

Politécnico
Castelo Branco
Escola Superior
de Tecnologia

Desenvolvimento de um Sistema de Controlo de Qualidade através de Visão por Computador para a Indústria de Cablagem Automóvel

Carlos Jorge Gomes Ribeiro

20210010

Orientadores

Doutor Paulo Jorge Sequeira Gonçalves, Professor Coordenador

Samuel Sousa Santos, Engenheiro (Versigent)

Trabalho de Projeto apresentado à Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco para cumprimento dos requisitos necessários à obtenção do grau de Licenciado em Engenharia e Gestão Industrial, realizada sob a orientação científica de Doutor Paulo Jorge Sequeira Gonçalves, do Instituto Politécnico de Castelo Branco.

Setembro de 2026

Presidente do júri

Doutor Armando Lopes Ramalho
Professor Coordenador do Instituto Politécnico de Castelo Branco

Vogais

Doutor Paulo Jorge Sequeira Gonçalves
Professor Coordenador do Instituto Politécnico de Castelo Branco (Orientador)

Samuel Sousa Santos
Engenheiro, Versigent Castelo Branco (Co-orientador)

Doutor Nuno Octávio Garcia Fernandes
Professor Coordenador do Instituto Politécnico de Castelo Branco (Arguente)

Resumo

O presente trabalho de projeto teve como objetivo o desenvolvimento do WireVision, um sistema de controlo de qualidade baseado em visão por computador, aplicado a várias fases do processo de construção de cabos na indústria de cablagem automóvel. O trabalho foi desenvolvido em articulação com a Versigent, sob orientação científica do Instituto Politécnico de Castelo Branco.

O sistema assenta num pipeline sequencial de três fases: deteção da existência dos fios vermelho e preto, verificação da correta separação entre os dois fios, prevenindo curto-circuitos originados por ligações incorretas, e verificação do isolamento dos fios de acordo com as normas de segurança da empresa. Cada fase utiliza um modelo YOLO, tendo o sistema evoluído do YOLOv5x, passando pelo YOLOv8x, até à arquitetura final YOLO26m, adotada nas três fases pela sua eficiência computacional superior, particularmente relevante num sistema executado inteiramente em CPU, no Raspberry Pi 5.

A construção do dataset envolveu a seleção manual de aproximadamente 400 imagens por fase, com exclusão criteriosa de imagens comprometidas por fios não visíveis, sombras e borrões de movimento, seguida da anotação com LabelImg. A inferência dos modelos é isolada em subprocessos independentes, comunicando via HTTP.

Os resultados de validação, obtidos contra imagens de fábrica reservadas e nunca vistas em treino, confirmam a fiabilidade do sistema: as fases de deteção do estado do conector atingem precisão próxima de 98,5% e mAP50 de aproximadamente 99%, enquanto a deteção de cor dos fios, mais suscetível a variações de iluminação, apresenta 88,5% de precisão, identificando-se uma oportunidade de melhoria futura. O sistema encontra-se implementado e testado em ambiente de produção com Docker e systemd, permitindo arranque e recuperação automática do serviço. Os testes evidenciaram, contudo, limitações térmicas do Raspberry Pi 5 sob carga computacional prolongada, identificadas como uma área prioritária de melhoria, e demonstram que é possível implementar sistemas de inspeção visual eficientes em hardware de baixo custo.

Palavras-chave: Visão por Computador; YOLO; Controlo de Qualidade Industrial; Raspberry Pi; HTTP.

Abstract

This project work aimed to develop WireVision, a computer vision-based quality control system applied to several stages of the wire harness manufacturing process in the automotive cabling industry. The work was carried out in collaboration with Versigent, under the scientific supervision of the Polytechnic Institute of Castelo Branco.

The system is built around a sequential three-stage pipeline: detection of the presence of the red and black wires, verification of correct separation between the two wires, preventing short circuits caused by incorrect connections, and verification of wire insulation according to the company's safety standards. Each stage relies on a YOLO model, with the system evolving from YOLOv5x, through YOLOv8x, to the final YOLO26m architecture, adopted across all three stages for its superior computational efficiency, particularly relevant for a system running entirely on CPU on a Raspberry Pi 5.

Building the dataset involved manually selecting approximately 400 images per stage, carefully excluding images compromised by non-visible wires, shadows, and motion blur, followed by annotation using LabelImg. Model inference is isolated in independent subprocesses, communicating via HTTP.

Validation results, obtained against reserved factory images never seen during training, confirm the system's reliability: the connector-state detection stages achieve precision close to 98.5% and mAP50 of approximately 99%, while wire-color detection, more susceptible to lighting variations, achieves 88.5% precision, identifying an opportunity for future improvement. The system is deployed and tested in production using Docker and systemd, enabling automatic startup and recovery of the service. Testing revealed, however, thermal limitations of the Raspberry Pi 5 under prolonged computational load, identified as a priority area for improvement, and demonstrate that efficient visual inspection systems can be implemented on low-cost hardware.

Keywords: Computer Vision; YOLO; Industrial Quality Control; Raspberry Pi; HTTP.

Índice

1 Introdução **1**

1.1 Contexto e Motivação 1

1.2 Objetivos 2

1.3 O que é o WireVision 3

2 Tecnologias Fundamentais: Docker e YOLO **4**

2.1 Docker 4

2.1.1 O que é e para que serve 4

2.1.2 Como Funciona 4

2.2 YOLO 5

2.2.1 O que é o YOLO 5

2.2.2 Para que Serve 5

2.2.3 Como é Usado: Ciclo de Treino e Inferência 6

2.2.4 Evolução do YOLO 6

2.2.5 Comparação entre Gerações 8

2.3 Estado da Arte 9

3 Materiais e Métodos **10**

3.1 Hardware 10

3.1.1 Câmaras e Ambiente de Captura 11

3.2 Stack de Software e Bibliotecas 11

3.3 Arquitetura do Sistema 12

3.4 Dataset e Treino dos Modelos 14

3.4.1 Seleção das Imagens 14

3.4.2 Anotação com LabelImg 15

3.4.3 Configuração do Treino 16

4 Implementação e Deployment **18**

4.1 Processo de Desenvolvimento 18

4.1.1 Limiar de Confiança Fixo (*Hardcoded*) 19

4.1.2 Gestão de Memória e Desempenho 19

4.1.3 Disponibilidade dos Dispositivos de Câmara no Arranque 19

4.2 Deployment no Raspberry Pi 20

4.3 Problemas de Produção 21

4.3.1 Throttling Térmico 21

5 Resultados e Validação **22**

5.1 Validação com Imagens de Fábrica 22

5.1.1 Limitação: Ausência de Peças Defeituosas no Teste 23

5.2 Comparação de Desempenho 24

5.3 Cumprimento dos Requisitos 25

5.4 Análise Crítica 26

6 Conclusão **28**

6.1 Lições Aprendidas 28

6.2 Trabalho Futuro 29

6.3 Considerações Finais 31

Referências Bibliográficas **33**

Lista de Figuras

2.1 Linha temporal da evolução do YOLO, com destaque para as versões utilizadas no WireVision 7

3.1 Raspberry Pi 5 utilizado no WireVision 10

3.2 Arquitetura do sistema WireVision 12

3.3 Exemplo de anotação de imagem com a ferramenta LabelImg 15

3.4 Curvas de convergência do treino (Fase 2): métricas de validação e *loss* por época 17

4.1 Fluxograma do funcionamento do WireVision 18

5.1 Exemplos de deteção nas três fases do pipeline 24

5.2 Exemplos de falsos positivos da classe *black* na Fase 1, causados por sombras e fundo escuro 26

Lista de Tabelas

1.1 Requisitos do sistema WireVision 2

2.1 Comparação entre YOLOv8 e YOLO26 por escala 8

5.1 Resultados de validação por fase (conjunto de imagens de fábrica reservado) 22

5.2 Cumprimento dos requisitos do sistema 25

Lista de abreviaturas, siglas e acrónimos

COCO Common Objects in Context

CPU Central Processing Unit

DFL Distribution Focal Loss

FAR False Acceptance Rate

FRR False Rejection Rate

GIL Global Interpreter Lock

GPU Graphics Processing Unit

HTTP HyperText Transfer Protocol

IoU Intersection over Union

JPEG Joint Photographic Experts Group

mAP mean Average Precision

MJPEG Motion JPEG

NMS Non-Maximum Suppression

PC Personal Computer

RAM Random Access Memory

SSE Server-Sent Events

UI User Interface

V4L2 Video4Linux2

VM Virtual Machine

YOLO You Only Look Once

1. Introdução

1.1 Contexto e Motivação

Um erro na montagem de um cabo ou conector cores trocadas, um posicionamento incorreto, um isolamento defeituoso pode comprometer sistemas elétricos inteiros num veículo. Na indústria automóvel, onde a fiabilidade não é negociável, isto torna o controlo de qualidade da cablagem um dos pontos mais sensíveis da linha de produção.

A inspeção destes processos é tradicionalmente feita por olho humano. Funciona, mas tem limites óbvios: cansaço, variação de critério entre operadores, ritmo de produção elevado que não perdoa distrações. A visão por computador oferece uma alternativa capaz de manter o mesmo nível de exigência peça após peça, sem essa variabilidade.

Este projeto nasceu precisamente dessa necessidade, desenvolvido em articulação entre o Laboratório de Robótica e Equipamentos Inteligentes do Instituto Politécnico de Castelo Branco e a Versigent, sob orientação científica do Professor Coordenador Doutor Paulo Jorge Sequeira Gonçalves e orientação externa do Engenheiro Samuel Sousa Santos.

1.2 Objetivos

O objetivo central é simples de enunciar, ainda que exigente de concretizar: desenvolver um sistema de controlo de qualidade por visão por computador para várias fases do processo de construção de cabos. Mais concretamente, o trabalho propõe-se a:

- Identificar quais as fases do processo de construção de cabos que justificam controlo por visão por computador;
- Definir, para cada fase, as características concretas a controlar;
- Aplicar algoritmos de visão por computador e outros métodos de análise de imagem capazes de detetar não conformidades com fiabilidade;
- Garantir tempos de análise compatíveis com o ritmo de uma linha de produção real;
- Elaborar um sistema capaz de gerir a informação produzida ao longo do processo de controlo de qualidade.

Ao longo do desenvolvimento, a fiabilidade acabou por prevalecer sobre a velocidade sempre que os dois entraram em tensão um falso negativo tem custo real na linha de produção, um sistema alguns milissegundos mais lento não.

Com base nos objetivos definidos, foram estabelecidos os requisitos apresentados na Tabela 1.1, que servem de referência para a avaliação do sistema no Capítulo 5.

Tabela 1.1: Requisitos do sistema WireVision

Req.	**Descrição**
R1	Identificar os fios vermelho e preto (Fase 1)
R2	Validar a correta separação dos fios (Fase 2)
R3	Validar o isolamento dos fios (Fase 3)
R4	Tempo máximo de inspeção inferior a 20 segundos, requisito definido pela Versigent
R5	Taxa máxima de falso aceite (FAR) a definir em trabalho futuro
R6	Arranque automático do sistema após falha
R7	Funcionamento contínuo em ambiente de produção a validar em trabalho futuro

1.3 O que é o WireVision

O WireVision é o sistema resultante deste projeto: um pipeline de visão por computador para deteção automatizada de defeitos em fios e conectores, ao longo de três fases sucessivas.

1. **Fase 1 Deteção de cor dos fios**: confirma a existência dos dois fios que compõem o cabo, um vermelho e um preto;
2. **Fase 2 Verificação de separação dos fios (2ndStep)**: verifica se esses dois fios estão devidamente separados, evitando ligações mal feitas que possam originar um curto-circuito;
3. **Fase 3 Verificação de isolamento (3rdStep)**: confirma que o isolamento dos fios cumpre as normas de segurança definidas pela empresa.

O sistema corre num Raspberry Pi 5, com interface gráfica para monitorização em tempo real, streaming de vídeo das câmaras de inspeção, e um conjunto de modelos YOLO responsável pela análise automática de cada fase.

Os capítulos seguintes apresentam as tecnologias fundamentais utilizadas (Capítulo 2), os materiais e métodos do desenvolvimento (Capítulo 3), a implementação e deployment em ambiente de produção (Capítulo 4), os resultados e respetiva validação (Capítulo 5), e as principais conclusões e trabalho futuro (Capítulo 6).

2. Tecnologias Fundamentais: Docker e YOLO

Este capítulo apresenta uma explicação conceptual das duas tecnologias centrais na conceção do WireVision: o Docker, utilizado para a containerização e implementação do sistema, e o YOLO, a arquitetura de deteção de objetos que constitui o núcleo de inteligência artificial do sistema. Apresenta-se também uma breve revisão do estado da arte em inspeção visual industrial baseada em visão por computador.

2.1 Docker

2.1.1 O que é e para que serve

Docker [10] é uma plataforma de containerização que permite empacotar uma aplicação, com todas as suas dependências e configurações, num pacote isolado e portátil designado *container*. Ao contrário de uma máquina virtual tradicional, um container partilha o núcleo (*kernel*) do sistema operativo anfitrião, tornando-o mais leve e rápido a iniciar, mantendo um isolamento eficaz entre aplicações.

O principal benefício do Docker é garantir que uma aplicação se comporta de forma idêntica independentemente da máquina onde é executada, resolvendo problemas comuns causados por diferenças de versões de software ou configurações entre ambientes.

2.1.2 Como Funciona

O ciclo de vida de uma aplicação em Docker assenta em quatro conceitos principais:

- **Dockerfile** um ficheiro de texto que descreve, passo a passo, como construir o ambiente da aplicação (por exemplo, a versão do sistema operativo base, as bibliotecas a instalar e o código a copiar);
- **Imagem** o resultado da construção (*build*) do Dockerfile: um "molde"estático e reprodutível que contém tudo o que a aplicação precisa para correr;
- **Container** uma instância em execução de uma imagem, isolada do resto do sistema, mas capaz de comunicar com o exterior através de portas de rede e volumes de dados configurados;

- **Docker Compose** uma ferramenta que permite definir, num único ficheiro (`docker-compose.yml`), toda a configuração necessária para arrancar e gerir a aplicação (portas, volumes, políticas de reinício), simplificando o processo para um único comando em vez de vários comandos Docker separados.

No contexto do WireVision, esta arquitetura garante que o sistema pode ser reconstruído e implementado de forma consistente noutro Raspberry Pi, sem necessidade de reconfigurar manualmente o ambiente de desenvolvimento. Os detalhes específicos da implementação do WireVision em Docker, incluindo o processo de atualização em produção, são discutidos na Secção 4.2.

2.2 YOLO

2.2.1 O que é o YOLO

YOLO (*You Only Look Once*) [1] é uma família de modelos de deteção de objetos em imagens, inicialmente apresentada como *preprint* em 2015 e publicada na CVPR em 2016. Ao contrário de abordagens anteriores de deteção de objetos, que processavam a imagem em duas etapas distintas primeiro propondo regiões candidatas onde poderiam existir objetos, depois classificando cada uma dessas regiões separadamente o YOLO processa a imagem inteira numa única passagem pela rede neuronal, daí o nome "só olhas uma vez". Esta abordagem torna-o consideravelmente mais rápido do que os métodos anteriores, o que o tornou a escolha dominante para tarefas de deteção de objetos em tempo real.

2.2.2 Para que Serve

O YOLO serve para localizar e classificar objetos dentro de uma imagem ou vídeo: para cada objeto que reconhece, o modelo devolve uma caixa delimitadora (indicando onde o objeto se encontra) e uma classe com uma pontuação de confiança associada (indicando o que é o objeto, e quão certo o modelo está dessa classificação). Esta tecnologia é amplamente utilizada em áreas como vigilância, condução autónoma, comércio retalhista, imagem médica e, como no caso do WireVision, inspeção e controlo de qualidade industrial.

2.2.3 Como é Usado: Ciclo de Treino e Inferência

A utilização de um modelo YOLO passa por duas fases distintas:

- **Treino** o modelo é alimentado com um conjunto de imagens previamente anotadas manualmente (cada objeto de interesse identificado com a respetiva caixa delimitadora e classe), ajustando os seus parâmetros internos de forma a aprender a reconhecer esses padrões visuais;
- **Inferência** o modelo já treinado recebe imagens novas, nunca vistas durante o treino, e devolve as respetivas deteções (caixas, classes e níveis de confiança), tipicamente em poucos milissegundos ou segundos, dependendo do hardware disponível.

No WireVision, cada fase do pipeline de inspeção utiliza o seu próprio modelo, treinado especificamente para reconhecer os elementos relevantes dessa fase. Uma imagem capturada pela câmara é enviada ao modelo da fase ativa; se a confiança da deteção ultrapassar um limiar definido e se mantiver estável ao longo de várias análises consecutivas, a fase é considerada concluída e o sistema avança para a fase seguinte, conforme detalhado na Secção 4.1.

2.2.4 Evolução do YOLO

Desde a sua introdução original por Joseph Redmon e colaboradores [1], o YOLO passou por várias gerações sucessivas, desenvolvidas tanto pelos autores originais como por diferentes equipas e empresas. O **YOLOv1** introduziu o conceito fundamental de deteção de objetos como um problema de regressão único, processando a imagem inteira numa só passagem pela rede. As versões seguintes, **YOLOv2** (2016) [2] e **YOLOv3** (2018) [3], ainda da autoria de Redmon, introduziram melhorias como as caixas âncora (*anchor boxes*) e previsões em múltiplas escalas.

Após 2020, a implementação e família YOLO utilizada neste projeto passou a ser mantida pela empresa **Ultralytics**, que lançou o **YOLOv5** em 2020 [4], marcando a transição da framework original Darknet para PyTorch. Em 2023, a Ultralytics lançou o **YOLOv8** [5], unificando o desenvolvimento da família sob um único pacote de software e introduzindo uma cabeça de deteção sem âncoras predefinidas. Por fim, em janeiro de 2026, a Ultralytics lançou o **YOLO26** [6, 7], a geração mais recente da família, e a que foi adotada como arquitetura final no WireVision (variante YOLO26m).

As duas alterações mais relevantes desta geração, no contexto deste projeto, são:

- **Remoção do NMS (*Non-Maximum Suppression*)** nas gerações anteriores, era necessário um passo adicional de pós-processamento para eliminar deteções duplicadas e sobrepostas. O YOLO26 elimina este passo, produzindo diretamente as deteções finais (*end-to-end*), tornando o tempo de inferência mais rápido e previsível;
- **Remoção da DFL (*Distribution Focal Loss*)** simplifica o cálculo da posição das caixas delimitadoras, substituindo uma modelação probabilística mais pesada por uma regressão direta das coordenadas.

Segundo a Ultralytics [7], estas alterações resultam numa melhoria de até 43% na velocidade de inferência em CPU face a gerações anteriores de tamanho equivalente um fator particularmente relevante para o WireVision, que corre inteiramente em CPU, sem GPU dedicada.

A Figura 2.1 resume esta evolução, destacando as versões utilizadas no WireVision.



Figura 2.1: Linha temporal da evolução do YOLO, com destaque para as versões utilizadas no WireVision

2.2.5 Comparação entre Gerações

A comparação de desempenho entre modelos de deteção de objetos é normalmente expressa através da métrica **mAP** (*mean Average Precision*), que resume, num único valor, a capacidade de um modelo detetar corretamente os objetos presentes numa imagem, considerando tanto a classificação correta como a exatidão da localização da respetiva caixa delimitadora.

Esta métrica é calculada com base num limiar de sobreposição (*Intersection over Union*, IoU) entre a caixa prevista e a caixa real anotada manualmente. O **mAP50** considera uma deteção correta quando a sobreposição é de, pelo menos, 50%, sendo um critério mais permissivo. O **mAP50-95** corresponde à média do mAP calculado para múltiplos limiares de IoU, entre 50% e 95%, sendo por isso um critério mais rigoroso motivo pelo qual o mAP50-95 de um modelo é sempre inferior ao seu mAP50.

A Tabela 2.1 apresenta a comparação oficial de desempenho entre as gerações YOLOv8 e YOLO26, medida em mAP50-95 sobre o conjunto de referência COCO [8], para as cinco escalas disponíveis em cada família.

Tabela 2.1: Comparação entre YOLOv8 e YOLO26 por escala

Escala	**YOLOv8 mAP50-95**	**YOLOv8 Parâmetros**	**YOLO26 mAP50-95**	**YOLO26 Parâmetros**
n (nano)	37.3%	3.2 M	40.1%	2.4 M
s (small)	44.9%	11.2 M	47.8%	9.5 M
m (medium)	50.2%	25.9 M	52.5%	20.4 M
l (large)	52.9%	43.7 M	54.4%	24.8 M
x (extra-large)	53.9%	68.2 M	56.9%	55.7 M

A escolha da arquitetura final do WireVision resultou de um processo iterativo que testou três gerações: **YOLOv5x**, **YOLOv8x** e **YOLO26m**. A análise da Tabela 2.1 justifica a migração para o **YOLO26m**: a perda de precisão é marginal (53.9% vs. 52.5%, mantendo o mAP50 $\approx$ 0,99), mas a redução de tamanho é substancial. O YOLO26m usa apenas $\sim$30% dos parâmetros do YOLOv8x, resultando num ficheiro mais leve (42 MB vs. 131 MB) e numa **redução estimada de $\sim$74% no custo de inferência**, calculada a partir da diferença de parâmetros entre os dois modelos — crucial para execução inteiramente em CPU. Este valor não resulta de uma medição experimental direta de tempos de inferência no sistema WireVision: os modelos YOLOv5x e YOLOv8x utilizados nas fases iniciais do projeto não foram preservados, nem as respetivas métricas documentadas de forma sistemática, o que impossibilita atualmente uma comparação experimental direta entre as três gerações no mesmo dataset e hardware. Esta comparação é identificada como limitação metodológica e prioridade de trabalho futuro (Secção 6.2). Detalhes do treino constam na Secção 3.4.

2.3 Estado da Arte

A automatização do controlo de qualidade por visão computacional tem sido amplamente estudada na literatura científica, com aplicações que vão desde a inspeção genérica de superfícies até casos muito próximos do contexto deste projeto.

Hütten et al. [17], numa revisão sistemática de 196 publicações sobre inspeção visual automatizada com deep learning em contextos de fabrico e manutenção, identificam o YOLO como uma das arquiteturas mais utilizadas em tarefas de localização multi-classe, confirmando a adequação desta família de modelos ao tipo de problema abordado no WireVision.

Na área específica de inspeção de conectores e cablagem, Wu e Li [18] aplicaram uma versão melhorada do YOLOv3 à inspeção visual de conectores elétricos, focando-se na deteção de pequenos defeitos de soldadura, um problema de natureza semelhante à deteção de estado de conectores realizada nas Fases 2 e 3 do WireVision. De forma semelhante, Chen et al. [19] propuseram o PO-YOLOv5, um modelo baseado em YOLOv5 especificamente otimizado para deteção de defeitos em conectores solenoides, com particular atenção à distinção entre elementos com cor e forma semelhantes um desafio próximo do enfrentado na Fase 1 do WireVision, na distinção entre fios de cores distintas sob condições de iluminação variáveis.

Relativamente à implementação em hardware de baixo custo, Okano et al. [20] apresentam um caso de estudo particularmente próximo deste projeto: a aplicação do YOLOv8 num Raspberry Pi para deteção de conformidade visual de tubos extrudidos na indústria automóvel, com treino realizado em GPU e inferência exclusivamente em CPU no dispositivo embebido. Os autores reportam tempos de inferência da ordem dos 470 ms (variante Nano) a 1315 ms (variante Small), valores da mesma ordem de grandeza dos observados no WireVision, reforçando a viabilidade mas também as limitações de desempenho da utilização de hardware embebido sem aceleração por GPU em aplicações de controlo de qualidade industrial em tempo real.

Em conjunto, estes trabalhos confirmam que a abordagem seguida no WireVision modelos da família YOLO aplicados à deteção de defeitos em componentes elétricos, implementados em hardware embebido de baixo custo está alinhada com tendências de investigação atuais, ao mesmo tempo que evidenciam desafios comuns à literatura, nomeadamente a sensibilidade a condições de iluminação e a limitação de desempenho inerente à execução exclusivamente em CPU.

3. Materiais e Métodos

Este capítulo descreve os materiais, tecnologias e metodologias utilizados no desenvolvimento do WireVision, incluindo o hardware de suporte, a stack de software, a arquitetura do sistema e o processo de construção e treino dos modelos de deteção.

3.1 Hardware

O sistema foi desenvolvido e é executado num **Raspberry Pi 5** [14], equipado com 16GB de memória RAM, escolhido pela sua capacidade de processamento suficiente para correr modelos de deteção de objetos em tempo real num formato compacto e de baixo custo, adequado à integração numa linha de produção industrial.

A aquisição de imagem é feita através de câmaras ligadas via V4L2 (*Video4Linux2*), disponíveis nos dispositivos `/dev/video0` e `/dev/video1`, permitindo a captura contínua das diferentes fases do processo de montagem dos cabos.

Durante a fase de desenvolvimento, foi também utilizada uma máquina virtual (VM) para testes preliminares antes da implementação em ambiente de produção no Raspberry Pi, permitindo validar alterações de forma segura sem interromper o funcionamento do sistema na linha de produção.

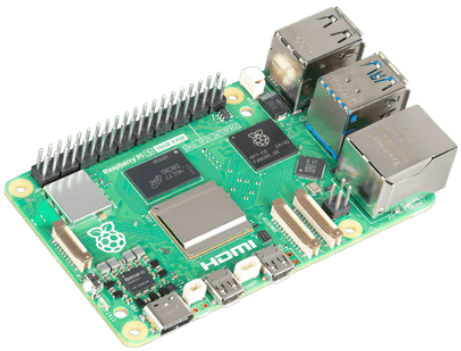


Figura 3.1: Raspberry Pi 5 utilizado no WireVision

3.1.1 Câmaras e Ambiente de Captura

A aquisição de imagem é feita através de duas câmaras USB genéricas (*webcams*), ligadas via V4L2 nos dispositivos `/dev/video0` e `/dev/video1`, com uma resolução de captura e análise de 640×480 pixels. Durante os testes, foi utilizada iluminação dedicada e controlada, direcionada ao objeto de inspeção, de forma a reduzir a dependência exclusiva da luz ambiente do local.

Não foram documentados de forma sistemática, ao longo do desenvolvimento, outros parâmetros do ambiente de aquisição relevantes para a reprodutibilidade do sistema, nomeadamente o modelo exato das câmaras, a taxa de imagens por segundo (FPS), a distância e o campo de visão (FOV) relativamente ao objeto, e se a exposição e o balanço de brancos são geridos de forma automática ou fixa. Dado que os problemas mais recorrentes identificados neste trabalho sombras, reflexos e falsos positivos da classe *black* (Secção 5.4) estão diretamente relacionados com condições de iluminação e captura, a documentação e controlo rigoroso destes parâmetros é identificada como uma via de melhoria prioritária, possivelmente com maior impacto na fiabilidade do sistema do que o simples aumento do volume de dados de treino.

3.2 Stack de Software e Bibliotecas

O WireVision foi desenvolvido em **Python 3.13**, recorrendo a um conjunto de bibliotecas e frameworks especializados:

- **Streamlit** [13] utilizado para a construção da interface gráfica (UI) principal do sistema, permitindo a monitorização e configuração do processo de inspeção;
- **Flask** responsável pelo servidor de streaming de vídeo em formato MJPEG, possibilitando a visualização do feed de câmara em tempo real;
- **OpenCV** [12] utilizado para o processamento de imagem e vídeo, incluindo captura, manipulação e pré-processamento dos frames antes da análise;
- **YOLO** / **Ultralytics** framework de deteção de objetos utilizado para identificar e classificar os elementos relevantes em cada fase de inspeção (ver Capítulo 2 para uma explicação detalhada desta tecnologia e da sua evolução);
- **PyTorch** [11] framework de deep learning subjacente aos modelos YOLO, responsável pela execução da inferência;
- **Docker** [10] plataforma de containerização utilizada para empacotar a aplicação de forma isolada e reprodutível (ver Capítulo 2 para uma explicação detalhada);

- **systemd** [15] utilizado para a gestão do serviço em produção, assegurando o arranque automático e reinício em caso de falha.

A comunicação de métricas em tempo real entre o backend e a interface é realizada através de *Server-Sent Events* (SSE), enquanto o streaming de vídeo é feito via MJPEG sobre Flask. A comunicação entre o processo principal da aplicação e os subprocessos de inferência (descrita na Secção 3.3) é também realizada através do protocolo **HTTP**, permitindo que cada modelo funcione como um microserviço local independente.

3.3 Arquitetura do Sistema

O WireVision está organizado num pipeline sequencial de três fases (descritas em detalhe no Capítulo 1), cada uma responsável por uma verificação específica no processo de montagem dos cabos.

Um aspeto central da arquitetura é o isolamento da inferência dos modelos YOLO em **subprocessos independentes**, geridos através de um módulo dedicado (`inference_worker.py`) que expõe cada modelo como um microserviço HTTP local, gerido por uma classe `WorkerPool` responsável pelo ciclo de vida dos processos, com um número configurável de *workers* (`NUM_WORKERS`) a correr em portas dedicadas.

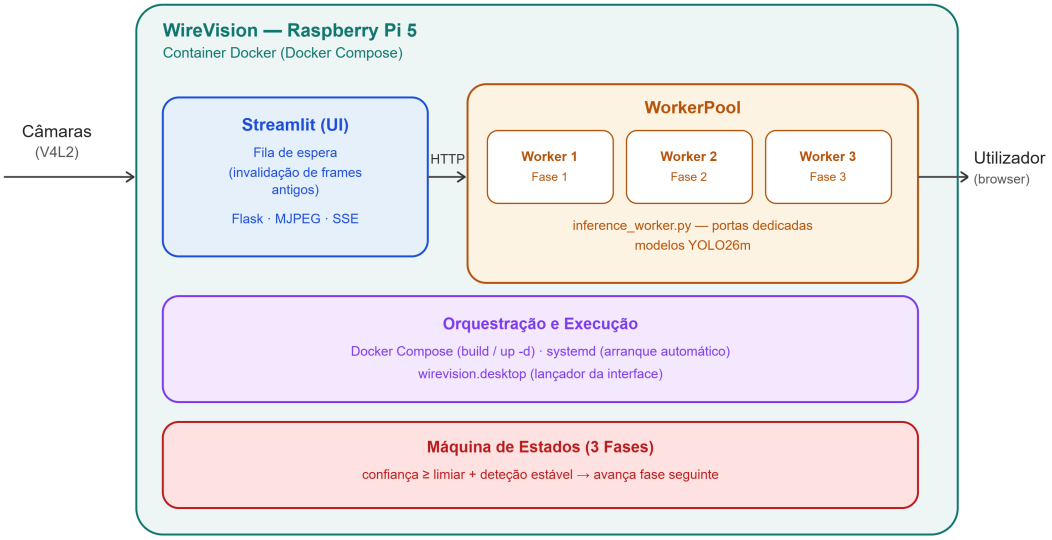


Figura 3.2: Arquitetura do sistema WireVision

Esta decisão resultou de um problema inicial: a execução da inferência em threads dentro do mesmo processo da interface Streamlit provocava contenção de recursos e perda de responsividade da interface durante a análise, tendo-se identificado a interação com o modelo de concorrência do Python (nomeadamente o *Global Interpreter Lock*) como uma possível causa. A migração para uma arquitetura de subprocessos isolados, comunicando via HTTP, resolveu empiricamente este problema.

A própria separação em subprocessos introduziu, no entanto, um novo desafio: a comunicação via HTTP entre o processo principal e os *workers* de inferência gerava atraso adicional, levando à acumulação de imagens por processar e, consequentemente, a um *delay* percetível entre a captura da imagem e a apresentação do resultado da deteção. Para resolver este problema, foi introduzida uma **fila de espera** para as imagens a processar, com um mecanismo de **invalidação**: sempre que chega uma imagem nova à fila, as imagens anteriores ainda não processadas são descartadas, garantindo que o sistema analisa sempre o *frame* mais recente disponível, em vez de acumular um histórico de imagens desatualizadas.

3.4 Dataset e Treino dos Modelos

3.4.1 Seleção das Imagens

A seleção das imagens para treino foi feita de forma manual, através da revisão visual de todas as capturas disponíveis provenientes do sistema de câmaras instalado na linha de produção. Esta abordagem, ainda que mais morosa do que uma amostragem automática, permitiu identificar e excluir imagens que comprometessem a qualidade do treino, nomeadamente casos em que:

- Os fios não estavam visíveis ou apresentavam posicionamento não conforme, dificultando a correta identificação da cor ou do estado dos fios/conectores;
- A imagem apresentava sombras acentuadas, resultantes das condições de iluminação da linha de produção, que alteravam a perceção visual das cores e contornos;
- Existiam borrões (*motion blur*), causados pelo movimento dos componentes ou vibração do sistema de captura, que tornavam os contornos dos objetos pouco definidos.

Estes três critérios foram aplicados de forma indiferenciada, sem distinguir entre imagens genuinamente inválidas (por exemplo, com o fio totalmente invisível ou um borrão extremo que impossibilita qualquer anotação fiável) e imagens que, apesar de mais difíceis, ainda representam condições reais e plausíveis de produção (por exemplo, sombras moderadas ou fundo escuro). Com a análise dos resultados de validação (Secção 5.4), tornou-se claro que esta exclusão sistemática de casos com sombras teve um efeito colateral: ao reduzir a exposição do modelo a este tipo de condição durante o treino, contribuiu indiretamente para a maior taxa de falsos positivos da classe *black* observada na Fase 1, precisamente nos casos de sombra e fundo escuro. Esta tensão entre "limpar"o dataset e representar fielmente a variabilidade real da produção é discutida como limitação e via de melhoria futura na Secção 6.2, onde se recomenda distinguir explicitamente entre imagens verdadeiramente inválidas a excluir e casos difíceis mas reais a incluir tanto no treino como no teste.

A seleção manual foi realizada separadamente para cada uma das três fases, uma vez que cada fase apresenta características visuais distintas.

3.4.2 Anotação com LabelImg

A anotação das imagens selecionadas foi realizada com recurso à ferramenta **LabelImg** [9], um software *open-source* amplamente utilizado para anotação de *bounding boxes* em tarefas de deteção de objetos. Para cada imagem, foram definidas as caixas delimitadoras correspondentes às classes relevantes de cada fase, exportadas no formato compatível com o treino dos modelos YOLO.

O número de classes anotadas variou entre fases, refletindo a natureza distinta de cada verificação: a Fase 1 conta com duas classes (*red* e *black*, correspondentes aos dois fios a identificar), enquanto as Fases 2 e 3 contam apenas com uma classe cada (*2ndStep* e *3rdStep*, respetivamente), uma vez que estas fases verificam um único estado do conector, e não a presença de múltiplos elementos distintos.

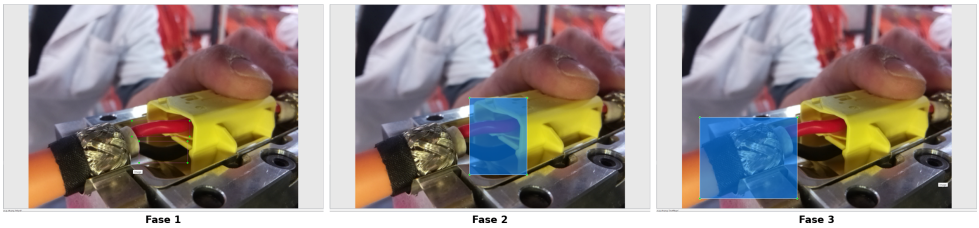


Figura 3.3: Exemplo de anotação de imagem com a ferramenta LabelImg

Todo o processo de anotação foi realizado por um único anotador, garantindo consistência no critério aplicado ao longo de todo o dataset e evitando a variabilidade que surge tipicamente quando múltiplos anotadores seguem critérios ligeiramente diferentes. Após uma primeira passagem de anotação, foi realizada uma revisão e correção das caixas delimitadoras, permitindo identificar e ajustar anotações imprecisas antes do treino dos modelos.

A precisão das anotações teve impacto direto nos resultados obtidos: imagens com fios não visíveis, sombras ou borrões, quando incluídas sem critério, tendem a introduzir ambiguidade e a degradar a generalização do modelo. Por isso, a revisão manual e criteriosa de cada imagem tanto na seleção inicial como na verificação das anotações foi essencial para maximizar a fiabilidade do sistema em produção.

3.4.3 Configuração do Treino

Conforme descrito no Capítulo 2, a arquitetura final adotada nas três fases do pipeline foi o **YOLO26m**, após um processo iterativo que testou também o YOLOv5x e o YOLOv8x. O treino foi realizado a partir dos pesos pré-treinados no conjunto COCO (`yolo26m.pt`), inteiramente em **CPU**, sem recurso a GPU dedicada, num computador portátil equipado com processador **AMD Ryzen 5 série 7000U**, com **Ultralytics 8.4.47** e **PyTorch 2.7.1**.

Para cada fase, o conjunto de imagens selecionado foi dividido em subconjuntos de treino e validação, utilizados internamente pelo Ultralytics ao longo do processo de treino:

- **Fase 1**: 300 imagens de treino, 101 de validação (401 no total);
- **Fase 2**: 323 imagens de treino, 116 de validação (439 no total);
- **Fase 3**: 365 imagens de treino, 130 de validação (495 no total).

Em todas as fases, esta divisão corresponde a uma proporção aproximada de 74–75% para treino e 25–26% para validação, seguindo a prática recomendada para conjuntos de dados de dimensão moderada. É importante notar que este conjunto de validação, usado internamente durante o treino para monitorizar a convergência do modelo, é distinto do conjunto de teste final utilizado na avaliação de desempenho apresentada no Capítulo 5, composto por imagens de fábrica reservadas e nunca vistas em qualquer etapa do treino.

O treino foi realizado com os seguintes hiperparâmetros explícitos: 40 épocas (*epochs*), tamanho de lote (*batch size*) de 8 imagens, resolução de treino de 640×640 pixels (`imgsz`), e *patience* de 30 épocas (interrupção antecipada do treino caso não haja melhoria da métrica de validação ao longo de 30 épocas consecutivas). Os restantes hiperparâmetros — nomeadamente o *optimizer*, a taxa de aprendizagem inicial e a semente aleatória (*seed*) — não foram explicitamente configurados, tendo sido utilizados os valores por omissão do Ultralytics 8.4.47, incluindo as técnicas de *data augmentation* ativadas por defeito nesta versão (nomeadamente *mosaic* e inversão horizontal aleatória).

A Figura 3.4 apresenta as curvas de convergência do treino da Fase 2, obtidas a partir do registo automático do Ultralytics (`results.csv`). As métricas de validação estabilizam a partir de aproximadamente a época 15, com precisão e mAP50 a manterem-se próximos de 0,98–0,99 até ao final das 40 épocas, enquanto as componentes de *loss* de classificação (treino e validação) decrescem acentuadamente nas primeiras 10 épocas e estabilizam nas restantes, confirmando a convergência do modelo dentro do número de épocas definido, sem sinais de *overfitting* não se observa divergência entre as curvas de *loss* de treino e de validação.

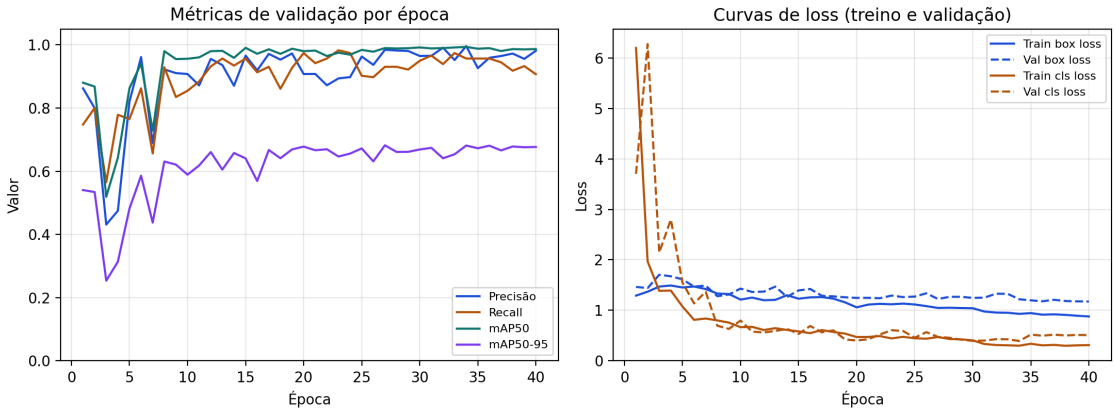


Figura 3.4: Curvas de convergência do treino (Fase 2): métricas de validação e *loss* por época

Esta configuração equilibra o tempo de treino disponível — limitado pela ausência de aceleração por GPU — com a necessidade de convergência estável do modelo, adequada à dimensão do dataset utilizado em cada fase.

4. Implementação e Deployment

Este capítulo descreve o processo de implementação, os principais desafios técnicos resolvidos e o deployment do WireVision no Raspberry Pi 5.

4.1 Processo de Desenvolvimento

O desenvolvimento seguiu uma abordagem iterativa, com testes numa VM antes de qualquer alteração em produção.

O limiar de confiança (*target_confidence*) é configurável na interface Streamlit, sem alterar código. Nos testes deste relatório, usou-se **60%**, com a deteção considerada estável após **5 *frames* consecutivos** acima do limiar critério igual nas três fases.

A Figura 4.1 ilustra o funcionamento: em cada fase, o sistema captura imagens e executa a inferência; a fase conclui-se quando a deteção ultrapassa 60% de confiança e se mantém estável em 5 análises consecutivas, avançando até à conclusão da inspeção.

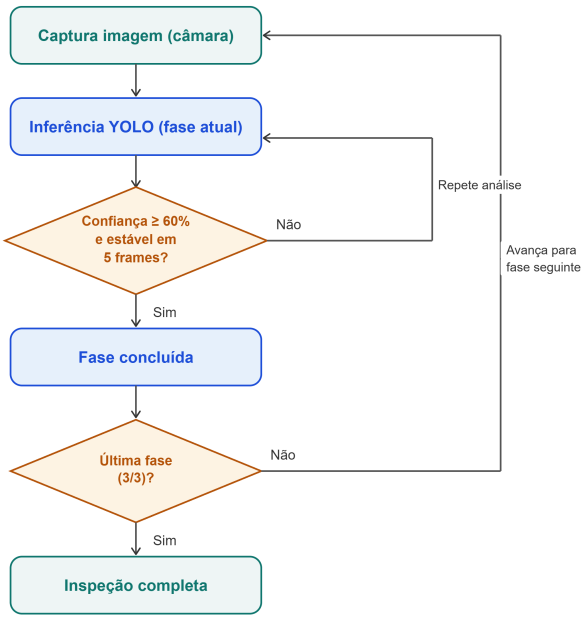


Figura 4.1: Fluxograma do funcionamento do WireVision

4.1.1 Limiar de Confiança Fixo (*Hardcoded*)

Foi identificado um problema no módulo `inference_worker.py`, no qual a Fase 1 do pipeline utilizava um valor de limiar de confiança fixo no código (`0.6`), em vez de respeitar o valor configurável `target_confidence` definido pelo utilizador. Este comportamento inconsistente entre fases foi corrigido, garantindo que todas as fases do pipeline utilizam de forma uniforme o valor de confiança configurável.

4.1.2 Gestão de Memória e Desempenho

Nos testes realizados, e desde que exista memória suficiente para evitar *swapping*, a quantidade adicional de RAM disponível no Raspberry Pi 5 (16GB) não produziu melhoria mensurável no tempo de inferência dos modelos, sendo a capacidade de processamento (CPU) o principal fator limitante observado. A maior quantidade de RAM contribui, ainda assim, para uma maior estabilidade geral do sistema, reduzindo o risco de problemas relacionados com falta de memória durante a execução prolongada.

4.1.3 Disponibilidade dos Dispositivos de Câmara no Arranque

Foi identificado um caso limite relacionado com o arranque do sistema: os dispositivos de câmara (`/dev/video0` e `/dev/video1`) podem não estar disponíveis no momento em que o contentor Docker é iniciado, uma vez que a inicialização do sistema operativo e dos dispositivos USB/V4L2 pode não estar totalmente concluída nesse instante. Este comportamento foi tido em conta na configuração do processo de arranque do sistema, de forma a evitar falhas de inicialização.

4.2 Deployment no Raspberry Pi

O WireVision é implementado em produção através de **Docker**, orquestrado com recurso a **Docker Compose**, garantindo um ambiente de execução isolado e reprodutível, independente das particularidades do sistema operativo do Raspberry Pi.

Após qualquer alteração ao código ou à configuração do sistema, o processo de atualização em produção segue a sequência de comandos apresentada de seguida:

```
docker-compose build
docker-compose up -d
```

O primeiro comando reconstrói a imagem com as alterações mais recentes ao código; o segundo arranca (ou reinicia) o serviço em segundo plano (*detached mode*), aplicando automaticamente a configuração definida no ficheiro `docker-compose.yml`, incluindo políticas de reinício em caso de falha.

Para facilitar o acesso à interface do sistema em ambiente de fábrica, foi ainda configurado um lançador de ambiente de trabalho (`wirevision.desktop`) no Raspberry Pi OS, que abre a interface do WireVision diretamente através do navegador Firefox.

4.3 Problemas de Produção

4.3.1 Throttling Térmico

Durante os testes em ambiente de produção, foi observado um padrão de comportamento específico ao longo das três fases de análise: as Fases 1 e 2 corriam normalmente, com tempos de resposta rápidos, mas ao chegar à Fase 3 o sistema bloqueava por completo, interrompendo a análise. Após um período de pausa, o sistema recuperava e reconhecia corretamente a Fase 3 logo no primeiro *frame* analisado após a retoma.

A temperatura do processador (`vcgencmd measure_temp`) e o estado de *throttling* (`vcgencmd get_throttled`) foram monitorizados durante os testes, tendo-se registado uma temperatura de aproximadamente **80 ℃** valor coincidente com o limiar oficial documentado pela Raspberry Pi Foundation [16] a partir do qual o Raspberry Pi 5 inicia a redução da frequência do processador. Esta coincidência constitui uma hipótese plausível e consistente com o comportamento observado, associando o bloqueio a um mecanismo de proteção térmica do processador, embora não tenha sido possível registar, ao longo de todo o ciclo de teste, a evolução contínua da frequência do processador e da *flag* de *throttling* em função do tempo, o que permitiria estabelecer esta relação de forma mais conclusiva.

Ainda assim, o diagnóstico é relevante para o planeamento de trabalho futuro, nomeadamente a eventual necessidade de uma solução de arrefecimento ativo, como o *Active Cooler* oficial da Raspberry Pi, e a recolha de um registo temporal mais detalhado (temperatura, frequência, estado de *throttling* e latência de inferência) que permita confirmar de forma inequívoca esta hipótese.

5. Resultados e Validação

Este capítulo apresenta os resultados obtidos na validação dos modelos YOLO26m nas três fases do pipeline de inspeção do WireVision, com base em testes realizados contra um conjunto de imagens de fábrica reservado, nunca utilizado durante o treino dos modelos, garantindo que os resultados refletem a capacidade real de generalização do sistema, e não apenas o desempenho sobre dados já conhecidos.

5.1 Validação com Imagens de Fábrica

Os resultados apresentados foram obtidos contra um conjunto de imagens de fábrica reservado para validação (`test_novas_imagens.py`), com 116 imagens por fase, distinto do conjunto usado no treino (Secção 3.4).

A seleção destas imagens seguiu o mesmo critério do dataset de treino: escolha manual, garantindo boa visibilidade do elemento a reconhecer. Ainda assim, este conjunto nunca foi utilizado em qualquer etapa do treino, constituindo um teste independente sobre dados não vistos.

Os testes foram realizados com o mesmo limiar de confiança (`target_confidence`) usado em produção, garantindo que os resultados refletem o comportamento real do sistema, e não um cenário artificialmente otimizado.

A Tabela 5.1 resume os resultados obtidos para cada uma das três fases.

Tabela 5.1: Resultados de validação por fase (conjunto de imagens de fábrica reservado)

Métrica	**Fase 1 (Vermelho/Preto)**	**Fase 2 (2ndStep)**	**Fase 3 (3rdStep)**
Precisão (P)	88.5%	98.5%	98.5%
Recall (R)	93.6%	93.0%	93.0%
mAP50	95.7%	99.0%	99.0%
mAP50-95	55.5%	68.2%	—

Nota: a Fase 3 foi validada contra o mesmo conjunto de 116 imagens de fábrica reservadas, numa sessão de teste separada e posterior às Fases 1 e 2. O valor de mAP50-95 não foi registado nessa sessão de teste.

5.1.1 Limitação: Ausência de Peças Defeituosas no Teste

O conjunto de teste utilizado nesta validação é composto exclusivamente por peças corretas (OK), refletindo a disponibilidade de dados no momento dos testes. Esta limitação impede o cálculo de duas métricas fundamentais para um sistema de controlo de qualidade industrial:

- **False Acceptance Rate (FAR)** a proporção de peças com defeito real (NOK) que o sistema classificaria incorretamente como corretas (OK), permitindo a sua passagem na linha de produção;
- **False Rejection Rate (FRR)** a proporção de peças corretas (OK) que o sistema classificaria incorretamente como defeituosas ou não reconheceria, originando uma rejeição ou repetição desnecessária.

Os valores de *recall* apresentados neste capítulo aproximam-se do complemento da FRR (peças OK corretamente reconhecidas), mas não fornecem qualquer informação sobre a FAR, que é a métrica mais crítica para um sistema de controlo de qualidade, uma vez que mede diretamente o risco de uma peça defeituosa avançar na linha de produção sem ser assinalada. A validação com peças NOK reais é identificada como uma prioridade de trabalho futuro (Secção 6.2).

5.2 Comparação de Desempenho

Todos os três modelos apresentam valores de *recall* consistentemente elevados, entre 93% e 94%. É importante notar que estes valores foram medidos exclusivamente sobre peças corretas (OK), uma vez que o conjunto de teste não inclui exemplos de peças com defeito real (NOK). Um falso negativo neste contexto significa, portanto, que uma peça corretamente montada não foi reconhecida como tal pelo sistema o que poderia originar uma repetição ou rejeição desnecessária dessa peça (falso rejeitamento), e não que uma não conformidade tenha passado despercebida, como uma interpretação apressada do recall poderia sugerir.

A diferença mais notória entre as fases está ao nível da **precisão**: enquanto as Fases 2 e 3 (deteção do estado do conector) atingem valores próximos de 98.5%, a Fase 1 (deteção de cor dos fios) apresenta uma precisão consideravelmente mais baixa, de 88.5%. Esta diferença não resulta de uma limitação da migração para o YOLO26m, mas antes da natureza distinta da tarefa: a deteção de cor está sujeita a variações de iluminação, sombras e reflexos, ao contrário da deteção do estado estrutural de um conector, que é uma tarefa visualmente mais estável e menos suscetível a condições ambientais.

A Figura 5.1 apresenta exemplos de deteção nas três fases, capturados na interface do sistema durante testes num PC/VM de desenvolvimento.

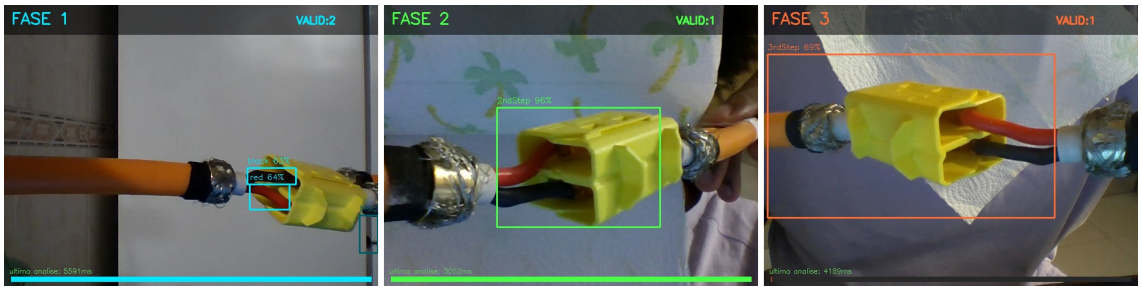


Figura 5.1: Exemplos de deteção nas três fases do pipeline

Os tempos observados nestas capturas (5591 ms, 3052 ms e 4189 ms, para as Fases 1, 2 e 3) superam o habitual devido ao *cold start* (carregamento inicial dos modelos) e a más condições de visibilidade, que também reduziram a confiança das deteções (64% e 69%). Estes valores não representam a velocidade média, servindo apenas para ilustrar o *pipeline*. Como as fases são sequenciais, o tempo total é a soma dos tempos individuais, fixando-se em $\sim$20 segundos no Raspberry Pi 5 em condições normais.

5.3 Cumprimento dos Requisitos

A Tabela 5.2 resume o grau de cumprimento dos requisitos definidos na Secção 1.2, com base nos resultados apresentados neste capítulo.

Tabela 5.2: Cumprimento dos requisitos do sistema

Req.	Descrição	Estado
R1	Identificação dos fios vermelho/preto	Cumprido (mAP50 = 95.7%)
R2	Validação da separação dos fios	Cumprido (mAP50 = 99.0%)
R3	Validação do isolamento	Cumprido (mAP50 $\approx$ 99%)
R4	Tempo de inspeção $<$ 20 s	Parcialmente cumprido
R5	FAR	Não avaliado
R6	Arranque automático após falha	Cumprido (`systemd`, `Restart=on-failure`)
R7	Funcionamento contínuo em produção	Não validado

O requisito R4 é classificado como parcialmente cumprido: o tempo médio de análise completa observado é de aproximadamente 20 segundos no Raspberry Pi 5, em condições de teste, o que se aproxima do limite definido pela Versigent. No entanto, este valor não foi ainda validado no ambiente real de fábrica, onde condições de iluminação, distância e posicionamento da câmara podem diferir das utilizadas nos testes e onde a Fase 1, identificada como a mais lenta a validar, tem maior peso no tempo total. A validação deste requisito em ambiente de produção real é, por isso, identificada como prioridade de trabalho futuro.

O requisito R5 (FAR) não pôde ser avaliado devido à ausência de peças defeituosas (NOK) no conjunto de teste, conforme discutido na Secção 5.1.1. O requisito R7 não foi ainda validado, uma vez que o sistema não se encontra em funcionamento contínuo em ambiente de produção real.

5.4 Análise Crítica

A análise detalhada dos resultados da Fase 1 por classe revela uma assimetria relevante: a classe *black* apresenta uma precisão de 87.7%, inferior à da classe *red* (89.3%). Esta diferença sugere que os falsos positivos da classe *black* constituem a principal fonte de erro do modelo nesta fase, sendo provavelmente causados por sombras, reflexos escuros ou zonas de fundo escuro na imagem, que podem ser confundidos com o fio preto pelo modelo de deteção.

A Figura 5.2 ilustra este padrão de erro com exemplos reais capturados durante os testes: em todos os três casos, o modelo assinala a classe *black* sobre elementos que não correspondem ao fio, nomeadamente uma sombra no canto da imagem, um pano de fundo escuro, e a própria mão do operador. Este comportamento confirma que o problema não está associado a uma classe de objeto específica presente na cena, mas sim a qualquer região suficientemente escura da imagem, o que aponta diretamente para uma limitação de contraste e iluminação, e não para uma falha na arquitetura do modelo.

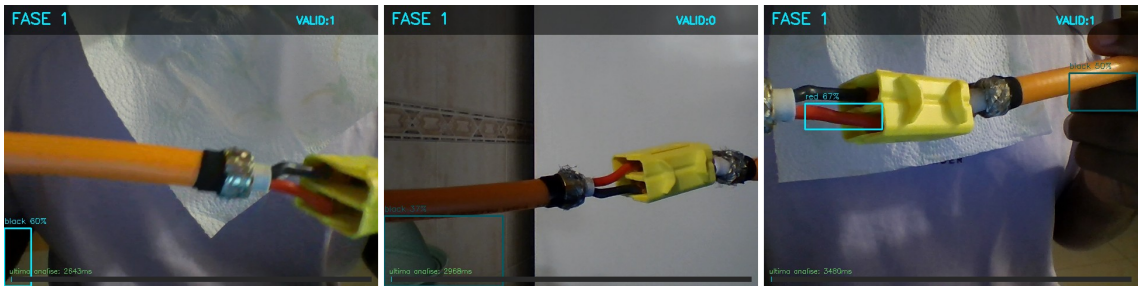


Figura 5.2: Exemplos de falsos positivos da classe *black* na Fase 1, causados por sombras e fundo escuro

Este resultado é consistente com os problemas de qualidade de imagem identificados durante a preparação do dataset (Secção 3.4), nomeadamente a presença de sombras nas capturas da linha de produção. Os falsos positivos observados durante os testes de validação foram, de forma geral, atribuídos a condições de fundo e iluminação da imagem capturada, não sendo exclusivos da Fase 1 reforçando a importância de, no futuro, se ampliar o dataset de treino com maior diversidade de condições ambientais em todas as fases do pipeline.

Para além da natureza da tarefa, a diferença de precisão entre fases pode também estar relacionada com a dimensão do dataset de treino disponível: a Fase 1, com a precisão mais baixa (88.5%), foi treinada com o menor número de imagens (401), enquanto as Fases 2 e 3, com precisões próximas de 98.5%, dispuseram de datasets progressivamente maiores (439 e 495 imagens, respetivamente). Não é possível isolar com certeza o peso relativo de cada fator complexidade da tarefa e volume de dados , mas ambos apontam na mesma direção e sugerem que um dataset mais robusto para a Fase 1 é uma via de melhoria a explorar.

O valor de *recall* observado nas três fases (93–94%) merece também uma leitura cuidada: num contexto de controlo de qualidade industrial, significa que, em média, cerca de 6 a 7% das peças corretas podem não ser reconhecidas como tal pelo sistema. Este tipo de erro o falso negativo não permite concluir nada sobre a taxa de não conformidades que passariam despercebidas (FAR), uma vez que o conjunto de teste não inclui peças defeituosas; reflete antes o risco de rejeições ou repetições desnecessárias de peças boas (FRR).

Em termos gerais, os resultados confirmam o sucesso da migração para o YOLO26m nas três fases, com a Fase 1 a apresentar a maior margem de melhoria não por limitação da arquitetura do modelo, mas por uma combinação de maior complexidade visual da tarefa e um dataset de treino comparativamente mais pequeno.

6. Conclusão

Este capítulo encerra o relatório com uma reflexão sobre as principais lições aprendidas ao longo do desenvolvimento do WireVision, identifica oportunidades de trabalho futuro e apresenta as considerações finais sobre o projeto no seu conjunto.

6.1 Lições Aprendidas

O desenvolvimento do WireVision permitiu retirar diversas lições relevantes, tanto ao nível da arquitetura de software como da engenharia de sistemas de visão por computador aplicados a um contexto industrial real.

Ao nível arquitetural, a experiência com a contenção do *Global Interpreter Lock* (GIL) do Python, causada pela execução da inferência dos modelos YOLO em threads dentro do mesmo processo da interface gráfica, evidenciou a importância de isolar tarefas computacionalmente intensivas em processos independentes, especialmente em aplicações que necessitam de manter uma interface responsiva em simultâneo com processamento pesado.

A investigação do bloqueio total observado na Fase 3, seguido de recuperação rápida, demonstrou também a importância de uma abordagem sistemática de diagnóstico: o que poderia ter sido erradamente atribuído a um número insuficiente de *workers* de inferência revelou-se, após monitorização adequada da temperatura do processador, ser um caso de *throttling* térmico, resultante da ausência de sistema de arrefecimento ativo no hardware utilizado.

Do ponto de vista da preparação de dados, a seleção manual e criteriosa das imagens de treino, com exclusão de casos problemáticos (fios não visíveis, sombras, borrões), confirmou-se como um fator determinante para a qualidade dos modelos treinados um investimento de tempo que se traduziu diretamente em modelos mais fiáveis, como demonstrado pelos resultados de validação apresentados no Capítulo 5.

Por fim, a evolução dos modelos utilizados (YOLOv5x YOLOv8x YOLO26m) ilustrou como a escolha da arquitetura de deteção de objetos deve ser revisitada ao longo do tempo, à medida que novas versões mais eficientes se tornam disponíveis, especialmente em sistemas com restrições de hardware como o Raspberry Pi 5.

6.2 Trabalho Futuro

Com base nos resultados obtidos e nas limitações identificadas ao longo deste projeto, são propostas as seguintes linhas de trabalho futuro:

- **Validação com peças defeituosas reais (NOK)**: o conjunto de teste utilizado neste trabalho contém exclusivamente peças corretas, o que impossibilitou o cálculo da *False Acceptance Rate* (FAR) a métrica mais crítica para um sistema de controlo de qualidade industrial, por medir o risco de peças defeituosas passarem despercebidas. Recomenda-se a constituição de um conjunto de teste com peças NOK reais e o cálculo explícito de FAR e FRR como validação prioritária antes de uma utilização autónoma do sistema em produção;
- **Investigação dos falsos positivos da classe *black* na Fase 1**: conforme identificado na análise crítica dos resultados (Secção 5.4), a classe *black* apresenta a precisão mais baixa entre as classes avaliadas, um resultado agravado pela exclusão sistemática de casos com sombras durante a construção do dataset de treino (Secção 3.4). Recomenda-se distinguir claramente, em trabalho futuro, entre imagens verdadeiramente inválidas (borrão extremo, objeto totalmente invisível) a excluir do dataset e casos difíceis mas representativos da produção real (sombras, fundo escuro, reflexos, mãos do operador) a incluir tanto no treino como no teste, de forma a melhorar a robustez do modelo nesta fase;
- **Solução de arrefecimento ativo para o Raspberry Pi 5**: dado o problema de *throttling* térmico identificado (Secção 4.3.1), recomenda-se a avaliação e implementação de uma solução de arrefecimento ativo (por exemplo, o *Active Cooler* oficial), de forma a manter o desempenho do sistema estável ao longo de todo o ciclo de inspeção, sem o bloqueio total observado atualmente na Fase 3;
- **Sistema integrado de gestão da informação gerada**: o âmbito original do projeto previa a elaboração de um sistema para gerir a informação gerada pelo processo de controlo de qualidade por visão. Esta componente representa uma oportunidade de expansão futura do WireVision, nomeadamente através do armazenamento estruturado do histórico de inspeções (por exemplo, em base de dados), possibilitando a geração de relatórios e estatísticas de produção ao longo do tempo;

- **Validação contínua em ambiente de produção**: recomenda-se a manutenção de um processo regular de validação dos modelos contra novas imagens de fábrica, de forma a monitorizar a fiabilidade do sistema ao longo do tempo e identificar atempadamente eventuais desvios de desempenho (*model drift*).
- **Comparação experimental direta entre gerações YOLO no sistema WireVision**: a redução de custo computacional reportada entre o YOLOv8x e o YOLO26m baseia-se na diferença de parâmetros dos modelos, não numa medição experimental direta no dataset e hardware do WireVision, uma vez que os modelos e métricas das versões anteriores (YOLOv5x, YOLOv8x) não foram preservados. Recomenda-se, em trabalho futuro, o retreino destas arquiteturas sobre o mesmo dataset e a medição sistemática de precisão, recall, mAP, latência no Raspberry Pi 5 e consumo de memória, de forma a validar experimentalmente esta comparação;
- **Controlo e documentação do ambiente de captura**: dado que os principais problemas identificados (sombras, reflexos, falsos positivos da classe *black*) estão diretamente relacionados com condições de iluminação e captura, recomenda-se a normalização e documentação completa do ambiente de aquisição modelo de câmara, distância, campo de visão, e configuração fixa de exposição e balanço de brancos como via de melhoria com potencial impacto superior ao simples aumento do dataset de treino;

6.3 Considerações Finais

O WireVision cumpre o objetivo geral proposto para este projeto: o desenvolvimento de um sistema de controlo de qualidade, baseado em visão por computador, para múltiplas fases do processo de construção de cabos na indústria automóvel. As três fases de inspeção implementadas — deteção de cor dos fios, verificação de separação dos fios e verificação de isolamento — foram validadas com resultados consistentes com os requisitos de um ambiente de produção industrial real.

A arquitetura desenvolvida, assente em subprocessos isolados de inferência e em modelos YOLO26m otimizados para hardware de baixo consumo, demonstra que é possível implementar sistemas de visão por computador fiáveis e eficientes em plataformas de custo reduzido, como o Raspberry Pi 5, sem necessidade de GPU dedicada. O projeto representou ainda uma oportunidade de aplicação prática de conhecimentos de visão por computador e engenharia de software num contexto empresarial real, em articulação com a Versigent, contribuindo para uma solução concreta com impacto direto na qualidade do processo produtivo.

Referências Bibliográficas

1. REDMON, Joseph; DIVVALA, Santosh; GIRSHICK, Ross; FARHADI, Ali — You Only Look Once: Unified, Real-Time Object Detection. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

2. REDMON, Joseph; FARHADI, Ali — YOLO9000: Better, Faster, Stronger. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

3. REDMON, Joseph; FARHADI, Ali — YOLOv3: An Incremental Improvement [em linha]. arXiv:1804.02767, 2018. [Consult. 2026]. Disponível em: `https://arxiv.org/abs/1804.02767`.

4. JOCHER, Glenn — Ultralytics YOLOv5 [em linha]. GitHub, 2020. [Consult. 2026]. Disponível em: `https://github.com/ultralytics/yolov5`.

5. JOCHER, Glenn; CHAURASIA, Ayush; QIU, Jing — Ultralytics YOLOv8 [em linha]. GitHub, 2023. [Consult. 2026]. Disponível em: `https://github.com/ultralytics/ultralytics`.

6. ULTRALYTICS — Ultralytics YOLO26 [em linha]. Documentação oficial, 2026. [Consult. 2026]. Disponível em: `https://docs.ultralytics.com/models/yolo26`.

7. JOCHER, Glenn; QIU, Jing; LIU, Mengyu; LYU, Shuai; AKYON, Fatih Cagatay; KALFAOGLU, Muhammet Esat — Ultralytics YOLO26: Unified Real-Time End-to-End Vision Models [em linha]. arXiv:2606.03748, 2026. [Consult. 2026]. Disponível em: `https://arxiv.org/abs/2606.03748`.

8. LIN, Tsung-Yi; MAIRE, Michael; BELONGIE, Serge; et al. — Microsoft COCO: Common Objects in Context. *European Conference on Computer Vision (ECCV)*, 2014.

9. TZUTALIN — LabelImg [em linha]. GitHub, 2015. [Consult. 2026]. Disponível em: `https://github.com/tzutalin/labelImg`.

10. DOCKER INC. — Docker Documentation [em linha]. [Consult. 2026]. Disponível em: `https://docs.docker.com`.

11. PASZKE, Adam; GROSS, Sam; MASSA, Francisco; et al. — PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

12. OPENCV TEAM — OpenCV Documentation [em linha]. [Consult. 2026]. Disponível em: `https://docs.opencv.org`.

13. STREAMLIT INC. — Streamlit Documentation [em linha]. [Consult. 2026]. Disponível em: `https://docs.streamlit.io`.

14. RASPBERRY PI FOUNDATION — Raspberry Pi 5 [em linha]. [Consult. 2026]. Disponível em: `https://www.raspberrypi.com/products/raspberry-pi-5/`.

15. POETTERING, Lennart; SYSTEMD CONTRIBUTORS — systemd System and Service Manager [em linha]. [Consult. 2026]. Disponível em: `https://systemd.io`.

16. RASPBERRY PI LTD — Heating and Cooling Raspberry Pi 5 [em linha]. 2023. [Consult. 2026]. Disponível em: `https://www.raspberrypi.com/news/heating-and-cooling-ras`

17. HÜTTEN, Nils; ALVES GOMES, Miguel; HÖLKEN, Florian; ANDRICEVIC, Karlo; MEYES, Richard; MEISEN, Tobias — Deep Learning for Automated Visual Inspection in Manufacturing and Maintenance: A Survey of Open-Access Papers. *Applied System Innovation*, 2024, 7(1), 11.

18. WU, W.; LI, Q. — Machine Vision Inspection of Electrical Connectors Based on Improved YOLO v3. *IEEE Access*, 2020, 8, 166184–166196.

19. CHEN, Ming; LIU, Yuqing; WEI, Xing; ZHANG, Zichen; GAIDAI, Oleg; SUI, Hengshou; LI, Bin — PO-YOLOv5: A Defect Detection Model for Solenoid Connector Based on YOLOv5. *PLoS ONE*, 2024, 19(1), e0297059.

20. OKANO, Marcelo T.; LOPES, William Aparecido Celestino; RUGGERO, Sergio Miele; VENDRAMETTO, Oduvaldo; FERNANDES, João Carlos Lopes — Edge AI for Industrial Visual Inspection: YOLOv8-Based Visual Conformity Detection Using Raspberry Pi. *Algorithms*, 2025, 18(8), 510.