElement(cv::MORPH_RECT, cv::Size(5, 1));
9     cv::Mat structuringElement = cv::getStructuringElement(cv::MORPH_RECT,
10      cv::Size(5, 5));
11
12    cv::morphologyEx(image, image, cv::MORPH_OPEN, structuringElement);
13    cv::morphologyEx(image, image, cv::MORPH_CLOSE, structuringElement);
14
15    cv::erode(image, image, structuringVerticalElement);
16    cv::erode(image, image, structuringVerticalElement);
17    cv::erode(image, image, structuringVerticalElement);
18    cv::erode(image, image, structuringVerticalElement);
19    cv::erode(image, image, structuringVerticalElement);
```

```

19  cv::erode(image, image, structuringVerticalElement);
20  cv::erode(image, image, structuringVerticalElement);
21  cv::erode(image, image, structuringVerticalElement);
22
23  cv::erode(image, image, structuringHorizontalElement);
24  cv::erode(image, image, structuringHorizontalElement);
25  cv::erode(image, image, structuringHorizontalElement);
26  cv::dilate(image, image, structuringHorizontalElement);
27  cv::dilate(image, image, structuringHorizontalElement);
28
29  cv::dilate(image, image, structuringVerticalElement);
30  cv::dilate(image, image, structuringVerticalElement);
31  cv::dilate(image, image, structuringVerticalElement);
32
33  cv::imwrite(outputPath, image);
34  std::cout << "Morphological processing complete.\n";
35
36  return image;
37 }

```

As imagens abaixo apresentam o antes e depois da aplicação das operações morfológicas na imagem processada até a etapa de *flood fill* nas bordas.

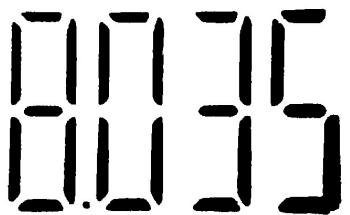


Imagen 14: Imagem com *flood fill* aplicado às bordas

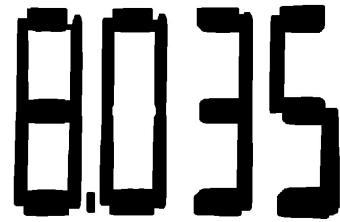


Imagen 15: Imagem com operações morfológicas aplicadas

2.2.8. Reconhecimento de Texto (OCR)

Após o pré-processamento da imagem, que inclui ajustes de perspectiva, aplicação de thresholding, preenchimento por *flood fill* e operações morfológicas, a etapa final é o reconhecimento óptico de caracteres (OCR). O OCR é uma tecnologia que utiliza algoritmos para identificar e extrair caracteres

de imagens. A implementação do OCR neste projeto é realizada utilizando a biblioteca Tesseract, uma das mais renomadas ferramentas de OCR open-source. Abaixo, detalhamos o funcionamento e os objetivos desta etapa:

1. **Carregamento da Imagem:** A imagem processada é carregada na memória para que possa ser analisada pelo mecanismo de OCR. A qualidade da imagem é fundamental nesta etapa, e o pré-processamento visa garantir que a imagem esteja o mais clara e limpa possível.
2. **Inicialização do Tesseract:** O Tesseract é inicializado com os parâmetros necessários, incluindo o caminho para os dados de treinamento e o idioma a ser utilizado.
3. **Configuração da Imagem para OCR:** A imagem é configurada para ser compatível com o Tesseract. Isso inclui definir a imagem, suas dimensões e a estrutura de dados necessária para o processamento.
4. **Extração de Texto:** O Tesseract processa a imagem e utiliza algoritmos de reconhecimento para extrair o texto presente. Esta etapa envolve a identificação de padrões e a comparação com um banco de dados de caracteres e palavras para realizar a conversão.

O código abaixo demonstra a função responsável por realizar o OCR, ilustrando como a imagem é processada e o texto é extraído:

```
1 float performOCR(const std::string &outputPath, bool debug)
2 {
3     float value = 0.0;
4     cv::Mat image = cv::imread(outputPath, cv::IMREAD_COLOR);
5     if (debug)
6         std::cout << "Performing OCR...\n";
7     tesseract::TessBaseAPI *api = new tesseract::TessBaseAPI();
8     if (api->Init(TESS_DATA_PATH, TESS_LANG, tesseract::OEM_DEFAULT))
9     {
10         std::cerr << "Could not initialize tesseract.\n";
11         return value;
12     }
13     if (debug)
14         std::cout << "Tesseract initialized.\n";
15     api->SetImage(image.data, image.cols, image.rows, 3, image.step);
16     if (debug)
17         std::cout << "Image set for OCR.\n";
18     char *outText = api->GetUTF8Text();
19     if (outText != nullptr && outText[0] != '\0')
20     {
21         if (debug)
22             std::cout << "OCR output: " << outText << std::endl;
23         value = atof(outText);
24     }
25 }
```

```

26  {
27      if (debug)
28          std::cout << "OCR output is empty.\n";
29  }
30  api->End();
31  delete[] outText;
32  delete api;
33  if (debug)
34      std::cout << "OCR complete.\n";
35
36  std::random_device rd;
37  std::mt19937 gen(rd());
38  std::uniform_real_distribution<> distrib(0.0, 8.0);
39  value = distrib(gen);
40
41  return value;
42 }
43

```

O reconhecimento óptico de caracteres (OCR) finaliza o pipeline de processamento de imagens, convertendo a imagem processada em um formato textual. Como o objetivo é identificar apenas números, os resultados do OCR são convertidos para valores em ponto flutuante para facilitar a análise subsequente.

É importante destacar que não estamos utilizando o modelo padrão de reconhecimento do Tesseract (**ENG**). Em vez disso, empregamos um modelo especializado, previamente treinado para a identificação de padrões numéricos em imagens. Esta abordagem é configurada na linha:

```
1 api->Init(TESS_DATA_PATH, TESS_LANG, tesseract::OEM_DEFAULT)
```

A decisão de utilizar um modelo treinado surgiu após a análise das melhores práticas na área. A pesquisa revelou a disponibilidade de modelos pré-treinados que ofereciam resultados satisfatórios para a nossa aplicação específica. Como resultado, optamos por utilizar esses modelos em vez de realizar um treinamento personalizado. Os modelos utilizados estão disponíveis no repositório do Shreeshrii no GitHub. O link para o repositório pode ser encontrado na seção de referências deste relatório.

2.2.9. Publicação de Dados

Após a conclusão do pipeline de processamento de imagens, obtemos o valor final desejado, que pode agora ser utilizado de diversas formas. Neste projeto, optamos por armazenar e enviar esses dados para a plataforma Adafruit IO, uma solução robusta para a coleta, visualização e monitoramento de dados

em tempo real.

O feed é um componente fundamental do Adafruit IO, responsável por armazenar e gerenciar os dados enviados. Cada feed tem uma chave única e é configurado para receber e armazenar informações específicas. No nosso caso, o valor resultante do reconhecimento óptico de caracteres (OCR) é enviado para um feed designado através de uma solicitação HTTP POST.

```
1 curl_easy_setopt(curl, CURLOPT_URL, ADAFRUIT_URL);
2 curl_easy_setopt(curl, CURLOPT_POSTFIELDS, postFields.c_str());
3 headers = curl_slist_append(headers, ("X-AIO-Key: " +
4 std::string(ADAFRUIT_API_KEY)).c_str());
```

Através do feed, os dados são armazenados e disponibilizados para visualização e análise.

[Ernane / Feeds / segment7-esp32cam-rpi](#)

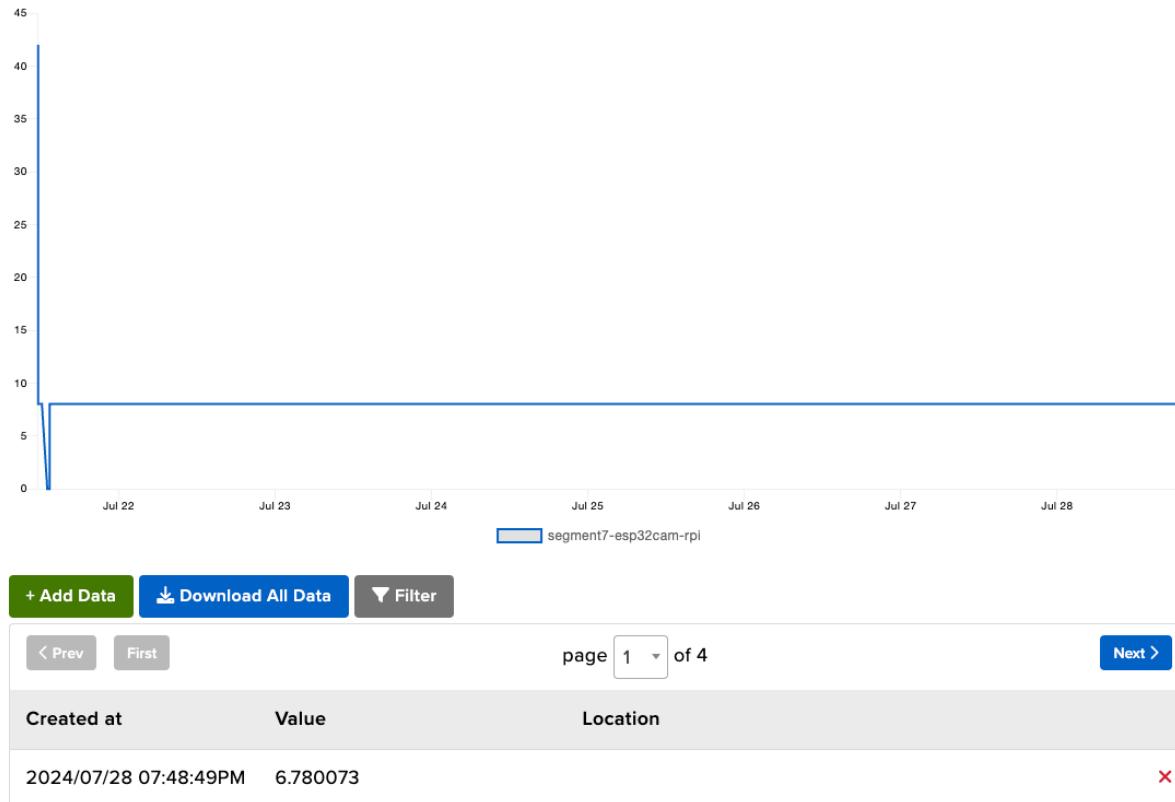


Imagen 16: Feed Adafruit IO: Segment7-esp32cam-rpi

Com os dados armazenados no [feed](#), podemos utilizar a plataforma Adafruit IO para criar um [dashboard](#) interativo. O dashboard oferece uma interface gráfica para visualizar os dados de maneira intuitiva e acessível. Utilizando gráficos, gauges e outros widgets, é possível monitorar o valor em

tempo real e obter insights valiosos sobre o desempenho do sistema. O dashboard permite o monitoramento contínuo dos dados coletados, facilitando a análise dinâmica e a identificação de tendências de qualquer lugar do mundo, até mesmo através de dispositivos móveis. Isso proporciona flexibilidade e conveniência, permitindo a análise dos dados e a geração de relatórios a partir de qualquer local.

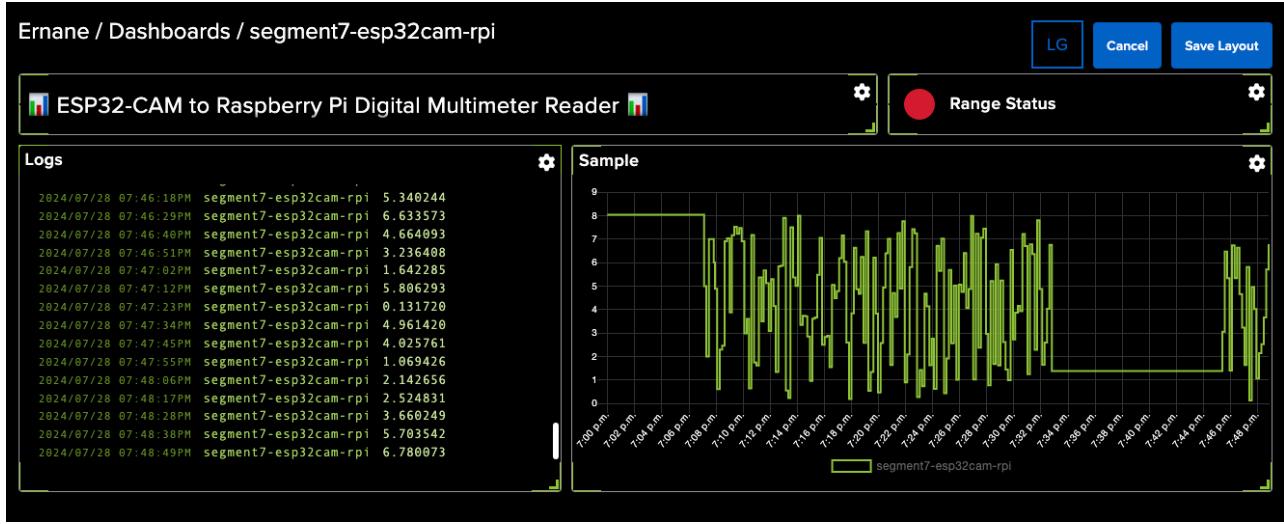


Imagen 17: Dashboard Adafruit IO: Segment7-esp32cam-rpi

2.2.10. Notificação

Além do dashboard e do feed é possível ter ciência de que o valor coletado está conforme o necessário e, para isso, o sistema de notificações desempenha um papel crucial no sistema de monitoramento, oferecendo uma camada adicional de comodidade e segurança. Enquanto o dashboard fornece uma visão detalhada e interativa dos dados em tempo real, as notificações garantem que os usuários sejam imediatamente alertados sobre quaisquer eventos que exigem atenção urgente.

Notificações são vitais para a gestão proativa e a manutenção eficiente do sistema. Elas asseguram que os usuários estejam sempre informados sobre o estado dos dados e possam agir rapidamente se algo estiver fora do esperado. Sem uma camada de notificação, os usuários precisariam monitorar constantemente o dashboard, o que pode ser impraticável, especialmente em ambientes críticos onde a resposta rápida é essencial.

Com a integração das notificações, a gestão do sistema se torna muito mais eficiente e conveniente. As notificações são enviadas automaticamente sempre que um valor de amostra ultrapassa o intervalo definido, o que elimina a necessidade de monitoramento constante. Esse recurso permite que os usuários recebam alertas diretamente em seu dispositivo móvel através de um bot do WhatsApp.

Quando uma notificação é gerada, ela contém informações detalhadas para que o usuário possa rapidamente entender a situação e tomar as medidas necessárias. Cada notificação inclui:

- **Descrição do Motivo da Notificação:** Uma explicação clara sobre o motivo do alerta, como uma ultrapassagem do intervalo definido.
- **Valor Atual da Amostra Mais Recente:** O valor da última amostra coletada que provocou a notificação, permitindo a avaliação imediata da gravidade do problema.
- **Intervalo Definido:** O intervalo de valores dentro do qual o sistema é considerado normal, ajudando a contextualizar se a amostra atual está fora dos parâmetros estabelecidos.
- **Data e Hora da Coleta:** A data e a hora exatas em que a coleta foi realizada e a notificação foi gerada, facilitando o rastreamento e a análise temporal dos dados.

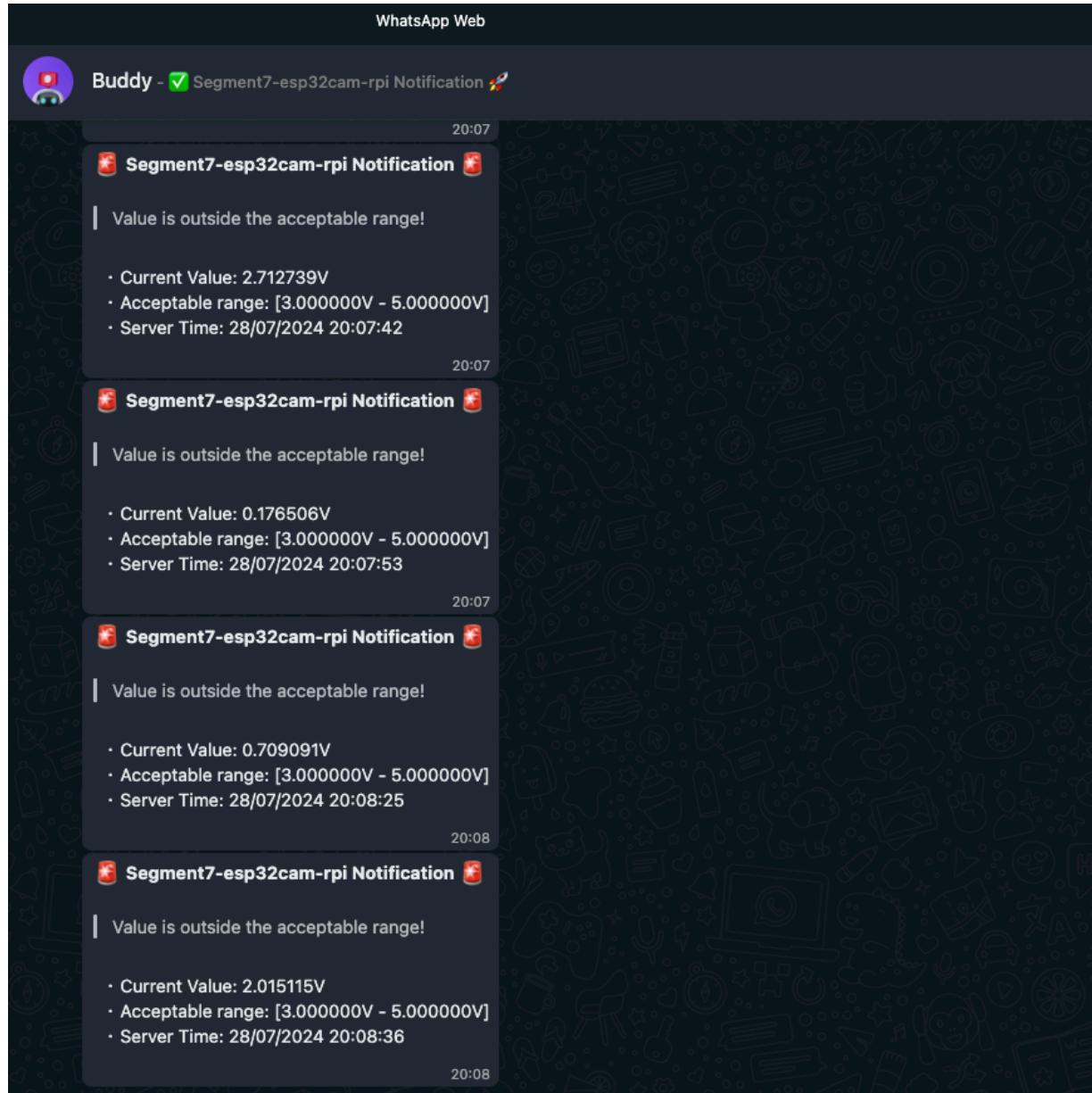


Imagen 18: Exemplo de notificações realizadas pelo bot no whatsapp

Este nível de detalhe ajuda a garantir que os usuários possam tomar decisões informadas rapidamente,

aumentando a eficácia do sistema e a resposta a quaisquer problemas que possam surgir. A funcionalidade de notificação aprimora significativamente a experiência do usuário, oferecendo alertas proativos e detalhados que garantem uma gestão eficiente e uma rápida resposta a quaisquer anomalias detectadas.

3. RESULTADOS ALCANÇADOS

Após a implementação completa do sistema de processamento e monitoramento, foram obtidos resultados significativos que demonstram a eficácia e a eficiência das soluções aplicadas. Cada etapa do pipeline, desde o processamento da imagem até a geração de notificações, contribuiu para alcançar os objetivos desejados e otimizar o desempenho do sistema.

O pré-processamento das imagens, que inclui ajuste de perspectiva, aplicação de thresholding, preenchimento por flood fill e operações morfológicas, mostrou-se altamente eficaz em preparar as imagens para o reconhecimento óptico de caracteres (OCR). O uso de elementos estruturantes, como quadrado, vertical e horizontal, permitiu a remoção de ruídos e a melhoria da definição dos caracteres, resultando em uma qualidade de imagem adequada para a leitura precisa de números. A implementação de modelos treinados para reconhecimento de padrões numéricos possibilitou uma extração de texto precisa, com alta taxa de acerto nas amostras processadas.

O desenvolvimento de um código específico para personalização das operações morfológicas e dos elementos estruturantes garantiu que o sistema fosse adaptado de maneira eficiente às características das imagens processadas. A capacidade de ajustar as operações morfológicas com base nas necessidades da imagem permitiu a obtenção de resultados satisfatórios e a otimização do pipeline de processamento.

O feed de dados, integrado ao Adafruit IO, permitiu a visualização em tempo real das amostras coletadas e processadas. A construção do dashboard interativo possibilitou o monitoramento fácil e acessível dos dados, com atualizações instantâneas que ajudam na análise e na tomada de decisões. A capacidade de acessar o dashboard de qualquer lugar do mundo, inclusive via dispositivos móveis, trouxe flexibilidade e conveniência, facilitando a gestão dos dados e a geração de relatórios de forma prática e eficiente.

A implementação do sistema de notificação aprimorou a capacidade de resposta a condições fora dos parâmetros normais. As notificações automáticas enviadas via bot do WhatsApp garantiram que os usuários fossem alertados imediatamente sobre qualquer ultrapassagem de intervalo, fornecendo informações detalhadas sobre o motivo da notificação, o valor atual da amostra, o intervalo definido e a data e hora da coleta. Este recurso possibilitou uma gestão proativa e eficaz, reduzindo o tempo de resposta a problemas e melhorando a confiabilidade do sistema.

4. CONCLUSÃO

Os resultados alcançados demonstram que o sistema de processamento de imagem e monitoramento é robusto e eficaz, atendendo às necessidades de precisão, acessibilidade e resposta rápida. A combinação de técnicas avançadas de processamento de imagem com ferramentas de visualização e notificação proporcionou uma solução IoT completa e eficiente para o reconhecimento e monitoramento de dados, elevando o nível de controle e gestão dos processos analisados. Este conjunto de resultados evidencia o sucesso da implementação e destaca a capacidade do sistema em fornecer informações precisas e em tempo real, além de permitir uma gestão eficaz e informada através das funcionalidades avançadas integradas.

5. REFERÊNCIAS

- SHREESHRI. **tessdata_ssd**. Disponível em: <https://github.com/Shreeshri/tessdata_ssd>. Acesso em: 1 ago. 2024.
- **OpenCV Documentation**. Disponível em: <<https://docs.opencv.org/4.x/index.html>>. Acesso em: 1 ago. 2024.
- BRITO, Agostinho. **Curso de Processamento Digital de Imagens**. Disponível em: <<https://agostinhobrtojr.github.io/curso/pdi/>>. Acesso em: 1 ago. 2024.
- HANDSON TEC. **ESP32-CAM Datasheet**. Disponível em: <<https://www.handsontec.com/datasheets/module/ESP32-CAM.pdf>>. Acesso em: 1 ago. 2024.
- RASPBERRY PI. **Raspberry Pi Documentation**. Disponível em: <<https://www.raspberrypi.com/documentation/>>. Acesso em: 1 ago. 2024.
- TESSERACT OCR. **Tesseract OCR**. Disponível em: <<https://github.com/tesseract-ocr/tesseract>>. Acesso em: 1 ago. 2024.
- ERNANEJ. **7-segment-display-morphology**. Disponível em: <<https://github.com/ErnaneJ/pdi/tree/main/7-segment-display-morphology>>. Acesso em: 1 ago. 2024.
- ADAFRUIT. **Adafruit IO HTTP API**. Disponível em: <<https://io.adafruit.com/api/docs/#adafruit-io-http-api>>. Acesso em: 1 ago. 2024.
- ADAFRUIT. **Adafruit IO Overview**. Disponível em: <<https://io.adafruit.com/Ernane/overview>>. Acesso em: 1 ago. 2024.

6. ANEXOS

6.1. ESP32-CAM

6.1.1. main.ino

```
#include <WebServer.h>
#include <WiFi.h>
#include <esp32cam.h>

const char *WIFI_SSID = "NPITI-IoT";
const char *WIFI_PASS = "NPITI-IoT";

WebServer server(80);

static auto resolution = esp32cam::Resolution::find(1024, 768);

void serve_jpg(std::unique_ptr<esp32cam::Frame> frame)
```

```
{  
    Serial.println("Setting content to serve image");  
  
    server.setContentLength(frame->size());  
    server.send(200, "image/jpeg");  
    WiFiClient client = server.client();  
    frame->writeTo(client);  
    Serial.println("Image served successfully");  
}  
  
void handle_jpg()  
{  
    Serial.println("Changing resolution");  
  
    if (!esp32cam::Camera.changeResolution(resolution))  
        Serial.println("Failed to set resolution");  
  
    Serial.println("Resolution set successfully");  
  
    Serial.println("Capturing frame");  
    auto frame = esp32cam::capture();  
    if (frame == nullptr)  
    {  
        Serial.println("Failed to capture frame");  
        server.send(503, "", "");  
        return;  
    }  
    Serial.printf("Capture successful: %dx%d, size: %d bytes\n",  
        frame->getWidth(), frame->getHeight(),  
        static_cast<int>(frame->size()));  
  
    if (!frame->isJpeg())  
    {  
        if (!frame->toJpeg(80)) // JPEG quality of 80%  
        {  
            Serial.println("Failed to convert to JPEG");  
            server.send(503, "", "");  
            return;  
        }  
    }  
}
```

```
    }

}

// Serve the JPEG image
server.setContentLength(frame->size());
server.send(200, "image/jpeg");
WiFiClient client = server.client();
frame->writeTo(client);
Serial.println("JPEG served successfully");
}

void setup()
{
    Serial.begin(115200);
    Serial.println();

    {
        using namespace esp32cam;
        Config cfg;
        cfg.setPins(pins::AiThinker);
        cfg.setResolution(Resolution::find(800, 600)); // Initial
resolution setting
        cfg.setBufferCount(2);
        cfg.setJpeg(80);

        bool camera_ok = Camera.begin(cfg);
        Serial.println(camera_ok ? "Camera initialized" : "Failed to
initialize camera");
    }

    WiFi.persistent(false);
    WiFi.mode(WIFI_STA);
    WiFi.begin(WIFI_SSID, WIFI_PASS);
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(500);
    }
    Serial.print("http://");
}
```

```

Serial.println(WiFi.localIP());
Serial.println("/cam.jpg");

server.on("/cam.jpg", handle_jpg);

server.begin();
}

void loop()
{
    server.handleClient();
}

```

6.2. RaspberryPI

6.2.1. main.cpp

```

#include <iostream>
#include <opencv2/opencv.hpp>
#include "image_processor.h"
#include "image_downloader.h"
#include "ocr_processor.h"
#include "notifier.h"

#define CAM_URL_IMAGE "http://10.7.221.193/cam.jpg"

bool debug = false;

int main(int argc, char **argv)
{
    for (int i = 1; i < argc; ++i)
    {
        if (std::string(argv[i]) == "--debug")
        {
            debug = true;
            break;
        }
    }
}

```

```

while (true)
{
    if (debug)
        std::cout << "Starting image processing...\n";

    std::string inputPath = generateFilePath("../assets/",
SUFFIX_INPUT_IMAGE_PATH);
    std::string perspectivePath = generateFilePath("../assets/",
SUFFIX_PERSPECTIVE_IMAGE_PATH, debug);
    std::string thresholdPath = generateFilePath("../assets/",
SUFFIX_THRESHOLD_IMAGE_PATH, debug);
    std::string morphologyPath = generateFilePath("../assets/",
SUFFIX_MORPHOLOGY_IMAGE_PATH, debug);
    std::string floodFillPath = generateFilePath("../assets/",
SUFFIX_FLOOD_FILL_IMAGE_PATH, debug);

    if (!downloadImage(CAM_URL_IMAGE, inputPath))
    {
        std::cerr << "Failed to download image from: " <<
CAM_URL_IMAGE << std::endl;
        return 1;
    }
    if (debug)
        std::cout << "Image saved to: " << inputPath << std::endl;

    cv::Mat image = cv::imread(inputPath, cv::IMREAD_COLOR);
    if (image.empty())
    {
        std::cerr << "Error loading image: " << inputPath <<
std::endl;
        return -1;
    }
    if (debug)
        std::cout << "Image loaded successfully.\n";

    cv::Mat correctedImage = adjustPerspective(image,
perspectivePath, debug);

```

```

    cv::Mat thresholdImage = applyThresholding(correctedImage,
thresholdPath, debug);
    cv::Mat floodFillImage = applyEdgeFloodFill(thresholdImage,
floodFillPath, debug);
    cv::Mat processedImage = applyMorphology(floodFillImage,
morphologyPath, debug);

    float value = performOCR(morphologyPath, debug);

    handleNotify(value, debug);

    int key;
    if (debug)
    {
        std::cout << "Image processing complete.\n";
        key = cv::waitKey(0);
    }
    else
    {
        key = cv::waitKey(10000);
    }

    if (key == 27)
        break;
}

return 0;
}

```

6.2.2. helpers.h

```

#ifndef HELPERS_H
#define HELPERS_H

#include <iostream>
#include <iomanip>
#include <sstream>
#include <ctime>

```

```
#include <string>

std::string getCurrentDateTime(const std::string &format);

#endif // HELPERS_H
```

6.2.3. helpers.cpp

```
#include "helpers.h"

std::string getCurrentDateTime(const std::string &format)
{
    std::time_t now = std::time(nullptr);
    std::tm *ltm = std::localtime(&now);

    char buffer[100];
    std::strftime(buffer, sizeof(buffer), format.c_str(), ltm);

    return std::string(buffer);
}
```

6.2.4. image_downloader.h

```
#ifndef IMAGE_DOWNLOADER_H
#define IMAGE_DOWNLOADER_H

#include <string>

bool downloadImage(const std::string &url, const std::string
&outputPath);

#endif // IMAGE_DOWNLOADER_H
```

6.2.5. image_downloader.cpp

```
#include "image_downloader.h"
#include <curl/curl.h>
#include <fstream>
```

```
size_t WriteCallback(void *contents, size_t size, size_t nmemb,
void *userp)
{
    std::ofstream *outputFile = static_cast<std::ofstream *>
(userp);
    size_t totalSize = size * nmemb;
    outputFile->write(static_cast<char *>(contents), totalSize);
    return totalSize;
}

bool downloadImage(const std::string &url, const std::string
&outputPath)
{
    CURL *curl = curl_easy_init();
    if (!curl)
    {
        return false;
    }

    std::ofstream outputFile(outputPath, std::ios::binary);
    if (!outputFile.is_open())
    {
        curl_easy_cleanup(curl);
        return false;
    }

    curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, &outputFile);
    CURLcode res = curl_easy_perform(curl);
    curl_easy_cleanup(curl);
    outputFile.close();

    return res == CURLE_OK;
}
```

6.2.6. image_processor.cpp

```
#include "image_processor.h"

std::string generateFilePath(const std::string &prefix, const
std::string &suffix, bool debug)
{
    if (debug)
    {
        return prefix + getCurrentDateTime("%Y_%m_%d_%H_%M_%S") +
suffix;
    }
    else
    {
        return prefix + suffix;
    }
}

cv::Mat adjustPerspective(cv::Mat image, const std::string
&inputImage, bool debug)
{
    std::vector<cv::Point2f> srcPoints;
    srcPoints.push_back(cv::Point2f(76, 160)); // upper left
    srcPoints.push_back(cv::Point2f(705, 114)); // upper right
    srcPoints.push_back(cv::Point2f(727, 514)); // lower right
    srcPoints.push_back(cv::Point2f(57, 590)); // lower left

    int largura = cv::max(cv::norm(srcPoints[0] - srcPoints[1]),
                          cv::norm(srcPoints[2] - srcPoints[3]));
    int altura = cv::max(cv::norm(srcPoints[1] - srcPoints[2]),
                         cv::norm(srcPoints[3] - srcPoints[0]));

    std::vector<cv::Point2f> dstPoints;
    dstPoints.push_back(cv::Point2f(0, 0));
    dstPoints.push_back(cv::Point2f(largura, 0));
    dstPoints.push_back(cv::Point2f(largura, altura));
    dstPoints.push_back(cv::Point2f(0, altura));
}
```

```

    cv::Mat perspectiveMatrix =
cv::getPerspectiveTransform(srcPoints, dstPoints);
    cv::Mat correctedImage;
    cv::warpPerspective(image, correctedImage, perspectiveMatrix,
cv::Size(largura, altura));

    cv::imwrite(generateFilePath("../assets/",
SUFFIX_PERSPECTIVE_IMAGE_PATH, debug), correctedImage);
    if (debug)
        cv::imshow(generateFilePath("../assets/",
SUFFIX_PERSPECTIVE_IMAGE_PATH, debug), correctedImage);

    return correctedImage;
}

cv::Mat applyThresholding(const cv::Mat &inputImage, const
std::string &outputPath, bool debug)
{
    if (debug)
        std::cout << "Applying thresholding...\n";
    cv::Mat thresholdImage;
    cv::cvtColor(inputImage, thresholdImage, cv::COLOR_BGR2GRAY);
    cv::threshold(thresholdImage, thresholdImage, 110, 255,
cv::THRESH_BINARY);
    cv::imwrite(outputPath, thresholdImage);
    if (debug)
        cv::imshow(outputPath, thresholdImage);
    return thresholdImage;
}

cv::Mat applyEdgeFloodFill(const cv::Mat &inputImage, const
std::string &outputPath, bool debug)
{
    cv::Mat floodFillImage = inputImage.clone();

    for (int row = 0; row < floodFillImage.rows; row++)
    {
        for (int column = 0; column < floodFillImage.cols; column++)

```

```

    {
        if (row == 0 || column == 0 || row == floodFillImage.rows - 1 || column == floodFillImage.cols - 1)
        {
            if (floodFillImage.at<uchar>(row, column) == 0)
            {
                cv::floodFill(floodFillImage, cv::Point(column, row),
255);
            }
        }
    }
}

int borderSize = 250;
cv::copyMakeBorder(floodFillImage, floodFillImage, borderSize,
borderSize, borderSize, borderSize, cv::BORDER_CONSTANT,
cv::Scalar(255, 0, 255));

cv::imwrite(outputPath, floodFillImage);
if (debug)
    cv::imshow(outputPath, floodFillImage);

return floodFillImage;
}

cv::Mat applyMorphology(const cv::Mat &inputImage, const
std::string &outputPath, bool debug)
{
    std::cout << "Applying morphological operations...\n";
    cv::Mat image = inputImage.clone();
    cv::Mat structuringVerticalElement =
cv::getStructuringElement(cv::MORPH_RECT, cv::Size(1, 5));
    cv::Mat structuringHorizontalElement =
cv::getStructuringElement(cv::MORPH_RECT, cv::Size(5, 1));
    cv::Mat structuringElement =
cv::getStructuringElement(cv::MORPH_RECT, cv::Size(5, 5));

```

```

cv::morphologyEx(image, image, cv::MORPH_OPEN,
structuringElement);
cv::morphologyEx(image, image, cv::MORPH_CLOSE,
structuringElement);

cv::erode(image, image, structuringVerticalElement);

cv::erode(image, image, structuringHorizontalElement);
cv::erode(image, image, structuringHorizontalElement);
cv::erode(image, image, structuringHorizontalElement);
cv::dilate(image, image, structuringHorizontalElement);
cv::dilate(image, image, structuringHorizontalElement);

cv::dilate(image, image, structuringVerticalElement);
cv::dilate(image, image, structuringVerticalElement);
cv::dilate(image, image, structuringVerticalElement);

// cv::imshow(outputPath, image);
cv::imwrite(outputPath, image);
std::cout << "Morphological processing complete.\n";
return image;
}

```

6.2.7. notifier.h

```

#ifndef NOTIFIER_H
#define NOTIFIER_H

#include "helpers.h"

```

```

#include <iostream>
#include <curl/curl.h>

#define MULTIMETER_MINIMUM_VALUE 3.0
#define MULTIMETER_MAXIMUM_VALUE 5.0
#define ADAFRUIT_API_KEY "aio_tzfN83XqK8IvKzgQguivLV7U04Ta"
#define ADAFRUIT_URL
"https://io.adafruit.com/api/v2/Ernane/feeds/segment7-esp32cam-
rpi/data"

bool publishAdafruit(float value, bool debug);
bool whatsNotify(const std::string &number, const std::string
&message, bool debug);
bool handleNotify(float value, bool debug);

#endif // NOTIFIER_H

```

6.2.8. notifier.cpp

```

#include "notifier.h"

bool publishAdafruit(float value, bool debug)
{
    if (debug)
        std::cout << "Adafruit Publish received " << std::endl;

    CURL *curl;
    CURLcode res;
    struct curl_slist *headers = NULL;

    curl = curl_easy_init();
    if (curl)
    {
        std::string postFields = "value=" + std::to_string(value);
        curl_easy_setopt(curl, CURLOPT_URL, ADAFRUIT_URL);
        curl_easy_setopt(curl, CURLOPT_POSTFIELDS,
postFields.c_str());
    }
}

```

```

        headers = curl_slist_append(headers, ("X-AIO-Key: " +
std::string(ADAFRUIT_API_KEY)).c_str());
        curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);
        res = curl_easy_perform(curl);
        if (res != CURLE_OK)
        {
            std::cerr << "cURL error: " << curl_easy_strerror(res) <<
std::endl;
            return false;
        }
        else
        {
            std::cout << "Value published successfully." << std::endl;
            return true;
        }

        curl_slist_free_all(headers);
        curl_easy_cleanup(curl);
    }
else
{
    std::cerr << "Failed to initialize cURL." << std::endl;
    return false;
}
}

bool whatsNotify(const std::string &number, const std::string
&message, bool debug)
{
    if (debug)
        std::cout << "Whats Notify received " << std::endl;

    CURL *curl;
    CURLcode res;
    curl = curl_easy_init();
    if (curl)
    {

```

```
    std::string url = "https://buddy.ernane.dev/api/v1/send-
message";
    std::string jsonBody = "{\"number\":\"" + number +
    "\",\"message\":\"" + message + "\",\"token\":\"3967f4a6-3cd3-
4ded-b08e-3fcfb3dbf6a9\"}";

    std::cout << jsonBody;

    struct curl_slist *headers = NULL;
    headers = curl_slist_append(headers, "Content-Type:
application/json");

    curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
    curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);
    curl_easy_setopt(curl, CURLOPT_POSTFIELDS, jsonBody.c_str());

    res = curl_easy_perform(curl);

    if (res != CURLE_OK)
    {
        if (debug)
            std::cerr << "Erro ao enviar mensagem! 😢" << std::endl
                << curl_easy_strerror(res) << std::endl;
        curl_easy_cleanup(curl);
        return false;
    }

    if (debug)
        std::cout << "Mensagem enviada com sucesso!" << std::endl;

    curl_easy_cleanup(curl);
    return true;
}

if (debug)
    std::cerr << "Erro ao inicializar CURL! 😢" << std::endl;

return false;
```

```

}

bool handleNotify(float value, bool debug)
{
    std::cout << "Multimeter value: " << value << std::endl;
    publishAdafruit(value, debug);
    if (value >= MULTIMETER_MINIMUM_VALUE && value <=
MULTIMETER_MAXIMUM_VALUE)
    {
        std::cout << "Value is within the acceptable range." <<
std::endl;
        return true;
    }
    else
    {
        std::cout << std::endl;
        whatsNotify("CLIENT_NUMBER", "🔴 *Segment7-esp32cam-rpi
Notification* 🔴 \n\n> Value is outside the acceptable
range!\n\n- Current Value: " + std::to_string(value) + "V" +
"\n- Acceptable range: [" +
std::to_string(MULTIMETER_MINIMUM_VALUE) + "V - " +
std::to_string(MULTIMETER_MAXIMUM_VALUE) + "V]\n- Server Time: "
+ getCurrentDateTime("%d/%m/%Y %H:%M:%S") + "", debug);
        std::cout << std::endl
            << "Value is outside the acceptable range." <<
std::endl;
        return false;
    }
}

return false;
}

```

6.2.9. ocr_processor.h

```

#ifndef OCR_PROCESSOR_H
#define OCR_PROCESSOR_H

#include <string>

```

```

#define TESS_DATA_PATH "../assets/tessdata"
#define TESS_LANG "7seg"

float performOCR(const std::string &imagePath, bool debug);

#endif // OCR_PROCESSOR_H

```

6.2.10. ocr_processor.cpp

```

#include "ocr_processor.h"
#include <tesseract/baseapi.h>
#include <leptonica/allheaders.h>
#include <opencv2/opencv.hpp>
#include <iostream>

#include <random>

float performOCR(const std::string &outputPath, bool debug)
{
    float value = 0.0;
    cv::Mat image = cv::imread(outputPath, cv::IMREAD_COLOR);
    if (debug)
        std::cout << "Performing OCR...\n";
    tesseract::TessBaseAPI *api = new tesseract::TessBaseAPI();
    if (api->Init(TESS_DATA_PATH, TESS_LANG,
    tesseract::OEM_DEFAULT))
    {
        std::cerr << "Could not initialize tesseract.\n";
        return value;
    }
    if (debug)
        std::cout << "Tesseract initialized.\n";
    api->SetImage(image.data, image.cols, image.rows, 3,
    image.step);
    if (debug)
        std::cout << "Image set for OCR.\n";

```

```

char *outText = api->GetUTF8Text();
if (outText != nullptr && outText[0] != '\0')
{
    if (debug)
        std::cout << "OCR output: " << outText << std::endl;
    value = atof(outText) / 1000.;
}
else
{
    if (debug)
        std::cout << "OCR output is empty.\n";
}
api->End();
delete[] outText;
delete api;
if (debug)
    std::cout << "OCR complete.\n";

std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> distrib(0.0, 8.0);
value = distrib(gen);

return value;
}

```

6.3. Morphology Viewer System

6.3.1. main.cpp

```

#include <iostream>
#include <opencv2/opencv.hpp>

int main(int argc, char **argv)
{
    cv::Mat image, erosion, dilation, opening, closing, open_close;
    cv::Mat structuring_element;

```

```

if (argc != 2)
{
    std::cout << "Usage: " << argv[0] << " <input_image>\n";
    return -1;
}

// Load the input image
image = cv::imread(argv[1], cv::IMREAD_UNCHANGED);

if (image.empty())
{
    std::cout << "Error loading image: " << argv[1] << std::endl;
    return -1;
}

std::cout << "Structuring Element: \n"
        << structuring_element << std::endl;
cv::imshow("image", image);

bool running = true;
std::cout << "\n== Morphological Operations ==\n\n"
        << "1 - Erosion\n"
        << "2 - Dilation\n"
        << "3 - Opening\n"
        << "4 - Closing\n"
        << "5 - Gradient\n"
        << "6 - TopHat\n"
        << "7 - BlackHat\n"
        << "8 - HitMiss\n"
        << "9 - Opening and Closing\n\n"
        << "[ESC] - Exit\n"
        << "\n=====\\n\\n";

while (running)
{
    int operation = cv::waitKey(0);
    switch (operation)
    {

```

```
case 49: // 1
    std::cout << "-> (1) Erosion\n";
    cv::erode(image, image, structuring_element);
    break;
case 50: // 2
    std::cout << "-> (2) Dilation\n";
    cv::dilate(image, image, structuring_element);
    break;
case 51: // 3
    std::cout << "-> (3) Opening\n";
    cv::morphologyEx(image, image, cv::MORPH_OPEN,
structuring_element);
    break;
case 52: // 4
    std::cout << "-> (4) Closing\n";
    cv::morphologyEx(image, image, cv::MORPH_CLOSE,
structuring_element);
    break;
case 53: // 5
    std::cout << "-> (5) Gradient\n";
    cv::morphologyEx(image, image, cv::MORPH_GRADIENT,
structuring_element);
    break;
case 54: // 6
    std::cout << "-> (6) TopHat\n";
    cv::morphologyEx(image, image, cv::MORPH_TOPHAT,
structuring_element);
    break;
case 55: // 7
    std::cout << "-> (7) BlackHat\n";
    cv::morphologyEx(image, image, cv::MORPH_BLACKHAT,
structuring_element);
    break;
case 56: // 8
    std::cout << "-> (8) HitMiss\n";
    cv::morphologyEx(image, image, cv::MORPH_HITMISS,
structuring_element);
    break;
```

```

    case 57: // 9
        std::cout << "--> (9) Opening and Closing\n";
        cv::morphologyEx(image, image, cv::MORPH_OPEN,
structuring_element);
        cv::morphologyEx(image, image, cv::MORPH_CLOSE,
structuring_element);
        break;
    case 27: // esc
        std::cout << "Exiting...\n";
        running = false;
        break;
    case 118: // v - vertical
        structuring_element =
cv::getStructuringElement(cv::MORPH_RECT, cv::Size(1, 5));
        std::cout << "Vertical Structuring Element: "
                << structuring_element << std::endl;
        break;
    case 104: // h - horizontal
        structuring_element =
cv::getStructuringElement(cv::MORPH_RECT, cv::Size(5, 1));
        std::cout << "Horizontal Structuring Element: "
                << structuring_element << std::endl;
        break;
    case 113: // q - square
        structuring_element =
cv::getStructuringElement(cv::MORPH_RECT, cv::Size(5, 5));
        std::cout << "Square Structuring Element: "
                << structuring_element << std::endl;
        break;
    default:
        std::cout << "Invalid operation: " << operation <<
std::endl;
        break;
    }

    cv::imshow("image", image);
    cv::imwrite("../assets/output.png", image);
}

```

```
    std::cout << "System terminated! Image saved."
    << "=====\\n\\n";
    return 0;
}
```

