UNIVERSIDADE PAULISTA – UNIP
INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS (ICET)

Enrico Casaquia Fernandes
Danilo Storti Movio
Fernando Soares Silva
Vinícius Prado Feliciano
Pedro Henrique Bernardo Lima

MONITORAMENTO DE VIAS PÚBLICAS POR PROCESSAMENTO DE IMAGEM: A TECNOLOGIA ALIADA À MOBILIDADE URBANA

Ribeirão Preto
2025

Enrico Casaquia Fernandes
Danilo Storti Movio
Fernando Soares Silva
Vinícius Prado Feliciano
Pedro Henrique Bernardo Lima

MONITORAMENTO DE VIAS PÚBLICAS POR PROCESSAMENTO DE IMAGEM: A TECNOLOGIA ALIADA À MOBILIDADE URBANA

Monografia submetida ao Curso de Ciência da Computação da Universidade Paulista, para a obtenção do Título de Bacharel em Ciência da Computação.

Orientadores:
Prof. Dr. Avelino Palma Pimenta Júnior
Prof.ª Dr.ª Eliana Leão do Prado Battaglion

Ribeirão Preto
2025

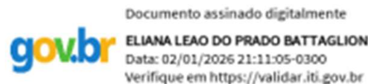
Enrico Casaquia Fernandes
Danilo Storti Movio
Fernando Soares Silva
Vinícius Prado Feliciano
Pedro Henrique Bernardo Lima

MONITORAMENTO DE VIAS PÚBLICAS POR PROCESSAMENTO DE IMAGEM: A TECNOLOGIA ALIADA À MOBILIDADE URBANA

Monografia submetida ao Curso de Ciência da Computação da Universidade Paulista, para a obtenção do Título de Bacharel em Ciência da Computação.

Aprovado em: 04/12/2025 Média Final - MF: 10,0 (dez)

BANCA EXAMINADORA



Prof.ª Dr.ª Eliana Leão do Prado Battaglion
Universidade Paulista – UNIP

A informática, a engenharia de tráfego, a eletrônica, a tecnologia comportamental e a democratização da informação são ferramentas essenciais e que modernamente compõem o que se chama de trânsito inteligente.

(Scaringella, 2001, p.59)

RESUMO

MOVIO, Danilo Storti; FERNANDES, Enrico Casaquia; SILVA, Fernando Soares; LIMA, Pedro Henrique Bernardo; FELICIANO, Vinícius Prado. **MONITORAMENTO DE VIAS PÚBLICAS POR PROCESSAMENTO DE IMAGEM: A TECNOLOGIA ALIADA À MOBILIDADE URBANA.** 2025. Trabalho de Conclusão de Curso (Curso de Ciência da Computação) – Instituto de Ciências Exatas e Tecnologia, Universidade de Paulista, Campus Vargas, Ribeirão Preto, 2025.

Este trabalho apresenta o desenvolvimento do sistema MEGA (Monitoramento Eletrônico e Gerenciamento Avançado), uma solução de visão computacional para monitoramento veicular em tempo real em vias públicas brasileiras. O objetivo consiste em detectar, identificar e acompanhar veículos em fluxos de vídeo provenientes de câmeras do DER-SP (Departamento de Estradas de Rodagem), gerando métricas quantitativas para auxílio ao planejamento urbano, gestão de tráfego e segurança viária. A pesquisa caracteriza-se como aplicada de natureza tecnológica, com abordagem experimental voltada à necessidade de ferramentas automatizadas para análise do tráfego urbano diante da crise de mobilidade urbana no Brasil, contexto discutido ao longo do trabalho. O sistema foi desenvolvido em *Python* utilizando *Flask* e visão computacional com *OpenCV* e *YOLOv11n* para processamento de imagem, além de interface *web* em *Next.js* e *React*. A metodologia permitiu validação e refinamentos progressivos dos módulos, principalmente em relação ao subsistema de captura, processamento de vídeo e exibição de estatísticas em tempo real, detectando cinco classes de veículos, contagem de travessias e identificação automática de congestionamentos. Os resultados demonstraram viabilidade técnica com detecção estável em condições variadas, contribuindo para automação do monitoramento de tráfego no contexto brasileiro.

Palavras-chave: Monitoramento Veicular. Trânsito Inteligente. Smart Cities. YOLO. Python. OpenCV.

LISTA DE FIGURAS

Figura 1 – Arquitetura da Aplicação..13
Figura 2 – Linhas de comando para inicialização configuradas com *argparse* 14
Figura 3 – Exemplificação do *endpoint* para renderização de visualização teste 18
Figura 4 – *Endpoint* para inicialização de *websocket* e processamento da câmera 18
Figura 5 – *Endpoint* para *streaming* do vídeo processado 19
Figura 6 – Apresentação de estrutura e exemplo de funcionalidade de *broadcasting* das estatísticas do *websocket*..20
Figura 7 – Método de conexão com a câmera e configurações de captura 21
Figura 8 – Método executado pela *thread* de captura...24
Figura 9 – Método executado pela *thread* de processamento 25
Figura 10 – Método executado pela *thread* de exibição 26
Figura 11 – Cálculo para identificar posição acima ou abaixo da linha 27
Figura 12 – Linhas de código da lógica de detecção de veículos parados 29
Figura 13 – Prototipagem da Tela Inicial...32
Figura 14 – Prototipagem da Tela de Exibição de Câmera 32
Figura 15 – Display de temperatura – Imagem original...39
Figura 16 – Display de temperatura – Componente verde...39
Figura 17 – Display de temperatura – Threshold aplicado..39
Figura 18 – Display de temperatura – Threshold invertido...39
Figura 19 – Aplicações para o OpenCV 40
Figura 20 – Exemplo de uso do YOLO para reconhecimento de entidades 41
Figura 21 – Exemplo de código fonte *Python* com uso do *OpenCV* 44
Figura 22 – Ilustração do Docker 45
Figura 23 – Interface de visualização da câmera processada 46
Figura 24 – Grid demonstrando diferentes resultados 47
Figura 25 – Detecção de tráfego intenso em funcionamento...50
Figura 26 – Página inicial do sistema MEGA...52
Figura 27 – Página de monitoramento da câmera 53

LISTA DE ABREVIATURAS E SIGLAS

ANSI - American National Standards Institute (Instituto de padronização nacional americano)
API - Application programming interface (interface de programação de aplicação)
COCO – Common Objects in Context (Dataset de visão computacional)
CPU – Central Processing Unit (Unidade central de processamento)
DER-SP – Departamento de Estradas de Rodagem do Estado de São Paulo
FPS – Frames Per Second (Quadros por segundo)
GIL – Global Interpreter Lock (Trava global do interpretador Python)
GPU – Graphics Processing Unit (Unidade de processamento gráfico)
GUI - Graphic user interface (interface gráfica do usuário)
HLS – HTTP Live Streaming (Streaming ao vivo via HTTP)
HTML – HyperText Markup Language (Linguagem de marcação de hipertexto)
HTTP – HyperText Transfer Protocol (Protocolo de transferência de hipertexto)
IA - Inteligência Artificial
ID – Identifier (Identificador)
JPEG – Joint Photographic Experts Group (Padrão de compressão de imagem)
JSON – JavaScript Object Notation (Notação de objeto JavaScript)
LP - Linguagem de programação
MJPEG – Motion JPEG (Sequência de quadros JPEG em vídeo)
MOT - Multiple Object Tracking (Rastreamento multi-objeto)
NPM - Node package manager (Gerenciador de pacotes do node)
PIP – Package installer for python (Instalador de pacotes para python)
REST - Representational State Transfer (transferência de estado representacional)
ROI – Region of interest (Região de interesse)
SEO – Search Engine Optimization (Otimização para mecanismos de busca)
SO - Sistema operacional
UI - User interface (Interface do usuário)
UX - User experience (Experiência do usuário)
YOLO - You only look once

SUMÁRIO

1 INTRODUÇÃO ... 10
2 OBJETIVOS ... 12
2.1 GERAL ... 12
2.2 ESPECÍFICOS ... 12
3 METODOLOGIA ... 13
3.1 EXECUÇÃO LOCAL DO SISTEMA ... 13
3.1.1 Execução nativa com Python ... 14
3.1.2 Execução conteinerizada com *Docker* ... 15
3.2 BACK-END ... 16
3.2.1 Python: a linguagem de programação escolhida ... 16
3.2.2 API REST com Flask e comunicação via *websocket* ... 17
3.2.3 Visão computacional: YOLO e OpenCV ... 20
3.2.4 Arquitetura de processamento e implementação de *buffering* ... 23
3.2.5 Algoritmos de análise de tráfego ... 27
3.3 FRONT-END ... 30
3.4 CONCLUSÃO DA METODOLOGIA ... 33
4. REVISÃO DE LITERATURA ... 33
4.1 MOBILIDADE URBANA ... 33
4.2 SMART CITIES E A TECNOLOGIA COMO ALTERNATIVA ... 34
4.3 INTELIGÊNCIA ARTIFICIAL ... 36
4.3.1 Teste de Turing ... 36
4.3.2 Aprendizado de máquina ... 37
4.4 VISÃO COMPUTACIONAL ... 38
4.4.1 Processamento de imagens ... 38
4.4.1.1 OpenCV ... 40
4.4.1.2 YOLO ... 41
4.5.1 Ferramentas para a interface (front-end) ... 42
4.5.2 Desenvolvimento do sistema (*back-end)* ... 43

4.5.2.1 Python 43
4.5.2.1.1 Flask 44
5 RESULTADOS E DISCUSSÕES 46
5.1 IMPLEMENTAÇÃO DO SISTEMA 46
5.2 VALIDAÇÃO DAS FUNCIONALIDADES 47
5.2.1 Detecção e rastreamento de veículos 47
5.2.2 Contagem por linha virtual 48
5.2.3 Identificação de veículos parados 49
5.2.4 Detecção de tráfego intenso 50
5.3 DESEMPENHO E APRESENTAÇÃO DO SISTEMA 51
5.4 DISCUSSÃO DOS RESULTADOS 54
6 CONSIDERAÇÕES FINAIS 57
REFERÊNCIAS 59

1 INTRODUÇÃO

Na atualidade, passamos por um momento onde o crescimento exponencial dos centros urbanos e aumento do tráfego de veículos é evidente, com isso, a segurança no trânsito vem a tona como uma preocupação de grande prioridade. Em grandes metrópoles, onde a fluidez das vias impacta diretamente na mobilidade urbana, na qualidade de vida da população e, em muitos casos, na preservação de vidas, cada segundo pode ser determinante para evitar tragédias. Nesse cenário, a tecnologia surge não apenas sendo visada como uma ferramenta de suporte, mas também como uma aliada essencial para a prevenção de acidentes, possibilitando a solução imediata de emergências por meio do monitoramento de situações críticas.

A visão computacional, área da inteligência artificial que permite que sistemas realizem a interpretação de imagens e vídeos da mesma forma que os seres humanos, tem se consolidado como uma solução altamente eficiente para a análise automatizada de ambientes urbanos. Combinada com o uso de linguagens de programação como *Python* e bibliotecas especializadas como o *OpenCV*, essa tecnologia tem possibilitado o desenvolvimento de sistemas inteligentes capazes de reconhecer padrões visuais, detectar comportamentos anômalos e identificar eventos relevantes em tempo real.

O avanço dessas ferramentas revolucionou a maneira como problemas complexos, como a detecção de acidentes, engarrafamentos, obstruções nas vias e até mesmo a identificação de veículos de emergência, são tratados. Por meio de algoritmos treinados para reconhecer características específicas em imagens, como movimentações bruscas, colisões entre veículos ou luzes e sirenes de ambulâncias e viaturas, torna-se possível gerar alertas automáticos que contribuem para uma resposta mais ágil das autoridades competentes.

Além disso, esses sistemas podem ser integrados a centros de controle de tráfego, auxiliando na tomada de decisões como a liberação de rotas prioritárias ou o redirecionamento de fluxos para evitar bloqueios. Diante dessa perspectiva, este estudo propõe o desenvolvimento de um sistema de visão computacional capaz de identificar automaticamente situações emergenciais em áreas urbanas.

A partir da utilização do *Python* e *OpenCV*, pretende-se construir uma solução que analise gravações e vídeos de câmeras públicas em tempo real, com o objetivo

de detectar e rastrear veículos, contabilizar travessias através de linhas virtuais, identificar e contabilizar veículos parados e sinalizar situações de tráfego congestionado. Com isso, o sistema se apresentará capaz de exibir estatísticas em tempo real e auxiliar a tomada de decisão, permitindo respostas rápidas e eficazes a incidentes ou situações adversas, promovendo *insights* para identificação de pontos de atenção e oportunidades de ação preventiva e permitindo também que os órgãos sejam capazes de construir estratégias e otimizar o trânsito, minimizando os problemas de mobilidade urbana.

O foco deste trabalho está em explicar como a visão computacional pode ser aplicada, por meio das ferramentas de programação mencionadas, para automatizar a análise contínua do tráfego urbano e das condições das vias, fornecendo subsídio para respostas em tempo hábil e tomadas de decisão. Com isto em mente, a proposta visa não apenas apresentar uma solução funcional, mas também contribuir para o debate sobre o papel das tecnologias inteligentes na construção de cidades mais seguras, resilientes e eficientes.

O método adotado será de natureza prática e exploratória, envolvendo a coleta de vídeos reais de vias públicas, implementação de algoritmos de processamento de imagem e a validação do sistema proposto em diferentes cenários urbanos. O intuito deste projeto é que, ao final, seja capaz demonstrar a viabilidade técnica e operacional de um sistema de monitoramento automatizado, capaz de auxiliar órgãos públicos, serviços de emergência e sistemas inteligentes de transporte.

Essa abordagem tecnológica também abre caminhos para futuras integrações com outras soluções, como redes neurais profundas (*deep learning*), análise preditiva e plataformas de cidades inteligentes, ampliando seu potencial de atuação para além do trânsito. Com isso, pretende-se não só melhorar o tempo de resposta a emergências, mas também criar um ambiente urbano mais seguro, conectado e preparado para lidar com os desafios da vida moderna.

2 OBJETIVOS

2.1 GERAL

Desenvolver um sistema inteligente de análise do tráfego urbano através do processamento de vídeo de câmeras de vias públicas, com a integração de tecnologias de análise de imagem e inteligência artificial para detectar, rastrear e analisar o comportamento de veículos, gerando métricas quantitativas automáticas, sobre o fluxo e condições de tais vias, que possam apoiar a gestão de tráfego e o planejamento urbano.

2.2 ESPECÍFICOS

Utilizar um conjunto de câmeras em pontos estratégicos das rodovias, capazes de capturar imagens em tempo real e enviá-las para um sistema centralizado de processamento de dados.

Integrar o sistema de câmeras com algoritmos de inteligência artificial para identificar e classificar o fluxo de veículos, reconhecendo padrões de tráfego, intensidade do movimento e congestionamentos em diferentes horários do dia.

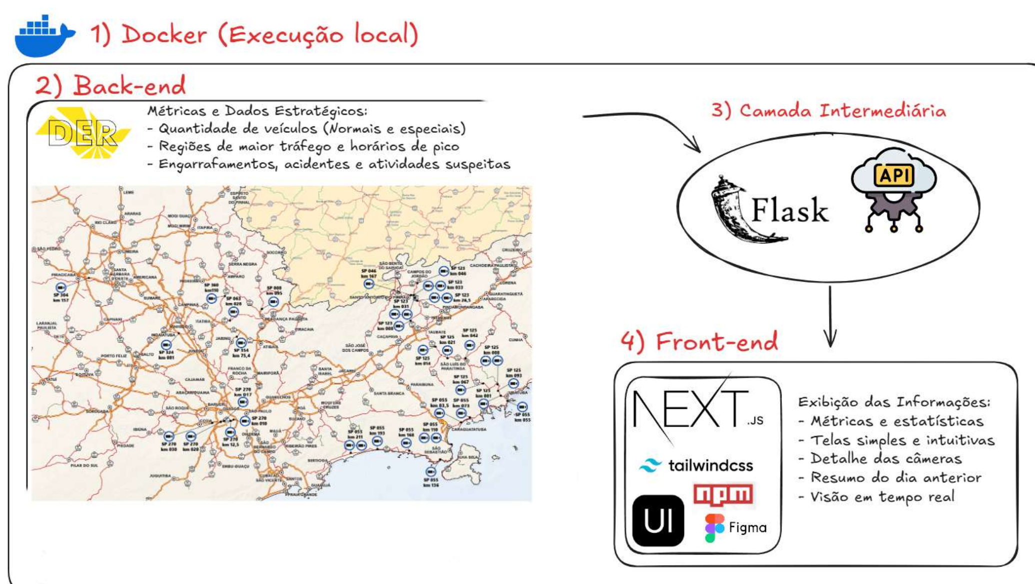
Criar um painel interativo acessível aos usuários por meio do painel *web*, onde possam ser consultadas as condições de trânsito em tempo real, facilitando providências.

Possibilitar, a partir de informações específicas e conhecimentos técnicos, a fiscalização e observação de qualquer rodovia escolhida pelo usuário, por meio de câmeras incluídas no sistema.

3 METODOLOGIA

O aplicativo foi construído e programado pensando, inicialmente, no seu uso por pessoas com um maior entendimento técnico no ramo da ciência da computação e áreas correlatas. Portanto, a estrutura foi devidamente pensada para a maior otimização possível dos recursos disponíveis da máquina do usuário. Sendo assim, todo o código é programado pensando em suas complexidades de tempo e espaço, bem como todo o código fonte é notoriamente documentado para evitar repetições, confusão, e facilitar no desenvolvimento para os envolvidos no projeto. A figura em seguida ilustra o esboço desenhado no momento em que foi pensado e dado os primeiros passos para o desenvolvimento da arquitetura.

Figura 1 – Arquitetura da Aplicação



Fonte: Autoria Própria, 2025

Seguindo os pontos da figura, pode-se aprofundar sobre esses assuntos.

3.1 EXECUÇÃO LOCAL DO SISTEMA

A arquitetura do sistema foi projetada para execução local, significando que a aplicação não é hospedada em servidores externos e não depende de quaisquer outras máquinas ou serviços para o funcionamento adequado. Todos os componentes da aplicação, como o processamento de imagem, a interface gráfica e a transmissão

de dados, operam na máquina do próprio usuário (ambiente local). Essa abordagem foi adotada para o presente trabalho visando maior praticidade de desenvolvimento e controle sobre recursos computacionais, reduzindo também os custos e facilitando a implantação em ambientes de pesquisa.

Embora o sistema se conecte a fontes externas de vídeo via *streaming M3U8* (as câmeras públicas de trânsito), o processamento computacional, que envolve a detecção de objetos com *YOLOv11*, a análise de *frames* em tempo real e o gerenciamento de *buffers* de vídeo, ocorre integralmente no ambiente local. Desta maneira, o usuário mantém total autonomia sobre a execução, podendo ajustar parâmetros pelos arquivos de configuração facilmente dispostos, monitorar o desempenho e inspecionar resultados dependendo apenas de seu próprio *hardware*.

3.1.1 Execução nativa com Python

A forma primária de execução do sistema desenvolvido, para efeitos de projeto acadêmico, é através do interpretador *Python* diretamente, sem necessidade de *conteinerização*. O sistema utiliza o gerenciador de pacotes conhecido como “*PIP*” para a instalação automática das dependências, que ficam disponíveis em arquivo de requisitos separado, no interior do repositório do *back-end*.

Para simplificação do processo de execução do programa, foram criadas linhas de comando por meio de argumentos, cada uma executando uma tarefa específica, implementados através de uma biblioteca específica chamada “*argparse”*, como demonstrado na figura 2:

Figura 2 – Linhas de comando para inicialização configuradas com *argparse*

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="MEGA - Monitoramento Eletrônico e Gerenciamento Avançado")
    parser.add_argument('--install', action='store_true',help='Instala as dependências')
    parser.add_argument('--check', action='store_true',help='Verifica dependências e configuração')
    parser.add_argument('--execute', action='store_true',help='Executa sistema')
    args = parser.parse_args()
```

Fonte: Autoria própria, 2025

3.1.2 Execução conteinerizada com *Docker*

Para realizar a execução local de forma encapsulada e não invasiva, utiliza-se as ferramentas *Docker* e *Docker-compose,* que representam uma alternativa simplificada à execução nativa clonando o repositório.

O *Docker* é uma ferramenta de suma importância para desenvolvedores, pois ele engloba todas as dependências do projeto em camadas de compatibilidade acima do nível do sistema operacional, as quais são chamadas de "contêineres". Esses contêineres ficam responsáveis por armazenar cada uma das ferramentas diversas que a aplicação utiliza sem, necessariamente, instalá-las na máquina do usuário. Por isso o uso das palavras "encapsulada" e "não invasiva" anteriormente: encapsulada, pois está tudo empacotado em uma só organização, não em vários *downloads* distintos; não invasiva, pois evita-se a instalação de pequenas aplicações necessárias de forma espalhada e, possivelmente desconhecida, pelo usuário que tem interesse na aplicação, evitando, assim, desentendimento, com a possibilidade da instalação de sistemas que, após o uso da nossa aplicação e de uma eventual desinstalação, poderiam "sobrar" no dispositivo.

O *Docker-compose* é uma extensão ao *Docker* que permite orquestrar o uso de múltiplos contêineres simultaneamente através de um único arquivo de configuração, localizado no diretório raiz do repositório. No contexto deste projeto, o arquivo define dois serviços principais: o *back-end* em *Python* (executado na porta 5000) e o *front-end* em *Next.Js* (na porta 3000).

Com toda essa configuração, a arquitetura permite que ambos os serviços sejam inicializados simultaneamente, através de um único comando, dispensando, como mencionado anteriormente, a instalação manual do *Python*, *Node.js* e todas as bibliotecas dependentes. O usuário necessita apenas de ter o *Docker* instalado no sistema operacional e executar o comando "*docker-compose up*", com o terminal executando no interior da pasta raiz do projeto. A partir desse comando, o *Docker-compose* automaticamente baixa as imagens base e instala as dependências necessárias especificadas, configurando a comunicação entre os contêineres e inicializando os serviços.

Esta abordagem mencionada se mostra vantajosa no contexto acadêmico e de demonstração em que o projeto se insere, tendo em vista que facilita a replicação da aplicação para a máquina, permitindo que avaliadores e interessados testem o

sistema rapidamente, além de garantir que o sistema possa rodar em diferentes máquinas com diferentes configurações.

3.2 BACK-END

O *back-end* é o coração da aplicação e o funcionamento do sistema em si. No sistema desenvolvido, ele é o responsável pela implementação das lógicas de negócio, o processamento das imagens em tempo real, a detecção dos veículos, o gerenciamento e manipulação dos dados e, por fim, pela comunicação com a camada de apresentação (*front-end*).

3.2.1 Python: a linguagem de programação escolhida

A linguagem usada para construir a aplicação foi o *Python,* devido a diversos motivos técnicos. A linguagem possui um ecossistema bastante completo e compatível com diversas bibliotecas e ferramentas especializadas em visão computacional, processamento de vídeo e imagens e aprendizado de máquina, como *OpenCV*, *Ultralytics* (*YOLO*) e *NumPy*, tecnologias tais que já são amplamente aplicadas pela comunidade em contextos semelhantes aos do projeto, garantindo confiança e segurança.

Além disso, o *Python* oferta suporte nativo a programação concorrente através de *threads*, que constituem uma peça fundamental do sistema desenvolvido, permitindo a implementação da arquitetura *multi-thread*. Embora o *Global Interpreter Lock* (*GIL*) do *Python* imponha limitações na execução paralela verdadeira de código puro, as operações computacionalmente intensivas, como processamento de imagens com *OpenCV* e inferência com *YOLO*, são executadas em bibliotecas compiladas em *C/C++*, que liberam o *GIL* durante sua execução, permitindo paralelismo efetivo.

A parte das dependências foi tratada através do gerenciamento utilizado *PIP* (*Package Installer for Python*), gerenciador de pacotes padrão do *Python*. Todas as bibliotecas necessárias, juntamente às versões específicas estão documentadas no arquivo “*requirements.txt*”, disponibilizado no repositório público.

Por fim, a facilidade de integração do *Python* com outras tecnologias utilizadas no projeto, como o *framework Flask* para a construção de *APIs* web e a biblioteca *Flask-SocketIO* para comunicação em tempo real, através de *websockets*, consolida a justificativa da escolha dessa linguagem com embasamento suficiente.

3.2.2 API REST com Flask e comunicação via *websocket*

Para a comunicação entre o *back-end* e o *front-end* do sistema foram definidas duas estratégias complementares, configurando uma *API REST* implementada com *Flask* para operações síncronas e solicitações pontuais, e um *websocket* para envio contínuo dos dados e estatísticas obtidos através do processamento dos vídeos das câmeras públicas.

De forma concisa, uma *API* (*Application Programming Interface*) *REST* (*Representational State Transfer*) se trata de uma arquitetura para construção de serviços *web* que utiliza o protocolo *HTTP* como meio de comunicação. No contexto do sistema, a *API* expõe as funcionalidades do *back-end* através dos *endpoints*, definidos utilizando *Flask*, que permitem que o *front-end* solicite operações como a inicialização do processamento do vídeo de uma determinada câmera. Através desta arquitetura definida, garante-se uma separação de responsabilidades bem definida, com o *front-end* concentrando-se apenas na apresentação e interação com o usuário, enquanto o *back-end* se dedica às lógicas de negócio e processamento, como mencionado anteriormente.

A escolha do *Flask* para este projeto se fundamenta em suas características técnicas específicas. Para começar, o *Flask* oferece apenas componentes que são essenciais para a construção de aplicações *web*, delegando funcionalidades adicionais a extensões opcionais, o que resulta em um código mais enxuto, facilitando posteriores manutenções, depurações e ajustes. Além disso, o *Flask* não impõe estruturas tão rígidas de organização, o que permitiu que os desenvolvedores arquitetassem a aplicação de acordo com suas necessidades específicas, possibilitando a implementação de uma arquitetura customizada com *threads* de processamento e *buffers*.

A estrutura da *API* implementada consistiu em três *endpoints* principais: o primeiro deles serve para renderizar uma interface *HTML* de teste e visualização, gerando dinamicamente através do sistema de *templates Jinja2* informações sobre a configuração do sistema, modelo *YOLO* utilizado e câmera ativa; o segundo é responsável por inicializar o processamento de uma câmera específica a partir de seu código identificador, encerrando instâncias anteriores, criando um novo sistema de captura e iniciando a transmissão de estatísticas via w*ebsocket*; e o terceiro implementa o *streaming* contínuo de vídeo processado através do protocolo *MJPEG*, enviando *frames* codificados em *JPEG* como partes sequenciais de uma resposta *HTTP* contínua, aguardando a inicialização completa do sistema antes de iniciar a transmissão para prevenir erros de acesso a recursos não disponíveis.

Figura 3 – Exemplificação do *endpoint* para renderização de visualização teste

```
@app.route('/')
def index():
    config_data = {
        'model_name': YOLO_CONFIG.get('model_path').split('/')[-1],
        'selected_camera': SYSTEM_STATE.selected_camera if SYSTEM_STATE.selected_camera else 'Nenhuma câmera',
        'detection_legend': VEHICLE_COLORS
    }
    return render_template('index.html', config=config_data)
```

Fonte: Autoria própria, 2025

Figura 4 – *Endpoint* para inicialização de *websocket* e processamento da câmera

```
@app.route('/start/<string:codigo_camera>')
def start_camera(codigo_camera):
    try:
        if SYSTEM_STATE.mega_cam_system:
            try:
                SYSTEM_STATE.mega_cam_system.stop()
            except Exception:
                pass
        stop_websocket_broadcast()
        SYSTEM_STATE.selected_camera = codigo_camera

        mega_cam_system = MegaBufferedCameraSystem(
            url=CAMERA_URLS.get(codigo_camera),
            height=CAMERA_CONFIG['height'],
            width=CAMERA_CONFIG['width'],
            buffer_delay=STREAM_CONFIG['buffer_delay'],
            max_buffer_size=STREAM_CONFIG['buffer_size']
        )
        SYSTEM_STATE.mega_cam_system = mega_cam_system
        start_websocket_broadcast(mega_cam_system)
        return {'message': 'OK'}, 200
    except Exception as e:
        print(f"\n❌ Erro ao inicializar câmera: {e}")
        sys.exit(1)
```

Fonte: Autoria própria, 2025

Figura 5 – *Endpoint* para *streaming* do vídeo processado

```python
@app.route('/video_feed')
def video_feed():
    # Tempo máximo de espera (em segundos)
    timeout = 10
    start_time = time.time()

    # Espera até o sistema estar pronto
    while True:
        system = getattr(SYSTEM_STATE, "mega_cam_system", None)
        if system is not None:
            break
        if time.time() - start_time > timeout:
            return jsonify({
                "error": "Camera system not initialized",
                "detail": "mega_cam_system is None after waiting 10 seconds"
            }), 503
        time.sleep(0.1)  # Evita loop travando CPU

    try:
        return Response(
            generate_mega_smooth_frames(system, STREAM_CONFIG['target_fps']),
            mimetype='multipart/x-mixed-replace; boundary=frame'
        )
    except Exception as e:
        return jsonify({
            "error": "Failed to generate video feed",
            "detail": str(e)
        }), 500
```

Fonte: Autoria própria, 2025

Mesmo com a *API* definida para operações pontuais, a construção do sistema MEGA exige transmissão contínua de dados, garantida em tempo real, com estatísticas sobre o tráfego sendo atualizadas a cada segundo, enquanto o vídeo é processado. Para esta necessidade, o padrão de comunicação baseado em solicitações e respostas do *HTTP* não é eficiente, pois exigiria *polling* constante do cliente, gerando tráfego de rede desnecessário e latência perceptível.

Por esse motivo, optou-se pela implementação de um *websocket,* que, diferentemente do *HTTP*, onde o cliente sempre inicia a comunicação, permite que tanto cliente quanto servidor enviem mensagens a qualquer momento após o estabelecimento da conexão, caracterizando uma comunicação bidirecional em tempo real. A implementação no sistema MEGA utilizou a biblioteca *Flask-SocketIO*, que integra a funcionalidade *Socket.IO*.

Figura 6 – Apresentação de estrutura e exemplo de funcionalidade de *broadcasting* das estatísticas do *websocket*

```python
from flask_socketio import SocketIO, emit
import threading

socketio = SocketIO(cors_allowed_origins="*", async_mode="threading")

_ws_thread = None
_ws_stop_event = None
_ws_lock = threading.Lock()

@socketio.on("connect")
def handle_connect():
    emit("server_message", {"status": "connected"})

def stats_broadcast_loop(cam_system_instance, stop_event):
    """Thread dedicada para transmitir estatísticas via WebSocket"""
    while not stop_event.is_set():
        try:
            if cam_system_instance and cam_system_instance.is_healthy():
                all_stats = cam_system_instance.get_mega_stats()

                payload = {
                    "total_objects": all_stats.get("last_vehicle_count", 0),
                    "crossing_count": all_stats.get("crossing_count", 0),
                    "stopped_vehicles": all_stats.get("stopped_vehicles", 0),
                    "traffic_detected": all_stats.get("traffic_detected", 0),
                    "fps": round(all_stats.get("fps_display", 0), 2),
                    "uptime_minutes": round(all_stats.get("uptime_minutes", 0), 1),
                    "buffer_capture_size": all_stats.get("buffer_sizes", {}).get("capture", 0),
                }

                socketio.emit("stats_update", payload)

        except Exception as e:
            print(f"Erro na thread do WebSocket: {e}")

        if stop_event.wait(1):
            break
```

Fonte: Autoria própria, 2025

Com isso, foi possível estabelecer um evento customizado emitido a cada segundo que transmite, através de um *thread* de *broadcasting*, um objeto *JSON* contendo estatísticas atualizadas do sistema contagem total de veículos na tela, número de travessias da linha virtual, lista de *IDs* de veículos parados, indicador *booleano* de tráfego intenso, taxa de *frames* por segundo (*FPS*) e tempo de atividade do sistema em minutos.

3.2.3 Visão computacional: YOLO e OpenCV

As principais funcionalidades do sistema (detecção e rastreamento de veículos em vídeo) só é possível por meio da implementação através da combinação de duas poderosas bibliotecas de visão computacional, o *OpenCV* e o *YOLO* (*You Only Look Once*). Esmiuçando as responsabilidades de cada uma dessas bibliotecas, o *OpenCV* foi utilizado para operações fundamentais de processamento de imagem e vídeo e o

YOLO para detecção de objetos baseada em aprendizado profundo e inteligência artificial.

No contexto do MEGA, o *OpenCV* é responsável por diversas atribuições importantes. Uma de suas responsabilidades mais fundamentais se trata do estabelecimento da conexão com os *streams* de vídeo provenientes das câmeras públicas do DER-SP, através do protocolo HLS/M3U8. O método utilizado para conexão implementa configurações específicas de captura, incluindo o valor do *buffer* interno, taxa de *frames* desejada e dimensão do quadro capturado.

Figura 7 – Método de conexão com a câmera e configurações de captura

```
def _init_mega_capture(self):
    """Inicia Captura da Câmera"""
    try:
        self.cap = cv2.VideoCapture(self.url)
        if self.cap.isOpened():
            self.cap.set(cv2.CAP_PROP_BUFFERSIZE, CAMERA_CONFIG['buffer_size'])
            self.cap.set(cv2.CAP_PROP_FPS, STREAM_CONFIG['target_fps'])
            self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, self.width)
            self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, self.height)

            self.logger.info(f"✅ Iniciando câmera")

            for i in range(5):
                ret, frame = self.cap.read()
                if ret and frame is not None:
                    frame = cv2.resize(frame, (self.width, self.height))
                    self.last_good_frame = frame.copy()
                    self.logger.info(f"✅ Frame teste {i+1}/5 capturado")
                time.sleep(self.sleep_time)

            self.initialized = True
        else:
            self.logger.error(f"❌ Falha ao conectar: {self.url}")
    except Exception as e:
        self.logger.error(f"❌ Erro na inicialização: {e}")
```

Fonte: Autoria própria, 2025

Após a captura, ainda utilizando a biblioteca do *OpenCV,* os *frames* recebidos são redimensionados para dimensões padronizadas (640x480), com o objetivo de reduzir a carga computacional nas etapas subsequentes do processamento, garantir consistência nas coordenadas de detecção, independentemente da resolução original da câmera, e otimizar o uso de banda na transmissão do vídeo recém-processado.

Com relação ao *YOLO*, este representa uma família de arquiteturas de redes neurais convolucionais projetadas especificamente para detecção de objetos em tempo real. Para o sistema MEGA, foi selecionada especificamente a versão *YOLOV11n* (“nano”), sendo está a versão mais compacta e otimizada da família “11”.

Esta escolha se fundamenta principalmente na busca por precisão e velocidade de processamento, considerando a utilização de *hardware* convencional. Versões mais robustas da família 11, por exemplo, seriam capazes de entregar resultados ainda mais precisos, mas exigiriam *hardware* mais potente.

Este modelo do *YOLO* utilizado na aplicação foi pré-treinado seguindo a base de dados do *COCO* (*Common Objects in Context*), que contém mais de 330 mil imagens com 80 classes de objetos anotadas. Entretanto, buscando precisão e otimização na identificação dos objetos, evitando processamento de classes desnecessárias, as classes relevantes ao *YOLO* foram limitadas através de código para as instâncias de carros (classe 2), motocicletas (classe 3), ônibus (classe 5), caminhões e caminhonetes grandes (classe 7) e bicicletas (classe 1).

Além disso, um aspecto fundamental da implementação é o uso de métodos de rastreamento multi-objeto (*MOT – Multiple Object Tracking*), em razão da necessidade de manter a identificação contínua e consistente dos objetos de diferentes classes, persistindo um identificador para cada veículo rastreado através de algoritmos que correlacionam detecções entre *frames* por meio de características visuais e proximidade espacial.

Este rastreamento persistente é essencial para as funcionalidades de contagem e detecção de paradas implementadas no sistema. Sem ele, seria impossível determinar se um veículo cruzou a linha virtual (pois não haveria como saber que se trata de uma mesma instância em *frames* consecutivos e diferentes) ou se está parado (pois cada detecção seria tratada como um novo objeto). Importante destacar que, com o algoritmo desenvolvido, mesmo que um objeto detectado se apresente temporariamente ocluso, ele deverá manter o identificador recebido inicialmente após se tornar visível novamente no vídeo.

O método principal da lógica de detecção recebe um *frame* como entrada e retorna uma tupla contendo: o próprio *frame*, agora anotado com visualizações, a contagem de veículos detectados neste *frame,* a contagem de travessias acumulada (total de veículos que cruzaram a linha traçada na câmera) e uma lista com os códigos identificadores dos veículos parados.

A visualização dos resultados da detecção é visível no vídeo processado como caixas que envolvem os veículos (cuja cor depende da classe detectada), e pontos

circulares coloridos que sinalizam veículos parados (em azul) e veículos cruzando a linha definida (em amarelo). Além disso, cada um dos veículos detectados é acompanhado por seu devido identificador, apresentado ao topo da caixa delimitadora. O restante dos dados e estatísticas são enviados à interface do usuário, para o *front-end*, onde são renderizados dentro de *cards* estilizados.

3.2.4 Arquitetura de processamento e implementação de *buffering*

A arquitetura do sistema implementa processamento *multi-thread* e *buffering*, estratégia adotada com o intuito de garantir fluidez no *streaming* do vídeo processado, mesmo enfrentando as latências variadas na captura e o processamento intensivo de vídeo contendo diversos veículos.

Ao refletir sobre e aplicar o processamento do vídeo em tempo real, percebe-se que a captura de *frames* de uma câmera, o processamento desses *frames* através de redes neurais e a codificação para transmissão são operações com tempos de execução variáveis e potencialmente bloqueantes. Neste cenário, executar essas operações sequencialmente levaria a uma latência inaceitável considerando o objetivo do projeto.

Por esse motivo, foi adotada uma solução onde o *pipeline* é decomposto em três *threads*, cada uma responsável por uma etapa específica e comunicando-se através de filas chamadas de *thread-safe*.

As funcionalidades dos *threads* são, respectivamente, capturar, processar e exibir. A primeira *thread* é responsável exclusivamente por capturar *frames* da câmera o mais rapidamente possível, operando em um *loop* infinito que sempre obtém o próximo *frame* disponível, redimensionando-o para as dimensões padrão, e inserindo-o no *buffer* de captura. Cada *frame* inserido é enriquecido com metadados incluindo *timestamp* de captura, *ID* sequencial e tempo de sistema, informações posteriormente utilizadas para cálculo de latências e diagnóstico de desempenho.

Importante destacar que esta mesma *thread* monitora a saúde da aplicação, contabilizando falhas do sistema de captura e, em caso de muitos erros, aplicando um método de reconexão da câmera. Além disso, ela é responsável por manter *frames* de *backup, que* são exibidos por alguns instantes caso a transmissão falhe temporariamente.

Figura 8 – Método executado pela *thread* de captura

```
def _mega_capture_loop(self):
        try:
            ret, frame = self.cap.read()
            current_time = time.time()

            if ret and frame is not None:
                consecutive_failures = 0

                frame = cv2.resize(frame, (self.width, self.height))
                self.last_good_frame = frame.copy()
                self.backup_frame_cache.append(frame.copy())

                timestamped_frame = {
                    'frame': frame,
                    'timestamp': current_time,
                    'frame_id': frame_count,
                    'capture_time': current_time
                }

                try:
                    self.capture_buffer.put_nowait(timestamped_frame)
                except queue.Full:
                    try:
                        self.capture_buffer.get_nowait()
                        self.capture_buffer.put_nowait(timestamped_frame)
                    except queue.Empty:
                        pass

                self.stats['total_frames_captured'] += 1

                frame_count += 1
                if frame_count % STREAM_CONFIG['target_fps'] == 0:
                    fps = STREAM_CONFIG['target_fps'] / (current_time - last_fps_time)
                    self.stats['fps_capture'] = fps
                    last_fps_time = current_time
```

Fonte: Autoria própria, 2025

A *thread* de processamento é responsável por executar um método que consome *frames* do *buffer* de captura e aplica o modelo *YOLO* para detecção, o que se configura como a operação mais intensiva do sistema computacionalmente.

Figura 9 – Método executado pela *thread* de processamento

```
def _mega_process_loop(self):
    """Thread de Veiculos Detectados"""
    process_count = 0
    last_fps_time = time.time()

    self.logger.info(f"🎯 Iniciando Thread Veiculos Detectados para {self.url}")

    while self.running:
        try:
            timestamped_frame = self.capture_buffer.get(timeout=self.buffer_delay)
            current_time = time.time()

            frame = timestamped_frame['frame']
            frame_id = timestamped_frame['frame_id']

            try:
                processed_frame, vehicle_count, crossing_count, stopped_vehicles = self.yolo_detector.detect(frame)
                self.stats['last_vehicle_count'] = vehicle_count
                self.stats['crossing_count'] = crossing_count
                self.stats['stopped_vehicles'] = stopped_vehicles
                traffic_detected = len(self.stats['stopped_vehicles']) >= 8
                self.stats['traffic_detected'] = traffic_detected

            except Exception as e:
                processed_frame = frame
                vehicle_count = 0
                if frame_id % 100 == 0:
                    self.logger.warning(f"⚠️ Erro no YOLO: {e}")

            processed_item = {
                'frame': processed_frame,
                'timestamp': timestamped_frame['timestamp'],
                'frame_id': frame_id,
                'vehicle_count': vehicle_count,
                'crossing_count': crossing_count,
                'stopped_vehicles': stopped_vehicles,
                'traffic_detected': traffic_detected,
                'process_time': current_time,
                'capture_time': timestamped_frame['capture_time']
            }

            try:
                self.process_buffer.put_nowait(processed_item)
            except queue.Full:
                try:
                    self.process_buffer.get_nowait()
                    self.process_buffer.put_nowait(processed_item)
                except queue.Empty:
                    pass

            self.stats['total_frames_processed'] += 1

            process_count += 1
            if process_count % STREAM_CONFIG['target_fps'] == 0:
                fps = STREAM_CONFIG['target_fps'] / (current_time - last_fps_time)
                self.stats['fps_process'] = fps
                last_fps_time = current_time

            time.sleep(self.sleep_time)

        except queue.Empty:
```

Fonte: Autoria própria, 2025

Destaca-se que a *thread* em questão respeita o atraso configurado de *buffer* ao consumir os *frames*, garantindo que sempre processe *frames* com pelo menos N segundos de em tempo de vida, definido por padrão como 2 segundos. Isso protege o sistema de variações na taxa de captura e no tempo de processamento, deixando sempre alguns *frames* disponíveis mesmo que o processamento individual demore mais que o esperado, e permitindo que a taxa de captura e de processamento operem em ritmos independentes.

Por fim, após a detecção, o *frame* processado e as estatísticas associadas a ele são inseridos em um *buffer* de processamento, que será posteriormente consumido pela *thread* final, a de exibição.

Figura 10 – Método executado pela *thread* de exibição

```python
def _mega_display_loop(self):
    while self.running:
        try:
            processed_item = self.process_buffer.get(timeout=self.buffer_delay)
            current_time = time.time()

            frame_age = current_time - processed_item['timestamp']
            wait_time = max(0, self.buffer_delay - frame_age)

            if wait_time > 0:
                self.logger.debug(f"⏰ Aguardando {wait_time:.2f}s para mega buffer")
                time.sleep(wait_time)

            display_item = processed_item.copy()
            display_item['display_time'] = time.time()
            display_item['total_latency'] = display_item['display_time'] - display_item['timestamp']
            display_item['buffer_delay'] = self.buffer_delay

            try:
                self.mega_display_buffer.put_nowait(display_item)
            except queue.Full:
                try:
                    self.mega_display_buffer.get_nowait()
                    self.mega_display_buffer.put_nowait(display_item)
                except queue.Empty:
                    pass

            self.stats['total_frames_displayed'] += 1
            self.stats['last_frame_time'] = time.time()

            display_count += 1
            if display_count % STREAM_CONFIG['target_fps'] == 0:
                fps = STREAM_CONFIG['target_fps'] / (time.time() - last_fps_time)
                self.stats['fps_display'] = fps
                last_fps_time = time.time()
```

Fonte: Autoria própria, 2025

Nesta etapa final, os quadros são consumidos e disponibilizados para *streaming,* assim que atingem a “idade” especificada na configuração de atraso mencionada anteriormente (2 segundos, por padrão). Isso garante uma taxa de transmissão estável ao usuário. O *streaming* é feito via *HTTP*, implementando o protocolo *MJPEG*, através de um gerador *Python* que produz segmentos de resposta *multipart*, cada um contendo um *frame JPEG* codificado. Essa estrutura é reconhecida facilmente por navegadores modernos, que por sua vez automaticamente descartam os *frames* antigos e exibem os novos, conforme a chegada. Portanto, pode-se dizer que a visualização contínua em formato de vídeo, na verdade, se trata de uma exibição sequencial de imagens estáticas.

3.2.5 Algoritmos de análise de tráfego

Além da parte da detecção dos veículos, o sistema, como já mencionado, envia também outras estatísticas a respeito da câmera visualizada. Isso é possível em razão da implementação de três algoritmos especializados que extraem essas estatísticas, sendo eles: algoritmo de contagem de travessias por meio de linha virtual, detecção de veículos parados e identificação de tráfego pesado.

A contagem de travessias de veículos por meio do desenho de uma linha é uma técnica muito utilizada em sistemas de monitoramento e processamento de vídeo, no geral, e principalmente em sistemas que acompanham o trânsito. No sistema desenvolvido, esta linha é traçada horizontalmente na região central do campo de visão da câmera (onde se espera obter maior visibilidade dos veículos), sendo definida por dois pontos, um inicial e um final, cujas coordenadas são especificadas e podem ser alteradas por meio do arquivo de configurações do sistema.

O algoritmo baseia-se em um cálculo que determina o centro da caixa delimitadora do objeto identificado, e avalia a posição deste ponto em relação à linha, indicando se está acima ou abaixo. Para detectar a travessia, o sistema mantém em memória a última posição de cada veículo relativa à linha, que é indexada pelo identificador do respectivo veículo. Com isso, a cada novo *frame*, a posição atual é comparada com a posição anterior (mantida em memória) e verifica-se se houve mudança, ou seja, se o veículo estava acima e passou para abaixo ou vice-versa. Ao cruzar a linha, o veículo é momentaneamente marcado com um círculo amarelo, que funciona como um *feedback* visual ao usuário, apresentando que a travessia foi identificada e contabilizada.

Figura 11 – Cálculo para identificar posição acima ou abaixo da linha

```
34
35      def _is_above_line(self, point):
36          """Verifica se um ponto está acima da linha"""
37          x, y = point
38          x1, y1 = self.line_start
39          x2, y2 = self.line_end
40
41          position = (y2 - y1) * (x - x1) - (x2 - x1) * (y - y1)
42          return position > 0
43
```

Fonte: Autoria própria, 2025

O contador de travessias é acumulativo durante toda a sessão de monitoramento de uma câmera. Assim que o usuário troca de câmera o contador é zerado através da destruição e recriação do objeto de detecção.

Uma limitação importante desta implementação é que ela conta travessias bidirecionais, ou seja, independente de qual a direção do veículo identificado. Para aplicações onde a direção é relevante, seria interessante manter informação adicional sobre a direção do movimento juntamente aos dados do veículo e filtrar travessias de acordo. Além disso, destaca-se que o algoritmo não assume uma visualização da câmera contínua, ou seja, que toda a trajetória do veículo será observada. Se um veículo mover-se muito rapidamente ou se houver perda de *frames*, é possível que o veículo apareça de um lado da linha em um *frame* e do outro lado no *frame* seguinte sem que sua trajetória intermediária tenha sido observada e, assim, a travessia seria possivelmente ignorada.

Para a identificação de veículos parados, a abordagem foi um pouco mais complexa do que no caso da detecção de travessias. Isso porque, para a realização desta análise, é necessário considerar o comportamento de cada veículo dentro de um espaço de tempo considerável.

Com esse objetivo, o sistema possui três parâmetros configuráveis através do arquivo de configurações, sendo eles: o tempo mínimo em segundos que um veículo deve permanecer parado antes de ser considerado como tal; o limiar de movimento relativo ao tamanho do veículo; e o número mínimo de *frames* consecutivos que o veículo deve estar imóvel antes de iniciar a contagem do tempo parado.

Em paralelo, o sistema mantém dados detalhados para cada veículo em um dicionário específico para essa detecção, armazenado a última posição do veículo, o *timestamp* do momento em que o veículo começou a ser observado como estacionário, um *booleano* indicando se o veículo está classificado como parado e um contador de *frames* nos quais o veículo foi observado como estacionário.

A cada novo *frame*, verifica-se se o veículo já possui estado registrado, caso não, cria-se uma entrada inicial com a posição atual, *timestamp* atual e *booleano* registrado como falso. Em seguida, calcula-se a distância euclidiana entre a posição atual e a última registrada. Se o movimento relativo for menor do que o limiar mencionado anteriormente, a quantidade de *frames* em que o veículo permaneceu

parado é incrementada e, por fim, se essa quantidade atingir o valor mínimo determinado, o veículo recebe o status de “parado” através da alteração do valor do *booleano* para verdadeiro.

Figura 12 – Linhas de código da lógica de detecção de veículos parados

```
# --- LÓGICA DE DETECÇÃO DE PARADA ---
if vehicle_id not in self.stopped_states:
    self.stopped_states[vehicle_id] = {
        'last_position': current_center,
        'stopped_time': current_time,
        'is_stopped': False
    }
else:
    state = self.stopped_states[vehicle_id]
    last_pos = state['last_position']

    bbox_size = np.sqrt((x2 - x1) * (y2 - y1))
    relative_movement = np.linalg.norm(
        np.array(current_center) - np.array(last_pos)
    ) / bbox_size

    if relative_movement < self.movement_threshold:
        if 'stopped_frames' not in state:
            state['stopped_frames'] = 1
        else:
            state['stopped_frames'] += 1


        if state['stopped_frames'] >= self.min_stopped_frames:
            if not state['is_stopped']:
                if (current_time - state['stopped_time']) >= self.stop_duration_limit:
                    state['is_stopped'] = True
                    self.stopped_vehicles_ids.add(vehicle_id)
    else:
        state['stopped_frames'] = 0
        state['is_stopped'] = False
        if vehicle_id in self.stopped_vehicles_ids:
            self.stopped_vehicles_ids.remove(vehicle_id)

    state['last_position'] = current_center
```

Fonte: Autoria própria, 2025

Os veículos considerados parados são sinalizados na visualização do vídeo através de círculos azuis, permitindo a identificação visual por parte do usuário. Importante destacar que assim que o estado relatado é trocado, o círculo azul some, configurando-o como um estado persistente mas reversível.

Acrescenta-se que a lista de identificadores dos veículos parados também é transmitida via *websocket* ao *front-end* a cada segundo, além da contagem total, o que garante uma granularidade de informação valiosa para análise detalhada de incidentes.

Por fim, a funcionalidade de identificação de tráfego intensa implementa uma heurística simples, mas funcional e eficaz. Esta funcionalidade utiliza como base as

informações já adquiridas através do algoritmo mencionado anteriormente, classificando o tráfego como "intenso" quando um determinado número de veículos parados é identificado (configurável pelo arquivo de configurações, mas definido como 8 por padrão) ou quando existe uma quantidade muito grande de veículos identificados na tela.

Este critério relacionado às paradas baseia-se na observação empírica (baseada nas observações ao longo do desenvolvimento) de que, em condições normais de tráfego, é raro ter múltiplos veículos completamente parados simultaneamente na mesma câmera. Percebeu-se que quando 8 ou mais veículos estão parados ao mesmo tempo, isto geralmente indica congestionamento severo ou incidente que requer atenção.

É evidente que heurísticas mais sofisticadas poderiam considerar outras métricas, como duração média de parada, taxa de crescimento do número de veículos parados, ou padrões históricos. Entretanto, considerando o contexto acadêmico e experimental deste projeto, a abordagem atual oferece bom equilíbrio entre simplicidade e utilidade prática.

O indicador de tráfego intenso é transmitido ao *front-end* como um *booleano*, que é transformado em um *feedback* visual claro para o usuário, alternando entre "Tráfego Normal" e "Tráfego Pesado".

3.3 FRONT-END

Como dito no tópico anterior, foi planejada também a construção de formas de exibir as informações processadas na aplicação de um jeito que seja facilmente compreendido pelo usuário, em vez de utilizar terminais e telas mais primordiais.

Através dessas telas, é possível criar gráficos para visualizar métricas e estatísticas em telas simples e intuitivas para o usuário, utilizando também de técnicas de *UI/UX*, exibir detalhes sobre as câmeras, informações sobre o decorrer dos dias e prazos, e visão em tempo real das câmeras conectadas atualmente.

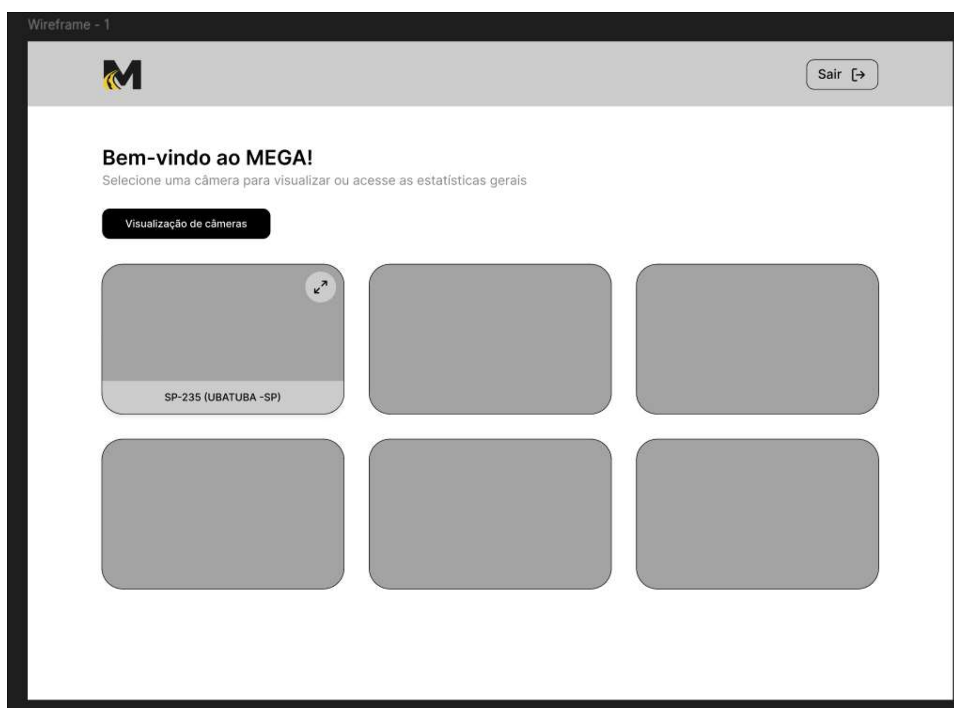
Para o desenvolvimento das telas, utiliza-se o *framework Next.JS*. Essa escolha foi tomada por ser uma tecnologia que está transformando a *internet* nos últimos anos, e tem sido inovadora em diversos sentidos, em especial, na sua

performance, reatividade, suporte a várias arquiteturas de “componentização” e desenvolvimento, técnicas de *SEO*, acessibilidade, e boas práticas gerais em tecnologias *web*. Antigamente, páginas na *web* eram completamente estáticas, porém, desde a última década, com o decorrer dos anos e do desenvolvimento de tecnologias reativas como o *React* (que o *NextJS* utiliza atualmente como parte de seu funcionamento), foi-se ficando cada vez mais fácil criar telas que podem responder às interações do usuário em tempo real (reatividade) de forma praticamente instantânea e ágil. Essa aplicação tem um *cache* extremamente avançado e fornecimento de arquivos estáticos gerados do lado do servidor, tirando trabalho do dispositivo do usuário (dispositivo cliente), que, normalmente, por conta da diferença do desempenho de *hardware* – que varia de dispositivo para dispositivo – influenciaria na velocidade do processamento.

Na questão do design e as escolhas de estilização das telas, foram utilizados conceitos de Interface Humano-computador e conceitos simples de *UI/UX*. Para facilidade no desenvolvimento, foram utilizadas as bibliotecas *TailwindCSS*, e um sistema de design já existente, chamada *HeroUI* (antigamente *NextUI*). A utilização desta biblioteca se deve ao fato dos componentes visuais fornecidos por ela já serem bem otimizados, agradáveis esteticamente, e com animações fluídas. Dessa forma, componentes simples, como botões, não precisariam ser completamente escritos e desenvolvidos do zero, permitindo agilidade na construção da aplicação, sem prejudicar a estética.

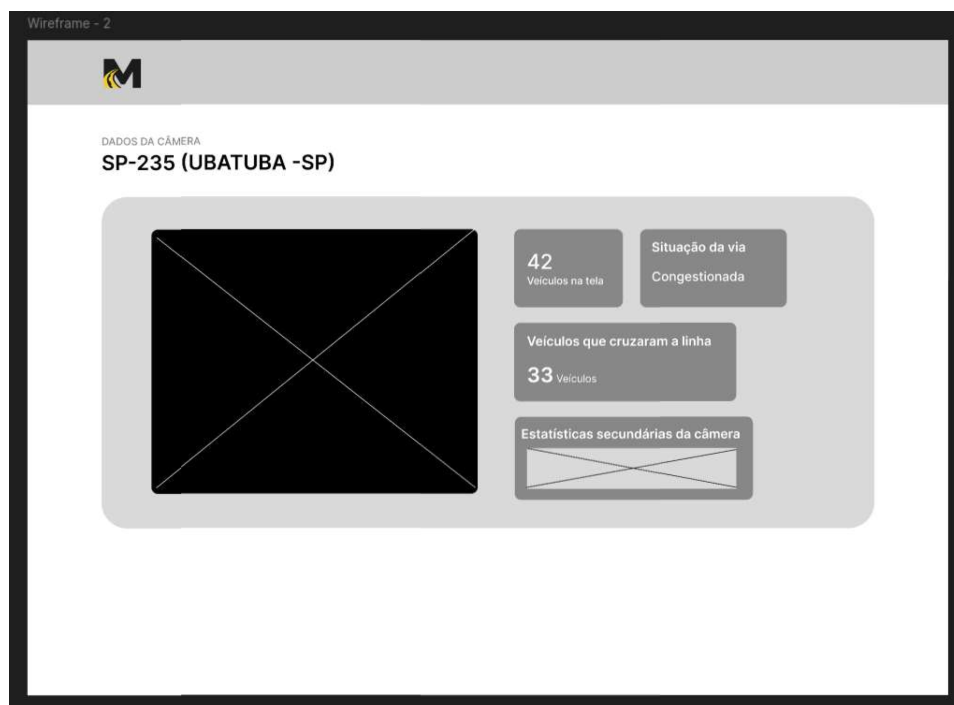
Para o planejamento da organização das telas e como cada informação seria disposta, é feito o uso do *Figma*, como ferramenta auxiliadora no desenho de cada página. A utilização desse aplicativo foi essencial para o desenvolvimento de *wireframes* e prototipação de baixo nível.

Figura 13 – Prototipagem da Tela Inicial



Fonte: Autoria própria, 2025

Figura 14 – Prototipagem da Tela de Exibição de Câmera



Fonte: Autoria própria, 2025

Já para o gerenciamento das bibliotecas, como *Tailwind* e *HeroUI* citadas anteriormente, no contexto do *front-end*, utilizamos o principal gerenciador de pacotes *Node*, o *NPM.* Foi debatido o uso de outros gerenciadores mais modernos, como *yarn*, *pnpm* ou *bun*, mas, por fim de estabilidade, robustez, e suporte, mantivemos o uso do *NPM*.

3.4 CONCLUSÃO DA METODOLOGIA

Através de muitas discussões e debates, tornou-se possível fornecer à aplicação uma arquitetura decente e bem pensada. Da infraestrutura até a visualização, a organização do código, e de todas as ferramentas utilizadas para concretizar o projeto, foi planejada com muito zelo para o desenvolvimento mais cuidadoso, otimizado, e eficiente possível.

Foram utilizadas tecnologias empregadas hoje no mercado de trabalho em diversas empresas, incluindo as maiores. Sendo assim, a definição desta arquitetura possibilitou a todos os envolvidos no projeto uma experiência aproveitável em outros cenários que podem, ou não, incluir o cenário acadêmico, bem como também o profissional.

4. REVISÃO DE LITERATURA

4.1 MOBILIDADE URBANA

Para a realização do trabalho, foram visitados vários artigos, revistas e livros virtuais que embasam o desenvolvimento dele. Ao revisar esses materiais, é possível encontrar a explanação de diversos processos que devem ser aplicados para o processamento dos vídeos das câmeras de trânsito, e identificação de possíveis riscos e pontos de melhoria e atenção nas estradas e ruas de circulação de veículos, além da relação existente entre essas tecnologias e os problemas de mobilidade urbana encontrados no Brasil, atualmente.

O tipo de solução que é objeto deste trabalho apresenta-se como algo de bastante relevância, principalmente em um país onde, de acordo com Andrade e

Galvão (2016, p.2), o adensamento populacional aumenta de maneira desordenada, faltando infraestrutura para acompanhar.

Atualmente, é possível afirmar que o Brasil ainda enfrenta uma crise de mobilidade urbana, que acontece, primeiramente, “por uma crise do transporte coletivo e uma reafirmação do modelo rodoviarista que alcança seus limites com o grande aumento de automóvel e motocicletas nas ruas” (Rodrigues, 2016, p.88). Mas não são apenas as rodovias que são afetadas por esse aumento da opção pela motorização individual, e isso pode ser notado por grande parte da classe trabalhadora brasileira, que tem de enfrentar os constantes congestionamentos diariamente em todo tipo de via pública.

Todo esse contexto problemático quanto ao transporte urbano, caracterizado como uma crise de mobilidade, influencia diretamente no bem-estar dos cidadãos, dificultando ainda a superação das desigualdades sociais (Scaringella, 2001; Rolnik e Klintowitz, 2010; Rodrigues, 2013 *apud* Rodrigues, 2016, p.86)[1]. Além disso, de acordo com Tavares et al. (2024, p.106), com todo esse crescimento da quantidade de vias asfaltadas, impulsionado pela alta demanda mencionada anteriormente, a redução de áreas verdes e emissão elevada de gás carbônico gera também uma preocupação com o meio ambiente, que não deve ser deixada de lado.

4.2 SMART CITIES E A TECNOLOGIA COMO ALTERNATIVA

Ao analisar o problema, e entendendo que a crise pode se estender de maneira permanente, tendo em vista que já é notada desde o século XX (Rodrigues, 2016, p.81), é importante buscar por alternativas para amenizá-lo. De acordo com Scaringella (2001, p. 55), “propostas alternativas de uma distribuição mais inteligente de viagens ou deslocamentos são uma forma de melhorar o trânsito sem grandes investimentos”.

Esse é um dos problemas que o conceito de “*smart city”* busca resolver, e um projeto como apresentado através desta monografia é capaz de auxiliar neste objetivo. O conceito de *“smart city”*, considera a tecnologia como um fator indispensável para o

[1] RODRIGUES, Juciano, *“Mobilidade urbana no Brasil: crise e desafios para as políticas públicas”*. Belo Horizonte: R. TCEMG, 2016. p.86.

crescimento ordenado e modernizado das cidades, segundo Galvão e Andrade (2016, p.5).

É importante destacar que, por mais que possa parecer uma abordagem "futurística", esses conceitos já vêm sendo aplicados na sociedade, mesmo que, muitas vezes, passem despercebidos pela população.

> A combinação cada vez mais eficaz das capacidades de integração entre os sensores e os sistemas computacionais têm permitido a criação e identificação de inúmeras oportunidades para o enfrentamento dos principais problemas que afetam as cidades. Sobre os transportes, por exemplo, pode-se destacar o monitoramento de tráfego e a manutenção preventiva. Já sobre a mobilidade, de maneira mais abrangente, destaca-se a possibilidade de localização e georreferenciamento, transportes alternativos e informações sobre itinerários e rotas (Galvão e Andrade, 2016, p.7).

No Brasil, já podemos destacar algumas cidades que aplicam estes conceitos, como o Rio de Janeiro. Segundo Galvão e Andrade (2016, p. 11), a cidade já foi premiada *Smart City World Award*, que reconhece iniciativas para modernização dos modelos de gestão e desenvolvimento urbano em diversas áreas, incluindo a mobilidade urbana.

Como aponta Scaringella (2001, p.59), aplicar soluções como a informática, engenharia de tráfego, a eletrônica e a tecnologia comportamental são algumas formas de começar a construir o chamado "trânsito inteligente". Além disso, a democratização da informação e a participação social são imprescindíveis nesse processo, tendo em vista que o desempenho do trânsito também depende de todos os cidadãos que estão envolvidos nele, como os pedestres e os motoristas, e precisam estar informados.

Uma questão importante a ser observada atualmente é que, muitas vezes, as tecnologias e sistemas são construídos apenas para solucionar problemas emergenciais, enquanto poderiam ser utilizados para encontrar pontos de melhoria e prevenção e desenvolver estratégias, evitando transtornos (Tavares *et al.*, 2024).

Quanto a esse tema, podemos afirmar que:

> A metodologia mais eficaz para o desenvolvimento de segurança no trânsito está na prevenção. Em primeiro lugar busca-se saber identificar riscos e, logo a seguir, fazer o gerenciamento dos mesmos riscos envolvendo o fator humano, o meio ambiente, a via pública e o veículo. É importante defender o primado da segurança em detrimento da fluidez a partir da importância que deve ser dada à preservação da vida (Scaringella, 2001, p.59).

De qualquer modo, deve-se ressaltar que a tecnologia de maneira isolada não é capaz de resolver nenhum problema. Seu papel é servir como uma ferramenta, mas é necessário que haja uma integração entre a governança, a infraestrutura e a participação da sociedade, que deve receber voz e ter influência ativa nas tomadas de decisão quanto ao desenvolvimento sustentável abordado (Tavares *et al*., 2024; Galvão e Andrade, 2016). Algumas ações podem ser destacadas no sentido complementar à tecnologia nessa missão.

> Dentre essas ações, tem-se políticas de segurança pública, educação no trânsito, campanhas de incentivo à intermodalidade (através da combinação de meios de transporte em massa e meios de locomoção ativa), melhores condições dos transportes públicos, acesso a meios de transporte sustentáveis e infraestrutura compatível. Isolada, a tecnologia não torna uma cidade mais inteligente ou sustentável (Tavares *et. al*, 2024, p.110).

4.3 INTELIGÊNCIA ARTIFICIAL

“Artificial intelligence (AI) is the study of how to make computers do things which, at the moment, people do better.” (RICH e KNIGHT, 1991, p.3). A frase dos autores e cientistas da computação norte-americanos, que diz que a inteligência artificial nada mais é do que um constante estudo que busca explicar como computadores podem fazer coisas que, até o momento, pessoas fazem melhor. Esse tipo de análise permite explorar o uso de computadores para realização de atividades que necessitam de um raciocínio lógico, análise aprofundada ou até mesmo o uso da persuasão.

4.3.1 Teste de Turing

Este famoso experimento foi criado em 1950 por Alan Turing – considerado o pai da computação – o qual consistia em uma análise experimental em que o computador passará no teste caso um interrogador humano, depois de propor uma série de perguntas por escrito, não consiga descobrir se as repostas vieram de uma pessoa ou de uma máquina (Russel e Norvig**,** 2022, p. 1).

O teste descrito teve sua grande importância na história da computação pois apontou a possibilidade de que computadores pudessem apresentar características

de inteligência similar ou superior às de seres humanos. Ainda com base na obra mencionada anteriormente, é possível observar algumas características mencionadas pelos autores como imprescindíveis para que se possa afirmar que uma máquina possui inteligência:

- Processamento de Linguagem Natural: para comunicação em linguagem humana.
- Raciocínio Automatizado: para responder perguntas e elaborar conclusões.
- Aprendizado de Máquina: para se adaptar a circunstâncias e contornar padrões.
- Visão Computacional: para reconhecimento e percepção do mundo.
- Robótica: para manipulação de objetos e movimento.

Todos estes conceitos são bases para a classificação de máquinas que possuem inteligência, o que conclui o entendimento de que um computador pode apresentar uma programação capaz de concedê-lo com uma inteligência similar ou superior à inteligência humana.

4.3.2 Aprendizado de máquina

Em continuidade aos tópicos apresentados, o aprendizado de máquina (*machine learning)* se trata de um conceito essencial na área de IA e pode ser explicado:

> O aprendizado de máquina é uma aplicação da inteligência artificial que concede ao sistema de IA a capacidade de aprender automaticamente com o ambiente e aplicar essas lições para tomar decisões melhores. Há uma variedade de algoritmos usados pelo aprendizado de máquina para, de forma iterativa, aprender, descrever e melhorar os dados, identificar padrões e, depois, agir com base nesses padrões (Hariom, Sahil, Brad, 2024, p.8).

Ao considerar existência de um sistema em que o próprio algoritmo se adapte e consiga realizar uma análise – com o objetivo de identificar padrões – é possível afirmar que uma máquina conseguirá avaliar um dado contexto e identificar fatos relevantes, que podem ser solucionados com respostas treinadas e direcionadas para determinado fato.

4.4 VISÃO COMPUTACIONAL

Diante do que fora abordado até o devido momento, deve ser aprofundado também o importante conceito de visão computacional, o qual faz referência a um novo campo de pesquisa, surgindo a partir de tópicos como a ciência da cognição e a já apresentada, inteligência artificial.

> No ano de 1982, Ballard e Brown, na obra Computer Vision, definiram Visão Computacional como a ciência que estuda e desenvolve tecnologias que permitem que máquinas enxerguem e extraiam características do meio, através de imagens capturadas por diferentes tipos de sensores e dispositivos. Essas informações extraídas permitem reconhecer, manipular e processar dados sobre os objetos que compõem a imagem capturada (Barelli, 2018, p.1).

Conforme explanado pelos autores, a visão computacional nada mais é do que um conceito que pode ser observado em várias máquinas na atualidade, permitindo uma exploração de dados e fenômenos abstraídos do mundo real, diretamente para o ambiente virtual.

Com isso, para esclarecer os princípios deste trabalho, deverá ser explicado e exemplificado como algumas ferramentas e sistemas computacionais poderão abordar o tema, permitindo que computadores apresentem um comportamento típico de inteligências artificiais, unidos a algoritmos de aprendizado de máquina e treinamento de modelos, os quais permitirão análises de padrões em imagens e vídeos de monitoramento e controle dos meios rodoviários.

4.4.1 Processamento de imagens

Ao se falar sobre processamento de imagens, é possível apresentar uma tecnologia que permite receber uma imagem como dado de entrada, transformando-a em algo novo, com características distintas daquilo que fora anteriormente. Essa exploração permite aplicar filtros e camadas que valorizam informações antes não perceptíveis naquele contexto.

> Dessa forma, o que chamamos neste livro de processamento digital de imagens envolve processos cujas entradas e saídas são imagens e, além disso, envolve processos de extração de atributos de imagens até — e

> inclusive — o reconhecimento de objetos individuais (Gonzalez e Woods, 2009, n.p.).

O processamento de imagens permite, por exemplo, realizar o isolamento de uma informação contida em uma imagem:

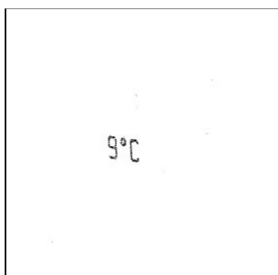
Figura 15 – Display de temperatura – Imagem original

Fonte: *Wikipedia, 2025*

Figura 16 – Display de temperatura – Componente verde

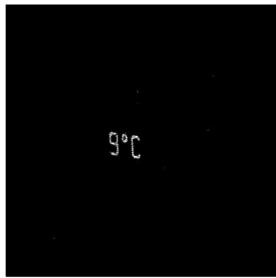
Fonte: *Wikipedia, 2025*

Figura 17 – Display de temperatura – Threshold aplicado



Fonte: *Wikipedia, 2025*

Figura 18 – Display de temperatura – Threshold invertido



Fonte: *Wikipedia, 2025*

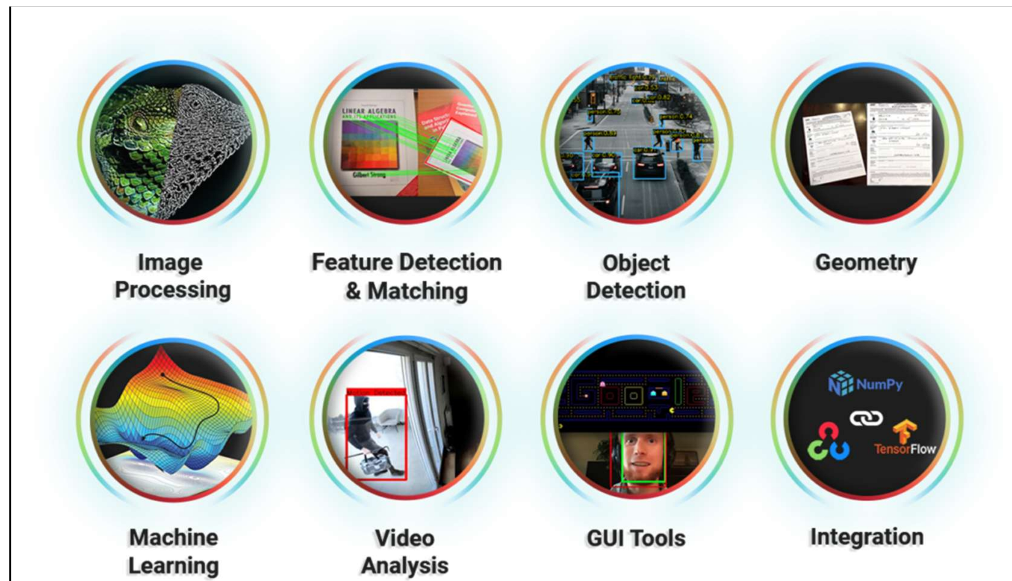
Ao analisar o resultado apresentado, é possível verificar que o processamento de imagens computacionais permite pelo sistema a realização de basicamente todos os tipos de transformações e análises de um determinado conjunto de dados, por exemplo, um *datase*t (conjunto de dados organizados) de fotos e vídeos do monitoramento de trânsito e veículos de uma cidade.

4.4.1.1 OpenCV

Para aplicar os conceitos citados, é de suma importância que seja apresentado o *OpenCV*, ferramenta da linguagem de programação *Python*, o qual permite realizar as devidas operações mencionadas. “O *OpenCV* implementa uma variedade de ferramentas de interpretação de imagens, indo desde operações simples como um filtro de ruído, até operações complexas, tais como a análise de movimentos, reconhecimento de padrões e reconstrução em 3D.” (MARENGONI e STRINGHINI, 2009, p. 126-127).

Com esta ferramenta, pode-se criar sistemas que permitem a análise de imagens e vídeos, alterando características visuais, inserindo figuras e valores para as matrizes estudadas.

Figura 19 – Aplicações para o OpenCV



Fonte: *OpenCV Blog*, 2025

Ao observar a imagem apresentada, é possível identificar aplicações como: processamento de imagens, detecção de funcionalidades e padrões, detecção de objetos, análise de geometria, *machine learning,* análise de vídeo, desenvolvimento de ferramentas da interface gráfica do usuário, além de trabalhar com integrações de outros sistemas.

4.4.1.2 YOLO

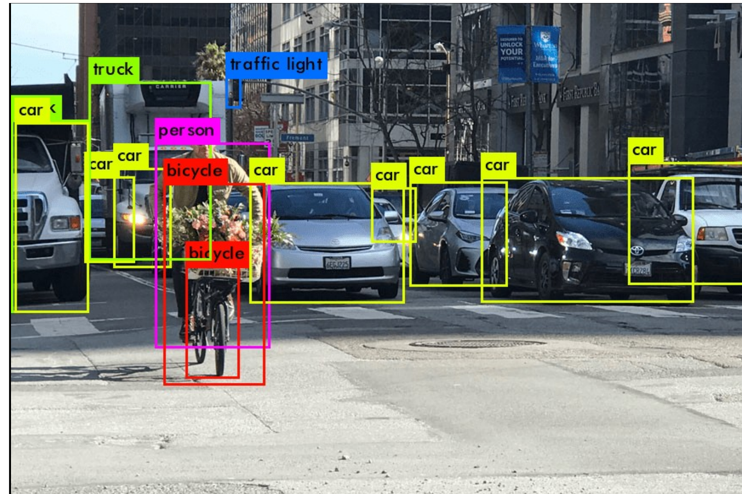
Ainda se tratando de processamento de imagens computacionais, é interessante aliar o poder de manipulação de dados e imagens do *OpenCV*, junto ao *YOLO*:

> A técnica de detecção de objetos "*You Look Only Once*" (*YOLO*) é uma família de modelos de aprendizado de máquina profundo projetados para detecção rápida. Em estudos comparativos, a *YOLOv2* e a *YOLOv3* demonstraram resultados inferiores a rede *MaskR-CNN*. Por outro lado, estes modelos demonstram capacidade de detectar, segmentar e rastrear com sucesso os aglomerados de uva em pomares, fornecendo dados valiosos para aplicações agrícolas, como previsão de rendimento e colheita automatizada (Sá et al**,** 2024, p. 116).

O *YOLO* nada mais é, do que um modelo já preparado ou treinado pelo usuário, que possibilitará a identificação de padrões em imagens e vídeos, além de outras operações computacionais, o que permite que qualquer algoritmo acoplado a ele

execute tarefas de alta complexidade com baixo custo de tempo para desenvolvimento e baixo custo de tempo para o treinamento do modelo.

Figura 20 – Exemplo de uso do YOLO para reconhecimento de entidades



Fonte: VisionPlatform.ai, 2025

4.5 DESENVOLVIMENTO DA APLICAÇÃO

Para a construção da aplicação que permitirá conectar, visualizar, analisar e gerenciar câmeras de monitoramento, será necessário explorar diversos conceitos técnicos da área da computação, tornando essencial garantir a devida contextualização destes.

4.5.1 Ferramentas para a interface (front-end)

Para a construção de uma ferramenta de acompanhamento dos dados e indicadores úteis referentes às vias públicas, a criação de uma interface amigável para o usuário é indispensável. Para o desenvolvimento dessa interface, será utilizado o *framework Next.Js*, um *framework* do *React*, que permite a construção de aplicações *web full-stack*. Através dele, é possível utilizar componentes *React* combinados com outras funcionalidades do próprio *framework*, além de otimizações, abstrações e configurações automáticas que são requisitadas pelo *React* (Next.Js, 2025, p.1).

Ao partir para uma abordagem mais detalhada e exploratória, percebe-se que o *Next.js* tem algumas vantagens bastante atrativas, que justificam sua utilização para o desenvolvimento deste trabalho. De acordo com, Jartarghar *et. al* (2022, p.28), podemos destacar entre essas vantagens: a técnica de separação automática do

código, onde o *“bundle”* da aplicação é dividido em partes menores, reduzindo o tempo de carregamento inicial; e o *“Lazy Loading*”, que permite importações dinâmicas, podendo realizá-las apenas quando os componentes em questão forem utilizados.

Os fatores abordados, portanto, ajudam a aumentar o desempenho da aplicação, reduzir a quantidade de tempo e memória necessária para o processamento e otimizar os recursos. Isso se apresenta como algo crucial no caso de uma aplicação proporção e complexidade consideráveis, principalmente por envolver processamento de imagem e vídeo. Além disso, existem diversas bibliotecas compatíveis com o *Next.Js* que podem acelerar o desenvolvimento do *front-end* da aplicação, como o *Hero UI* e *ShadCN*, que oferecem componentes pré-montados e facilita a criação de temas e outros estilos.

4.5.2 Desenvolvimento do sistema (*back-end)*

“*Back-End* é uma parte da aplicação que tem duas responsabilidades principais: executar lógicas mais complexas e armazenar os dados da aplicação” (SANTANA, 2024, n.p.).

Como mencionado ao início do tópico, para falar sobre desenvolvimento de *softwares*, é preciso destacar as tecnologias utilizadas, e com isso, apresentar uma das principais áreas responsáveis pela sustentação de qualquer aplicação, isto é, o *back-end.* Pode ser incluído nesta categoria todo tipo de desenvolvimento que envolva a lógica de programação, armazenamento e manipulação de dados, além da organização dessas informações para levá-las à interface gráfica do usuário (*GUI*).

A exemplo dos conceitos importante a serem mencionados neste projeto, é válido elencar:

- Uso do *Python* como linguagem de programação para regras de negócio e processamento das imagens.
- Transformação e disponibilização das informações via *API.*
- Gestão da estrutura *back-end* por meio da *conteinerização* dos componentes.

4.5.2.1 Python

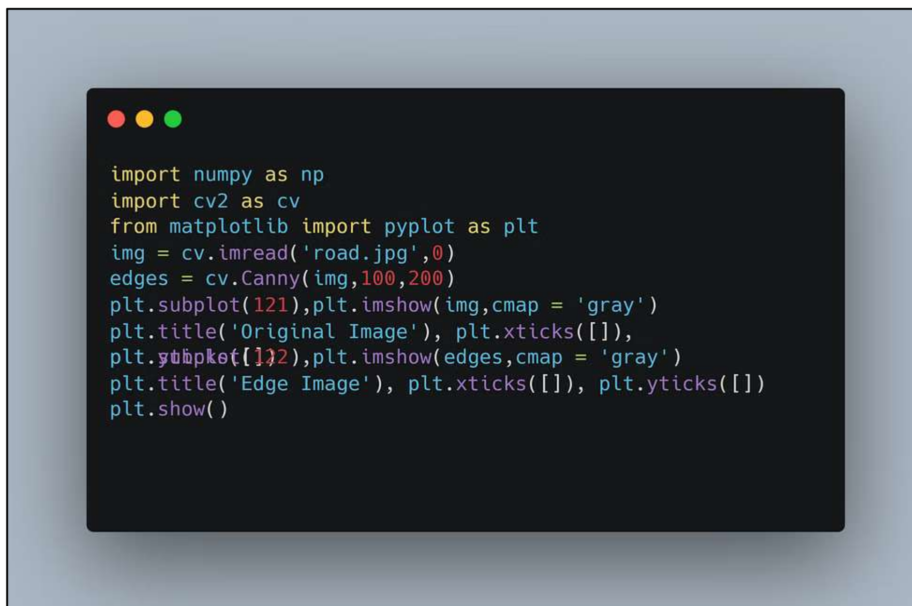
Conforme evidenciado no tópico anterior, quando se fala sobre programação, mais especificamente sobre linguagens de programação, talvez *Python* seja a LP mais

falada nesse momento, isso acontece principalmente devido ao alto número de aplicações e usos para ela, além da rápida curva de aprendizado.

> *Python é uma linguagem de programação que vem sendo empregada na construção de soluções para os mais diversos fins — educacionais, comerciais e científicos — e plataformas — web, desktop e, mais recentemente, móvel. É uma linguagem de fácil aprendizado, expressiva, concisa e muito produtiva; por isso, sua adoção tem crescido bastante nos últimos anos pelos mais variados perfis de profissionais no meio científico e acadêmico, tanto para desenvolvimento de ferramentas quanto para ensino de algoritmos e introdução à programação (*Cruz**,** 2015, n.p.).

A exemplo do contexto de visão computacional e processamento de imagens, ambos abordados anteriormente, é possível definir a responsabilidade por esses processos computacionais ao *Python*, que descreverá os comportamentos do sistema, cálculos e considerações durante o processamento, sendo assim, a ferramenta principal para que os desenvolvedores possam transformar suas ideias em realidade.

Figura 21 – Exemplo de código fonte *Python* com uso do *OpenCV*



Fonte: *ProjectPro*, 2025

4.5.2.1.1 Flask

Em continuidade ao que foi apresentado anteriormente, cabe falar sobre o *Flask*, que nada mais é do que uma micro-biblioteca de *framework web Python* que pode ser utilizada para desenvolvimento de aplicações *web*, incluindo *APIs REST* (Martins, 2023, n.p.).

Pode-se resumir que o uso desta ferramenta será exclusivamente para disponibilizar os principais dados da aplicação, como métricas e análises gerais das imagens processadas pela aplicação, diretamente para uma camada intermediária que disponibilizará esses dados para o que estará disponível aos olhos do usuário final. Esta interface visual é chamada comumente como *front-end* e será elaborada por meio dos dados disponibilizados pela *API*.

> API significa Application Programming Interface (Interface de Programação de Aplicação). No contexto de APIs, a palavra Aplicação refere-se a qualquer software com uma função distinta. A interface pode ser pensada como um contrato de serviço entre duas aplicações. Esse contrato define como as duas se comunicam usando solicitações e respostas (AMAZON WEB SERVICES, s.d., n.p.).

Concluindo, o uso do *Flask* para criação de *APIs* se enquadra como uma ferramenta poderosíssima que permitirá a comunicação entre o *back-end* e *front-end*.

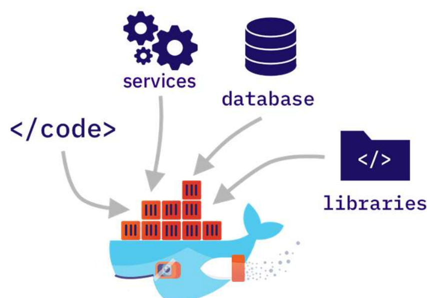
4.5.3 Docker

Para concluir a estrutura *back-end* e executar todas as camadas apresentadas até o momento, como o *front-end*, poderá ser utilizado o *Docke*r, que se trata de uma ferramenta ideal para permitir a sustentação dos ambientes de um sistema, sem necessidade de um servidor exclusivo para isto. Essa realidade existe pois o *Docker* permite a *conteinerização* – isolamento de cada estrutura – e execução do sistema sem tornar-se necessário custos com servidores, pois tudo irá ser executado na própria máquina do usuário.

> Imagine que o servidor é um navio cargueiro e que cada container leva várias mercadorias. *Docker* é uma ferramenta para criar e manter containers, ou seja, ele é responsável por armazenar vários serviços de forma isolada do *SO host*, como: *web server*, banco de dados, aplicação, *memcached* etc (Romero, 2015, n.p.).

O uso desta ferramenta reunirá todos os desenvolvimentos realizados até o momento e permitirá a consolidação das tecnologias utilizadas, confirmando assim, a sua grande importância neste projeto.

Figura 22 – Ilustração do Docker



Fonte: Eliel Lança, 2025

5 RESULTADOS E DISCUSSÕES

5.1 IMPLEMENTAÇÃO DO SISTEMA

O presente trabalho implementou de maneira bem-sucedida e conforme planejado o sistema de monitoramento de tráfego em tempo real (MEGA), que se mostrou capaz de identificar, classificar e quantificar veículos a partir de fluxos de vídeo, calcular métricas de tráfego e transmitir informações continuamente aos usuários.

A aplicação foi implementada em *Python* e estruturada a partir do *framework Flask* para o servidor *web* e do *Socket.IO* para a comunicação em tempo real entre servidor e interface de usuário. O sistema suporta tanto transmissões ao vivo de câmeras públicas, fornecidas no formato *M3U8* (*HLS*), quanto vídeos gravados previamente, possibilitando testes controlados e validação em cenários reais.

A Figura 24 a seguir apresenta a interface do sistema, implementada em *Next.Js,* que revela a câmera processada e as devidas estatísticas extraídas do processamento de tal câmera, exibindo as caixas delimitadoras de cada veículo, *IDs* de rastreamento, marcadores de eventos e o painel de estatísticas em tempo real.

Figura 23 – Interface de visualização da câmera processada

Fonte: Autoria própria, 2025

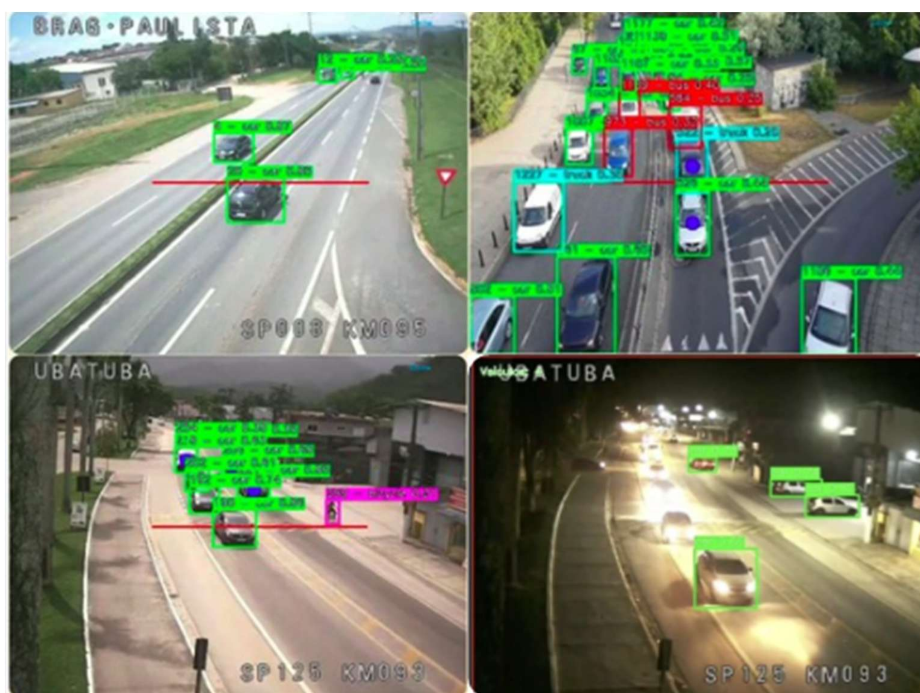
Constata-se também que a integração entre o *back-end* (*Python/Flask*) e o *front-end* (*Next.js/React*) através de *API REST* e *websocket* funcionou adequadamente, permitindo a transmissão do vídeo processado de maneira contínua via *MJPEG* e atualizando as estatísticas segundo a segundo sem perda de sincronização observável.

5.2 VALIDAÇÃO DAS FUNCIONALIDADES

5.2.1 Detecção e rastreamento de veículos

A funcionalidade central do sistema (detecção e rastreamento de veículos utilizando o *YOLO*) apresentou um bom desempenho nos cenários testados. O modelo foi capaz de identificar de maneira consistente as cinco classes de veículos delimitadas nas configurações em diferentes condições de iluminação (funcionando melhor no período diurno), ângulos de câmera e densidades de tráfego. A Figura 25 apresenta o sucesso na detecção em diferentes cenários:

Figura 24 – Grid demonstrando diferentes resultados



Fonte: Autoria própria, 2025

O algoritmo de rastreamento multi-objeto também foi capaz de manter identidade dos veículos detectados ao longo de diferentes *frames* consecutivos o que é fundamental para a detecção de paradas, por exemplo.

Além disso, a diferenciação visual por cores para cada veículo funcionou de maneira adequada, facilitando a identificação rápida e intuitiva por parte do usuário, enquanto que os rótulos contendo ID de rastreamento, classe e nível de confiança forneceram informações detalhadas para análise mais aprofundada quando necessário.

5.2.2 Contagem por linha virtual

A funcionalidade de contagem de travessias, estabelecida por meio de uma linha traçada horizontalmente, possível de ser visualizada pelo vídeo processado da câmera, apresentou-se operacional e consideravelmente precisa em testes qualitativos. As coordenadas definidas, que podem ser ajustada através das linhas de código, funcionaram adequadamente para as câmeras testadas.

De maneira geral, a maior parte dos veículos que atravessam a linha são detectados de maneira bem-sucedida, com exceção de alguns raros casos onde os veículos visualizados na câmera acabam se dispersando do rastreamento no último instante anterior ao cruzamento. Ademais, o algoritmo baseado em geometria computacional não apresentou detecção de falsos positivos nas travessias observáveis.

Mesmo em casos de tráfego intenso, o algoritmo apresentou bons resultados na identificação das múltiplas travessias ocorrendo simultaneamente, contabilizando cada travessia individualmente, sem duplicá-las ou omitir tais detecções de forma aparente. Essas travessias também são evidente e efetivamente sinalizadas ao usuário através do círculo amarelo que destaca os veículos, permitindo identificação imediata das travessias contabilizadas.

Uma limitação identificada é que, como já mencionado anteriormente no presente trabalho, o sistema conta travessias bidirecionais indistintamente. Para aplicações onde direção de movimento é relevante (por exemplo, monitorar apenas entrada em uma região específica), seria necessário implementar lógica adicional de diferenciação direcional.

5.2.3 Identificação de veículos parados

A identificação de veículos parados, implementada através da avaliação temporal do movimento do veículo rastreado em relação ao tamanho de sua caixa delimitadora, demonstrou eficácia na identificação de imobilizações prolongadas.

Os parâmetros (definidos para mínimo de duração de parada, *threshold* de movimento e quantidade mínima de *frames* em que o veículo ficou parado), quando bem configurados, apresentam-se equilibrados e adequadamente funcionais, reduzindo significativamente falsos positivos que podem ser causados por oscilações naturais nas posições de identificação e principalmente na taxa de movimento tolerada (o *threshold*).

A transmissão desses dados via *websocket*, juntamente à lista de *IDs* dos veículos parados, permitiu que o *front-end* exibisse tanto a contagem total quanto os

identificadores específicos, informação bastante útil para análise detalhada de incidentes, por exemplo.

Em testes com vídeos gravados, onde o tráfego era mais intenso, propositalmente selecionados para realizar experimentos de estresse na aplicação, essa detecção também se apresentou eficiente, mantendo a classificação enquanto os veículos permaneciam estacionários e removendo-a assim que retomavam movimento.

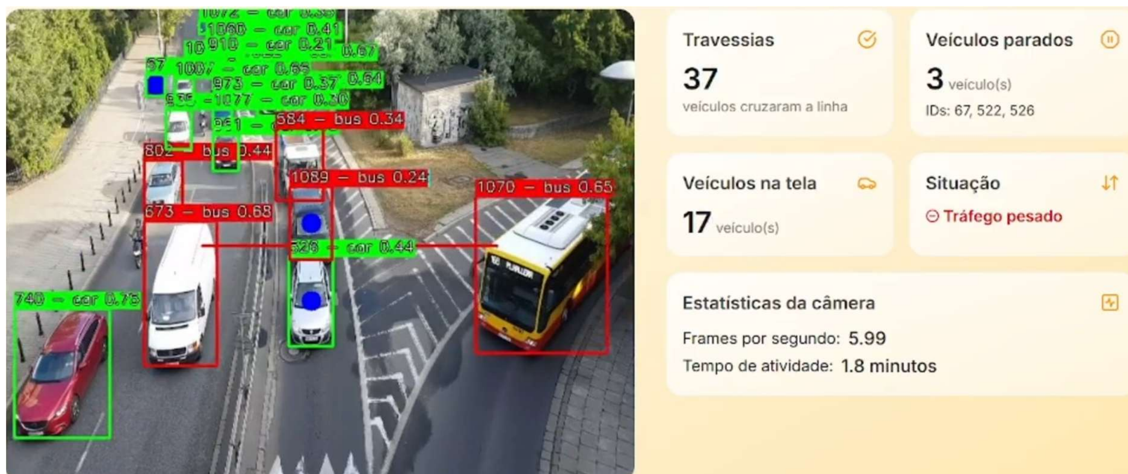
Destaca-se novamente que a configuração do limite de movimento tolerado para a classificação de um veículo ser considerado estacionário ou não merece bastante atenção por parte do responsável pelo *setup* do sistema, tendo em vista que, a depender do ângulo da câmera, o algoritmo pode alegar veículos em movimento como parados. A sensibilidade deste parâmetro ao ângulo da câmera decorre de um fenômeno de perspectiva. Câmeras posicionadas em ângulos laterais (visão oblíqua da via), por exemplo, acabam por capturar maior deslocamento aparente de veículos em movimento longitudinal do que câmeras posicionadas perpendicularmente (visão frontal ou traseira).

A implementação atual do sistema MEGA utiliza um valor único global aplicado a todas as câmeras. Entretanto, pensando em uma oportunidade de desenvolvimento mais avançado e personalizado, a calibração de um *threshold* específico por câmera, considerando seu ângulo de visão, distância focal e posicionamento em relação à via, seria ideal para evitar uma ocorrência maior de falsos positivos.

5.2.4 Detecção de tráfego intenso

A heurística de detecção de tráfego intenso cumpriu seu propósito de identificar situações e sinalizar situações anormais de trânsito, principalmente congestionamentos.

Figura 25 – Detecção de tráfego intenso em funcionamento



Fonte: Autoria própria, 2025

Em horários de pico, quando múltiplos veículos permaneciam imóveis (a partir de 8 veículos) por períodos prolongados ou uma quantidade muito grande de veículos era rastreada (definido como 14, por padrão), o sistema sinalizou corretamente o estado de tráfego intenso. Por outro lado, em horários de fluxo contínuo e mais livre, quando as vias se apresentam mais limpas, o estado mantinha-se constantemente sinalizando tráfego normal.

A escolha do *threshold* de 8 veículos parados mostrou-se razoável para as câmeras testadas, embora seja importante notar que este valor foi definido empiricamente e poderia requerer ajuste para câmeras com diferentes campos de visão ou posicionamentos, o que pode ser feito facilmente através dos arquivos de configuração.

Importante observar, de qualquer modo, que o algoritmo de detecção de parada, mencionado anteriormente, contabiliza todos os veículos estacionários identificados no campo de visão da câmera, que pode abranger acostamentos ou estacionamentos, por exemplo. Portanto, em câmeras de visão ampla, estes veículos estacionados além da via podem influenciar a métrica. Uma implementação futura possível e adequada seria a implementação de uma região de interesse (*ROI*), limitando a área que deve ser considerada relevante para o processamento e identificação dessas condições de tráfego, aprimorando a precisão da aplicação.

5.3 DESEMPENHO E APRESENTAÇÃO DO SISTEMA

Embora não tenham sido armazenadas medições sistemáticas utilizando instrumentação e ferramentas específicas, observações qualitativas e acompanhadas pelas estatísticas de saúde implementadas no sistema permitiram caracterizar o desempenho operacional do sistema.

O vídeo processado transmitido aos usuários pela exibição no *front-end* apresentou uma reprodução fluida, sem muitos congelamentos ou *stuttering*, com a implementação da *bufferização* cumprindo seu papel de amenizar as variações de latência.

Sobre as estatísticas enviadas ao *front-end* via *websocket* mantiveram atualização constante a cada segundo, apresentando boa sincronização com o processamento e exibição do vídeo. Em ambiente de desenvolvimento, não foram observados atrasos entre os eventos no vídeo e a reflexão nos contadores da interface, enquanto em ambiente de produção foi possível observar atrasos mínimos, quase insignificantes para a identificação desses eventos.

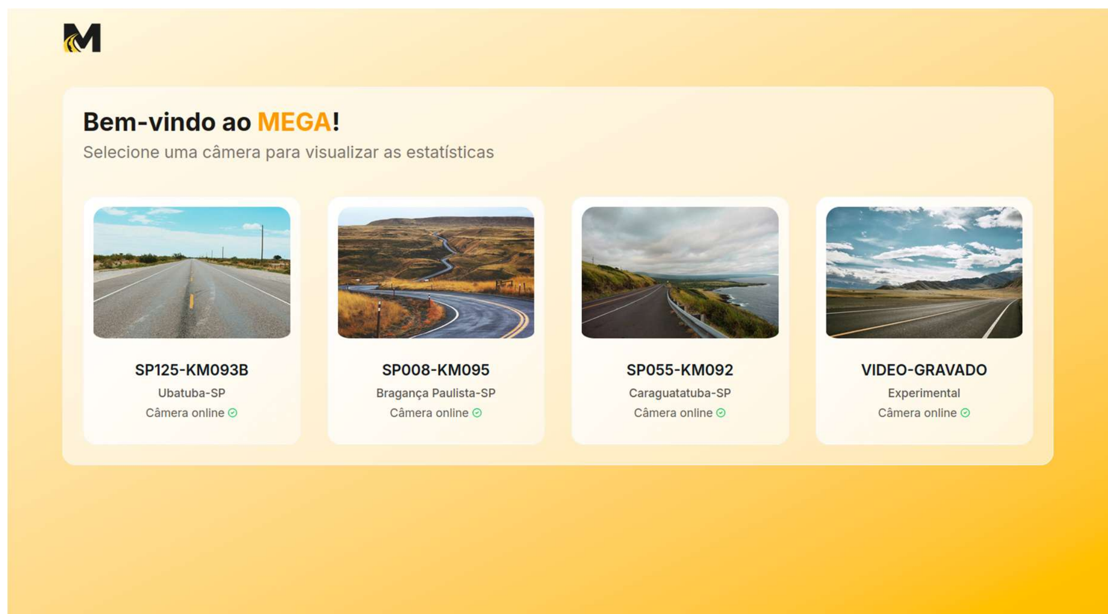
Considerando o contexto acadêmico e de projeto em fase inicial ou de experimentação, a latência total entre a captura do vídeo e exibição dele processado, resultante do *buffer delay* de 2 segundos implementado mostrou-se aceitável para o propósito de monitoramento não-crítico, considerando as estatísticas geradas e o propósito a ser alcançado. Em casos mais críticos ou para aprimoramento futuro, o *delay* poderia ser reconfigurado e testado buscando otimizar ainda mais a rapidez das respostas.

Durante testes de maior duração, o sistema manteve-se operacional no processamento e extração das métricas e dados das câmeras, sem necessidade de reinicialização manual. Além disso, em casos onde as câmeras apresentaram interrupções temporárias, a reconexão automática funcionou de maneira satisfatória.

É importante ressaltar que se observou um uso significativo da *CPU* durante o processamento, com taxas mais elevadas de *FPS* quando as condições de *hardware* permitiam aceleração por *GPU*, mas sem comprometer o funcionamento do sistema operacional ou impossibilitar a execução de outras tarefas, mesmo com *hardware* considerado básico em vista das atividades executadas.

Ao tratar da apresentação do sistema e dos dados extraídos, a interface *web* desenvolvida em *Next.Js,* apesar de simples apresentou-se funcional e de acordo com as heurísticas padrões de experiência do usuário, garantindo facilidade e intuitividade no uso.

Figura 26 – Página inicial do sistema MEGA



Fonte: Autoria própria, 2025

A página inicial permite seleção entre as câmeras disponíveis através de *cards* (apresentados com imagens genéricas para efeito do projeto acadêmico), código da câmera e localização. Ao clicar em uma câmera, o sistema inicializa o processamento da câmera selecionada através de uma chamada ao devido *endpoint* e redireciona automaticamente para a página de visualização.

A página de monitoramento e exibição da câmera processada exibe o *streaming* do vídeo, um painel lateral com todas as estatísticas (total de veículos, travessias, veículos parados, indicador de tráfego) e informações técnicas, como a taxa de *FPS* atual e o tempo de duração da sessão.

Figura 27 – Página de monitoramento da câmera

Fonte: Autoria própria, 2025

A escolha do *framework Next.Js* para o desenvolvimento do *front-end* contribuiu de maneira significativa para a produtividade durante o projeto, permitindo foco nas lógicas de negócio em vez de configurações infraestruturais. Funcionalidades nativas do *framework* eliminaram complexidade técnica desnecessária, enquanto a integração com *React* e bibliotecas especializadas, como *socket.io-client* e *react-hot-toast,* acelerou a implementação de comunicação em tempo real e *feedback* ao usuário. Além disso, o suporte nativo ao *TypeScript* auxiliou na verificação de tipos e preveniu erros durante desenvolvimento e a combinação com *TailwindCSS* e *HeroUI* para os elementos visuais possibilitou uma construção rápida da interface, eliminando a necessidade de uma estilização completamente manual.

5.4 DISCUSSÃO DOS RESULTADOS

Os resultados obtidos com os estudos e o desenvolvimento do sistema MEGA permitem estabelecer uma análise crítica tanto em relação aos objetivos propostos no início do trabalho quanto às questões teóricas abordadas no referencial bibliográfico sobre mobilidade urbana e cidades inteligentes, além das tecnologias utilizadas e também mencionadas no estudo.

Quanto ao primeiro objetivo específico, de utilizar câmeras em pontos estratégicos para captura e processamento de imagens em tempo real, o sistema demonstrou plena capacidade técnica de capturar e processar os *streams* de vídeo, utilizando câmeras públicas estrategicamente posicionadas pelo DER-SP, aproveitando a infraestrutura existente para democratizar o acesso a tecnologias de monitoramento, conforme defendido pelos autores Galvão e Andrade (2016) ao discutirem a viabilidade de *smart cities.* Além disso, a estrutura utilizada, ainda atualmente funcionando localmente, demonstra que sistemas eficazes de monitoramento não dependem necessariamente de uma infraestrutura computacional extremamente sofisticada, por mais que este possa ser um aspecto relevante para aplicabilidade em municípios de médio porte, por exemplo.

Partindo para o segundo objetivo, que se configura como a integração com algoritmos de inteligência artificial para identificação e classificação de veículos, este foi facilmente alcançado através da implementação bem-sucedida do modelo de detecção *YOLOV11n*. Os resultados evidenciaram que as redes neurais convolucionais atuais já oferecem precisão suficiente para o tipo de trabalho executado no monitoramento de trânsito e tráfego, mesmo através de suas variantes mais leves para *hardware* convencional. Este aspecto se conecta diretamente com as discussões de Sá et al. (2024) sobre a evolução da família YOLO e sua aplicabilidade em detecção de objetos em tempo real.

Ainda nesta linha, a capacidade do sistema de reconhecer padrões como os congestionamentos, os veículos parados e travessias coloca em prática o conceito de “trânsito inteligente” idealizado por Scaringella (2001), onde a informática e a tecnologia comportamental, além de outros aspectos, convergem para uma melhor gestão do tráfego.

Relativamente ao terceiro objetivo, que abrange a criação do painel interativo *web* com as condições de tráfego, o sistema entregou a interface de maneira funcional, atendendo plenamente ao propósito de facilitar a tomada de providências por parte dos operadores responsáveis. As estatísticas apresentadas pela interface são apropriadas para uso em estações de trabalho e coordenação de respostas a incidentes, permitindo que providências sejam tomadas com base em dados objetivos e em tempo real.

O quarto objetivo específico, que visava possibilitar fiscalização e observação de rodovias através das câmeras, também foi atingido, com a fiscalização viabilizada por natureza estatística e operacional, visando o uso preventivo e estratégico da tecnologia, defendido por Tavares et al. (2024). O sistema MEGA, se posiciona nesse espectro fornecendo informações contínuas que podem subsidiar decisões e observações de médio prazo (como ajustes de sinalização e identificação de necessidades infraestruturais), e não apenas resposta a incidentes imediatos.

Ao analisar criticamente o conjunto de resultados frente ao problema de mobilidade urbana discutido ao longo do trabalho, percebe-se que o sistema oferece contribuição, mas não soluciona o problema sozinho. Isso porque, assim como Rodrigues (2016) argumenta, a crise de mobilidade urbana estabelecida no Brasil tem raízes estruturais bem profundas, raízes tais que apenas as tecnologias de monitoramento não são capazes de resolver por si só. O sistema MEGA não resolve congestionamentos, por exemplo, ele apenas os torna visíveis e quantificáveis. A tecnologia isolada não constrói as cidades inteligentes, conforme reforçam as ideias de Galvão e Andrade (2016), é necessário integração com governança participativa, políticas públicas consistentes e infraestrutura adequada.

A arquitetura de processamento desenvolvida e documentada para o projeto, considerando aspectos como o sistema de “*bufferização”* e o gerenciamento *multi-thread,* representam não só contribuição técnica para o domínio específico de monitoramento de tráfego, mas também constitui material de referência para trabalhos futuros em áreas correlatas.

Além disso, as limitações identificadas, como a calibração manual por câmera para melhor extração de certos dados, não invalidam a contribuição, mas delineiam claramente as diferenças entre a prova de conceito acadêmica e um sistema de produção robusto. Este trabalho representa a etapa inicial de uma trajetória evolutiva ao considerar um sistema de trânsito inteligente, que costuma evoluir incrementalmente.

Finalmente, deve-se reconhecer que o maior valor do trabalho reside não apenas no sistema funcional produzido, mas também no processo formativo proporcionado por tal produção. O grupo formado para o desenvolvimento do presente trabalho enfrentou desafios técnicos reais, como sincronização de *threads*,

gerenciamento de *buffers*, implementação de *websocket*, integração de tecnologias heterogêneas, refinamento de detecções em vídeo, entre outros. Tudo isso permitiu o desenvolvimento de competências diretamente aplicáveis em contextos além do acadêmico, contextos profissionais. O aprendizado sobre as limitações do *Python* para processamento paralelo, o equilíbrio entre velocidade e precisão dos diferentes modelos de detecção, a integração entre o *front-end* e o *back-end* desenvolvidos em diferentes tecnologias, as heurísticas de desenvolvimento, entre tantos outros conhecimentos adquiridos e aprimorados constituem capital intelectual tão relevante para os membros quanto o *software* produzido em si. Esta dimensão pedagógica, mesmo que menos tangível, alinha-se com um dos principais objetivos de um trabalho de conclusão de curso: preparar os alunos como profissionais tecnicamente competentes e capazes de desenvolver não só sistemas *full-stack* funcionais, mas também o pensamento crítico sobre escolhas arquiteturais e suas implicações.

6 CONSIDERAÇÕES FINAIS

O trabalho desenvolvido demonstrou a viabilidade da utilização de técnicas de visão computacional e aprendizado profundo para o monitoramento automatizado de tráfego urbano. A integração entre um pipeline de inferência (baseado em modelos da família *YOLO*), um mecanismo de *buffer* otimizado e uma interface *web* com comunicação em tempo real permitiu a construção de uma solução funcional, escalável e robusta. A escolha do modelo de detecção *YOLOv11,* em detrimento a outros modelos disponíveis, foi baseada em estudos e testes previamente realizados, o que resultou em ganhos de precisão e de eficiência computacional, com efeitos diretos na qualidade das medições de contagem e na redução de falsos positivos.

As limitações identificadas neste trabalho incluem a sensibilidade do sistema a condições extremas de iluminação, a dependência do desempenho de *hardware* para alcançar maiores taxas de *FPS* e a vulnerabilidade inicial a problemas de sincronização de *threads*. Entretanto, as soluções implementadas — como o

mecanismo de inicialização automática, os quadros de carregamento/erro e o *Mega Buffer* — mitigaram grande parte dessas limitações, consolidando um ambiente de execução mais estável e reprodutível.

Como trabalhos futuros, recomenda-se a ampliação do sistema para suportar múltiplas câmeras simultâneas de maneira distribuída, a integração com um banco de dados histórico para análises temporais e preditivas, a implementação de técnicas de rastreamento multi-objeto mais avançadas para reduzir duplicidades de contagem e a avaliação sistemática de desempenho em diferentes variantes de *hardware* (incluindo comparativos entre *CPU* e *GPU*). Além disso, sugere-se a realização de testes em maior escala e a validação com conjuntos de dados anotados para obter métricas experimentais mais precisas e comparáveis com *benchmarks* públicos.

Por fim, este trabalho comprova que soluções de código aberto, quando combinadas com práticas sólidas de engenharia de *software*, podem resultar em ferramentas viáveis para apoio à gestão de mobilidade urbana e ao desenvolvimento de cidades mais inteligentes, fornecendo dados úteis para planejamento e tomada de decisão.

REFERÊNCIAS

A JARTARGHAR, H. et al. React Apps with Server-Side Rendering: Next.js. **Journal of Telecommunication, Electronic and Computer Engineering (JTEC)**, v. 14, n. 4, p. 25–29, 30 dez. 2022.

ALVI, Farooq. **Why You Need To Start Learning OpenCV in 2025**. OpenCV Blog, 2025. Disponível em: https://opencv.org/blog/learning-opencv/. Acesso em: 29 mar. 2025.

AMAZON WEB SERVICES. **O que é uma API (interface de programação de aplicações)?** Amazon Web Services, [s.d.]. Disponível em: https://aws.amazon.com/pt/what-is/api/. Acesso em: 30 mar. 2025.

BARELLI, Felipe. **Introdução à visão computacional:uma abordagem prática com python e opencv**. São Paulo, SP: Casa do Código, 2018.E-book. Disponível em: https://plataforma.bvirtual.com.br. Acesso em: 29 mar. 2025.

CRUZ, Felipe. **Python:escreva seus primeiros programas**. São Paulo, SP: Casa do Código, 2015.E-book. Disponível em: https://plataforma.bvirtual.com.br. Acesso em: 29 mar. 2025.

GALVÃO, D.; ANDRADE, J. O conceito de smart cities aliado à mobilidade urbana. **Revista Human@e**, v. 10, n. 1, p. 1-19, 23 mar. 2016.

GONZALEZ, R. C.; WOODS, R. E. **Processamento digital de imagens**. 3. ed. São Paulo, SP: Pearson, 2009.E-book. Disponível em: https://plataforma.bvirtual.com.br. Acesso em: 29 mar. 2025.

HARIOM, Tatsat,; SAHIL, Puri,; BRAD, Lookabaugh, **Blueprints de aprendizado de máquina e ciência de dados para finanças: desenvolvendo desde estratégias de trades até robôs Advisors com Python**. Rio de Janeiro: Editora Alta Books, 2024.E-book.p.8. ISBN 9788550821726. Disponível em: https://app.minhabiblioteca.com.br/reader/books/9788550821726/. Acesso em: 29 mar. 2025.

MARENGONI, Maurício; STRINGHINI, Denise. **Introdução à Visão Computacional usando OpenCV**. Revista de Informática Teórica e Aplicada (RITA), v. 16, n. 1, p. 125-142, 2009. Disponível em: https://seer.ufrgs.br/index.php/rita/article/view/rita_v16_n1_p125/7289. Acesso em: 29 mar. 2025.

MARTINS, Mateus. **Como criar uma API REST com Flask**. Medium, 03 fev. 2023. Disponível em: https://mateus-dev-me.medium.com/como-criar-uma-api-rest-com-flask-29b6845f1f1c. Acesso em: 30 mar. 2025.

RICH, Elaine; KNIGHT, Kevin. **Artificial Intelligence**. 3. ed. New York: McGraw-Hill, 1991.
Disponível em:
http://103.83.136.203:802/KDK-%20DATA%20CENTER/2.3)%20Knowledge%20Res

ources%20for%20Library%20Enrichment/E%20books/CT-IT%20Department/VII%20and%20VIII%20Sem/artificial-intelligence-rich-knight.pdf. Acesso em: 29 mar. 2025.

RODRIGUES, J. Mobilidade urbana no Brasil: crise e desafios para as políticas públicas. **R. TCEMG**, v. 34, n. 3, p. 80–93, jul. 2016.

ROMERO, Daniel. **Containers com docker: do desenvolvimento à produção.** São Paulo, SP: Casa do Código, 2015.E-book. Disponível em: https://plataforma.bvirtual.com.br. Acesso em: 29 mar. 2025.

RUSSELL, Stuart J.; NORVIG, Peter.**Inteligência Artificial: Uma Abordagem Moderna**. 4. ed. Rio de Janeiro: GEN LTC, 2022.E-book.p.1. ISBN 9788595159495. Disponível em: https://app.minhabiblioteca.com.br/reader/books/9788595159495/. Acesso em: 29 mar. 2025.

SÁ, Priscila Cathlen Alves; QUEZIA, Ana; MARCUS, Cleber; MACIEL, Alexandre M. A.; BASTOS-FILHO, Carmelo; JUNIOR, Claudemiro Lima. **YOLOv8 para Controle de Produção Pós-colheita e Beneficiamento de Frutos**. Revista de Engenharia e Pesquisa Aplicada, v. 9, n. 1, p. 115-122, 2024. Disponível em: https://doi.org/10.25286/repa.v9i1.2788. Acesso em: 29 mar. 2025.

SANTANA,Eduardo Felipe Zambom. **O que faz uma pessoa desenvolvedora Back-End?** Alura, 04 jul. 2024. Disponível em: https://www.alura.com.br/artigos/backend#:~:text=na%20nuvem%20AWS.-,O%20que%20faz%20uma%20pessoa%20desenvolvedora%20Back%2DEnd?,implantar%20aplica%C3%A7%C3%B5es%20na%20nuvem%20AWS.. Acesso em: 30 mar. 2025.

SCARINGELLA, R. S. A CRISE DA MOBILIDADE URBANA EM SÃO PAULO. **São Paulo em Perspectiva**, v. 15, n. 1, p. 55–59, jan. 2001.

SIMRAN. **Guide to OpenCV and Python-Dynamic Duo of Image Processing**. ProjectPro, 28 out. 2024. Disponível em: https://www.projectpro.io/article/opencv-python/792. Acesso em: 30 mar. 2025.

TAVARES, E. M. et al. CIDADES INTELIGENTES: UMA REVISÃO DA RELAÇÃO ENTRE A TECNOLOGIA E A MOBILIDADE URBANA SUSTENTÁVEL. **EPITAYA eBooks**, p. 106–114, 1 jan. 2024.

VERCEL. **Getting Started | Next.js**. Disponível em: https://nextjs.org/docs. Acesso em: 01 abr. 2025

VISIONPLATFORM.AI. **YOLOv8: Detecção de Objetos de Última Geração em Visão Computacional**. VisionPlatform.ai, 10 maio 2024. Disponível em: https://visionplatform.ai/pt/yolov8-deteccao-de-objetos-de-ultima-geracao-em-visao-computacional-computer-vision/. Acesso em: 30 mar. 2025.

WIKIPÉDIA. **Segmentação (processamento de imagem)**. Disponível em: https://pt.wikipedia.org/wiki/Segmenta%C3%A7%C3%A3o_%28processamento_de_imagem%29. Acesso em: 29 mar. 2025.