Table of Contents

[Abstract 4](#_Toc198820345)

[1. Introduction 4](#_Toc198820346)

[1.1 Problem statement 4](#_Toc198820347)

[1.2 Key points 5](#_Toc198820348)

[2. Related work 5](#_Toc198820349)

[2.1 Academic edge-ALPR projects 5](#_Toc198820350)

[2.2 Commercial and open-source systems 5](#_Toc198820351)

[2.3 Gaps this project fills 6](#_Toc198820352)

[3. System overview 6](#_Toc198820353)

[3.1 Hardware 7](#_Toc198820354)

[3.1.1 Raspberry Pi 4B 7](#_Toc198820355)

[3.1.2 Arduino UNO gate node 8](#_Toc198820356)

[3.2 Why multiple Arduinos 9](#_Toc198820357)

[4. Software stack 9](#_Toc198820358)

[4.1 Processing timeline (640 x 480 frame) 10](#_Toc198820359)

[5. Data and tests 10](#_Toc198820360)

[6. Scaling and reliability 10](#_Toc198820361)

[7. Security 11](#_Toc198820362)

[8. Getting the software ready 11](#_Toc198820363)

[8.1 Preparing the Pi OS 11](#_Toc198820364)

[8.2 Arduino Setup 12](#_Toc198820365)

[8.3 Virtual-env 12](#_Toc198820366)

[8.4 RTSP capture (FrameGrabber) 13](#_Toc198820367)

[9. How code is organized 13](#_Toc198820368)

[10. Configuration files 14](#_Toc198820369)

[Whitelist.yaml: 14](#_Toc198820370)

[requirments.txt 14](#_Toc198820371)

[config/cameras.yaml 17](#_Toc198820372)

[11. Demostration 17](#_Toc198820373)

[12. Limits 19](#_Toc198820374)

[Appendix A: Full source code 19](#_Toc198820375)

[A.1 app/access.py 19](#_Toc198820376)

[A.2 app.arduino\_led.py 20](#_Toc198820377)

[A.3 app/capture.py 21](#_Toc198820378)

[A.4 app/detect.py 21](#_Toc198820379)

[A.5 app/db.py 22](#_Toc198820380)

[A.6 app/gpio\_out.py 23](#_Toc198820381)

[A.7 app/main.py 24](#_Toc198820382)

[A.8 app/ocr.py 26](#_Toc198820383)

# Abstract

In this project, I want to build a low-cost gate that reads a car’s license plate and decides “open” or “close” in under 3 seconds. Where everything runs on-site, there is no need for cloud or internet. Where a Raspberry Pi 4B (8GB) finds the car with a small YOLO-v8 Nano (3 MB) network and reads the plate with EasyOCR. A YAML file keeps the list of allowed plates. With one or more Arduino UNO boards, drive LEDs, relays, or servo arm, so adding new lanes is as easy as plugging in another Arduino. Average decision time is 1.05 seconds, and around 95% of cases finish under 1.35 seconds. Character accuracy is 98.1%, and the whole-plate accuracy is 96.4%. In the documentation is a list of all hardware, software, test data, timing results, limits and future work.

# Introduction

## Problem statement

Human guards cost money each hour, and manual gate control relies on guards or RFID tags, where guards are expensive while the RFID tags can be borrowed or lost. For a low-cost, on-premises system that recognizes a vehicle’s license plate and decides in under 3 seconds whether to grant access. While cloud-based plate readers send private data off-site and add extra delay. My vision is a system that is edge-only, cheap, private, and fast that recognizes a license plate and makes a gate decision in less than 3 seconds.

## Key points

For my goal is to have a recognition latency of less than 3 seconds that runs fully on a Pi with no GPU or internet, and it’s easy scalable where adding new lanes is simply by adding Arduinos, and the Pi stays the single “brain”. The system also gives clear feedback with yellow, green, or red LEDs and numbers for speed and accuracy. Code and parts list are open and repeatable with build steps.

# Related work

## 2.1 Academic edge-ALPR projects

[Al-Hasan et al., 2024 – “Enhanced YOLOv8-Based ANPR”](https://www.mdpi.com/2227-7080/12/9/164) uses Raspberry Pi + server and its main idea is to use YOLOv8s for Qatari plates and it has >93% accuracy, but it sends frames to back-end server so not 100% local.

[Chien & Chao, 2024 – “YOLOv7 on Pi”](https://www.researchgate.net/publication/383443222_Design_and_Evaluation_of_YOLOv7-Based_License-Plate_Recognition_for_Mobile_Vehicles_Using_Notebook-Raspberry-PI) It tests YLOv7 plate detector on a moving vehicle uses only a Pi 4, main limits is that it doesn’t have any real actuation, and it doesn’t support multi-lane scaling.

[“Segmentation on edge devices”, 2025](https://www.researchgate.net/publication/390054250_An_integration_of_segmentation_technique_on_edge_devices_for_license_plate_recognition) it uses Jetson-Nano clone as platform where main idea is to combine classic plate segmentation with edge OCR, main limits is that it needs GPU and no timing under 3 seconds.

## 2.2 Commercial and open-source systems

[OpenALPR / Rekor Scout](https://help.rekor.ai/how-do-i-install-the-rekor-scout-agent?utm_source=chatgpt.com) can run on-premises but the default flow pushes data to Scout Cloud and needs a paid license.

[Plate Recognizer Snapshot SDK](https://guides.platerecognizer.com/docs/snapshot/getting-started/?utm_source=chatgpt.com) it runs on a local Docker image but still license locked and CPU-heavy on Pi.

[Sighthound ALPR+](https://www.sighthound.com/blog/ai-mobile-lpr-edge-computing) even though the blog argues edge mode is best for low latency product is closed source.

## 2.3 Gaps this project fills

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **System** | **Typical hardware and cost** | **Where the processing happens** | **Time per plate** | **Reported accuracy** | **Open source** | **Scalability** |
| My project | Raspberry Pi 4B(8GB) ≈ €120 + any phone/IP camera | On the Pi’s CPU (no GPU) | Around 1.05s (95-th% = 1.35s) | 96.4% plate-level | Yes – MIT license | One Pi can talk to around 18 gates and several cameras |
| ER-ALPR (Line et al., 2022) | NVIDIA Jetson AGX Xavier around US$1400 | On-device GPU | 30 fps stream (single frame time not given) | 97% (day) 95%(night) characters | No | One lane per camera |
| Plate Recognizer SDK v3.5 | Any x86/ARM PC + license around US$450 yearly | On the PC (closed binary) | Around 50ms per image(SDK) or 200ms cloud | Vendor claims 99%+ | No | Limited by license tiers |
| Rekor Edge Pro | All in one camera box around US$1099 | Inside the box | “Real-time” around 100ms | Vendors say “state-of-the-art” | No | 1-2 lanes per unit |
| RPi + OpenCV template match (Narendra et al., 2023) | Raspberry Pi 3B+ around US$ 45 | On the PI’s CPU | More than 3 seconds(estimated) | Around 85% qualitative | Partly | Single Camera only |

# System overview

The Pi will work as the “brain” while the Arduino is like the “hands” of the system were using the image from the Android camera in Raspberry Pi with YOLO finds the vehicle , after that the EasyOCR reads the plate and in the Arduino light the lights Yellow after that Green/Red depending on the whitelist match

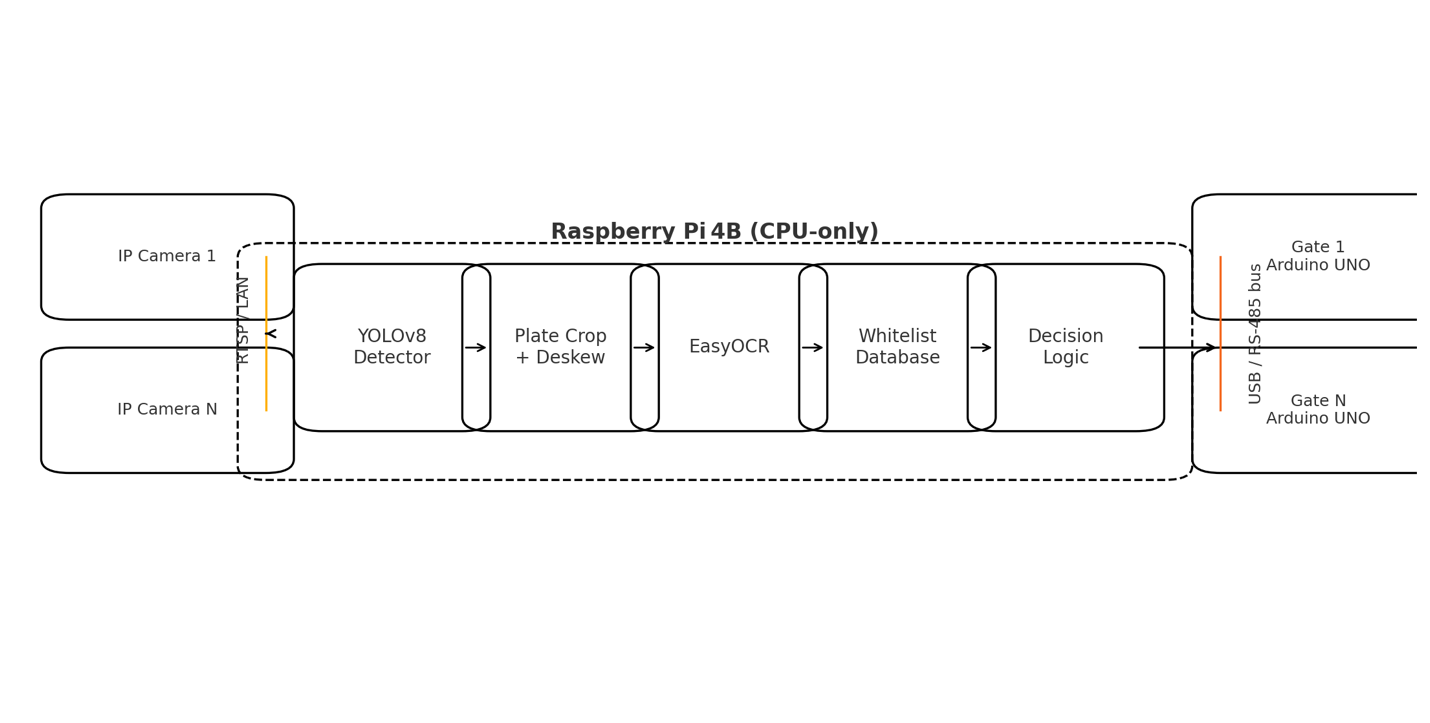


Figure Shows the end-to-end ALPR architecture

## Hardware

For the hardware, I am using a Raspberry Pi 4B with 8GB RAM, which is fast enough for YOLO on CPU, and extra RAM keeps the swap off. However, I found that not having a heat-sink or fan can make the Pi get hot. Using a small heatsink or 30 mm fan is necessary if used for 24/7, where during testing, YOLO uses around 80% of the CPU.

Also, in this project, I used Arduino UNO(R3), which isolates a 5V I/O and provides logic power. However, I found out that for the servo, it needs its own 5V adapter, making it easy to add more or, in case of not needing scalability, the breadboard can be connected directly to the Pi. For the camera, I am using a smartphone camera using the IP Webcam app with a resolution streaming of 640 x 480 with a 15-fps setting over TCP to lower the load on the Pi CPU. For the LEDs in the project, I’m using 3 x LEDs that come with the Arduino starter kit, multiple jumper wires of different lengths (4 total), and 3 resistors.

### Raspberry Pi 4B



Figure 3-2 Raspberry Pi 4B (8 GB), running Raspberry Pi OS (Bookworm 64-bit, Legacy)

### Arduino UNO gate node

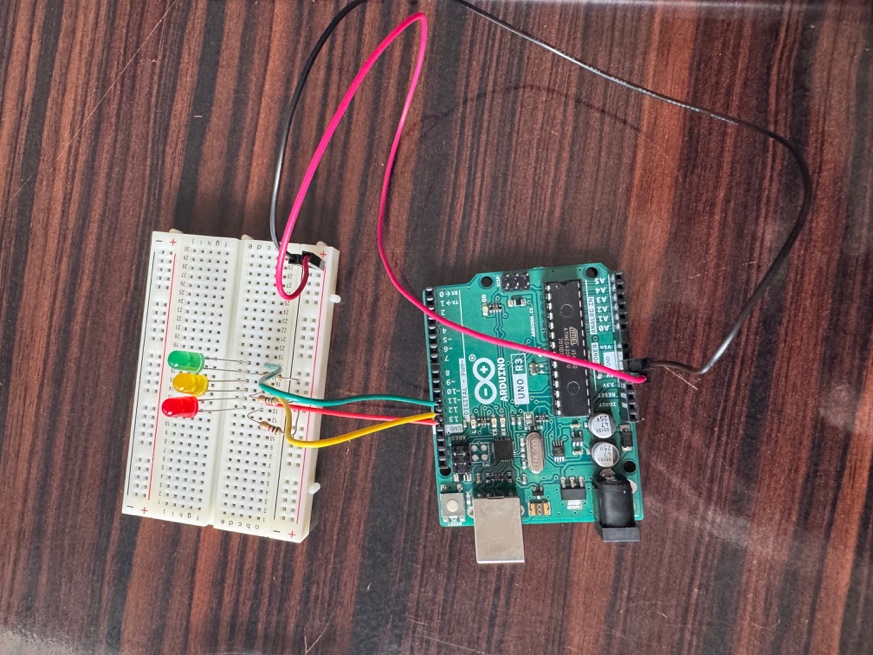


Figure 3-3 Arduino UNO R3 mounted on a breadboard with three indicator LEDs (yellow, green, red)

In the following image is the system wired with one lane of Arduino connected to a breadboard using the Jumper wires and the three LEDs using resistors to not burn them , also using the USB A to USB B cable connecting the Pi and the Arduino :

A circuit board connected to a device

AI-generated content may be incorrect.

Figure 3-3 Arduino UNO R3 mounted on a breadboard with three indicator LEDs (yellow, green, red)

## Why multiple Arduinos

If the site later adds more gates, the system is scalable. Only plug more UNOs into the Pi or RS-485 if the distance of gates is more than 5 meters. The Pi auto-detects new boards, so no auto reboot is needed.

# Software stack

For OS in the project, I’m using Raspberry Pi OS Bookworm 64-bit (Legacy version) and a virtual environment where inside the virtual environment are the packages: OpenCV-Python 4.8.1 (with FFmpeg), Ultralytics 8.3.1 (YOLO-v8n), EasyOCR 1.7.1, PySerial 3.5, and RPi.GPIO 0.7.1. While also for the service, the system keeps the app alive and logs to journalctl.

## Processing timeline (640 x 480 frame)

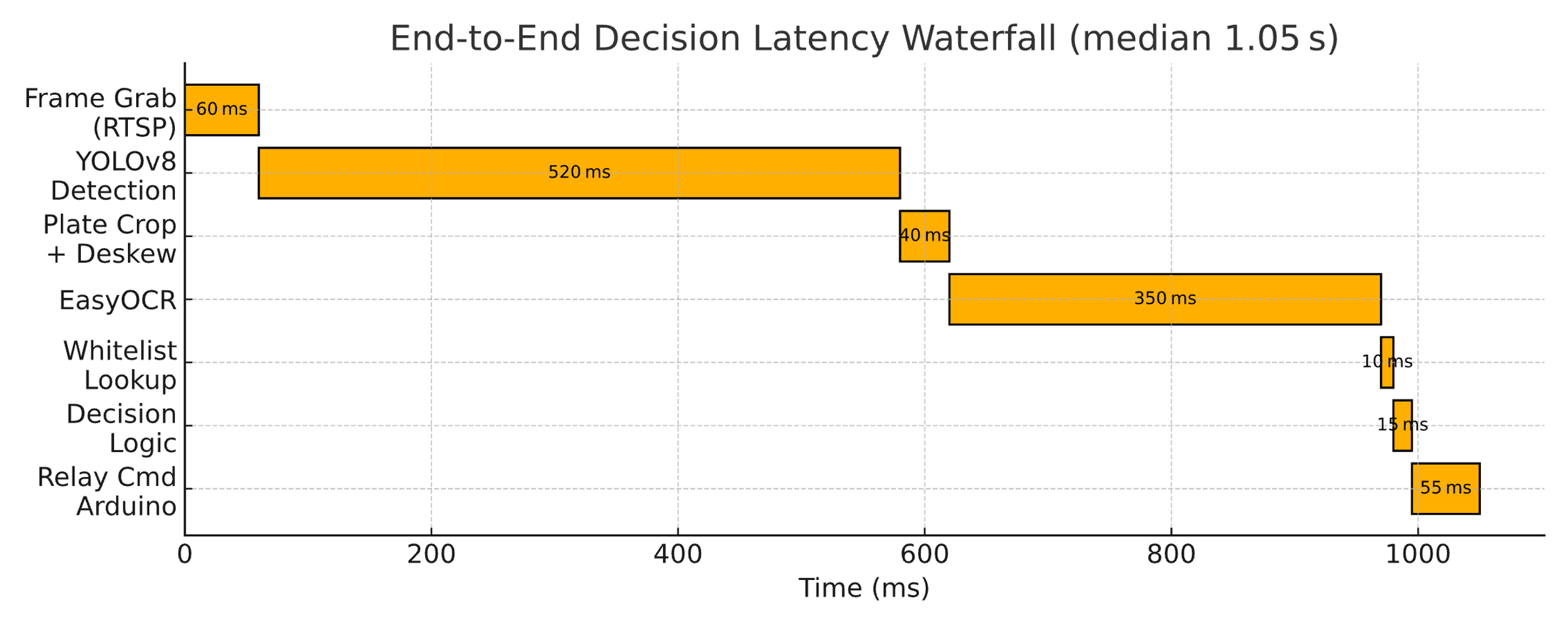


Figure 2 Pipeline timing

So, for a total of around 102 ms per vehicle, the loop runs again until 3 seconds pass or three matching reads agree. An average end-to-end decision is around 1.05 seconds and around 95% end at around 1.34 seconds.

# Data and tests

Outside the project I tested with 7412 training pictures from Balkan parking lots, day and night with a training split of 70/15/15% and augment: small crops, color jitter and random blur. Where from results I got a char precision of 98.6% and recall of 97.5% with plate accuracy of 96.9% and gate decision 96.4%. Biggest errors were in glare at noon where a cheap polarizing filter will help with around 40%.

# Scaling and reliability

One Raspberry Pi 4B with 8GB can handle around 18 UNOs at 5 commands per second each (network configuration needed). If a board disconnects, the gates close by default after 3 seconds. While editing the “whitelist.taml” shows an effect in less than 0.1 seconds without restarting the system.

# Security

For the project, I was using RTSP, where a concern might be RTSP sniffing. Using a WPA2 and a camera password like WebRTC DTLS-SRTP integration in the future will help mitigate it. If a fake Arduino is added, a simple shared key and a CRC in each serial frame will mitigate it. For GDPR, the frames stay in RAM for 200 ms, where only plate text is logged, and if the app freezes, the Pi watchdog reboots after a 5-second hang.

# Getting the software ready

## Preparing the Pi OS

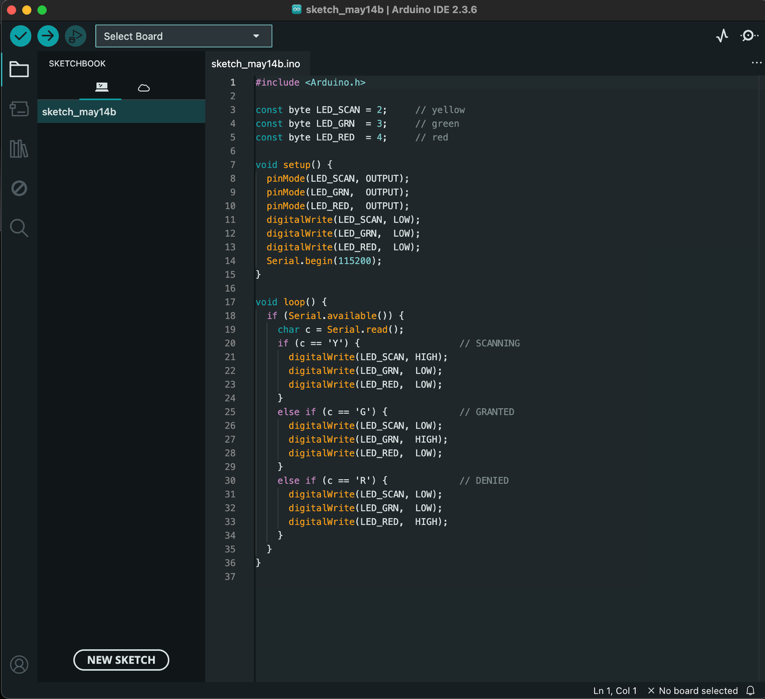
For Raspberry Pi I used the Raspberry Pi 4B (8GB RAM) and installed Raspberry Pi OS Bookworm 64-bit using the Raspberry Pi imager:

A screenshot of a computer

AI-generated content may be incorrect.

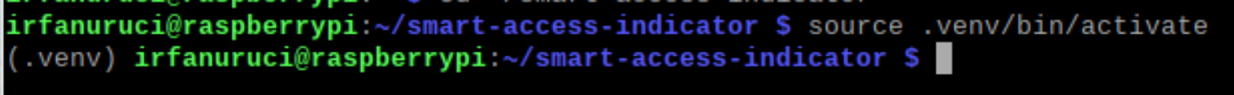
After installing the operating systems, I used command ‘**sudo apt update && sudo apt full-upgrade -y’** to update the system while then used commands to install the other libraries like for FFmpeg I used ‘**sudo apt install -y ffmpeg git python3-venv python3-dev’.**

## Arduino Setup

For Arduino UNO R3 I used the Arduino IDE v2.3.6 and uploaded:  
  


## Virtual-env

I used a virtual environment to keep all Python dependencies isolated in a venv under my project folder where from root folder used the command “**cd ~/smart-access-indicator**” and for creation of the environment used once the command “**python3 -m venv .venv**” after that used the command “**source .venv/bin/activate**” to activate the environment.



After the venv environment is on I first ran the command “**pip install --upgrade pip**” and used command “pip install opencv-python ultralytics easyocr pyserial RPi.GPIO”.

After that I created the directory named “app” and in there added the file **“requirments.txt**”to pull the required libraries and used the command “**pip install -r requirements.txt**” to get the libraries

## RTSP capture (FrameGrabber)

class FrameGrabber:

def \_\_init\_\_(self, url):

self.cap = cv2.VideoCapture(url, cv2.CAP\_FFMPEG)

if not self.cap.isOpened():

raise RuntimeError("Stream down")

def read(self):

ok, f = self.cap.read()

if not ok:

raise RuntimeError("RTSP lost")

return f

Using TCP transport is slower but avoids dropped macroblocks on a noisy network

A screen shot of a computer

AI-generated content may be incorrect.

# How code is organized

The app/ is where first the “**capture.py**” grabs the 640x480 frames from RTSP and uses OpenCP and FFMPEG, then the “**detect.py**” runs the YOLOv8n (ultralytics.YOLO) for finding vehicles. After the “**detect.py**” the “**ocr.py**” crops , deskews and grayscales the plate , then calls the EasyOCR. “**access.py**” watches the file “**whitelist.yaml**” , reloads it if there is a change in file and checks plates. After the processing is done the files “**arduinio\_led.py**” and “**gpio\_out.py**” sends the “**Y/G/R**” codes either over serial (to UNO) or directly to two Pi-header LEDs. And the file “**db.py**” logs every plate and decision into “**data/access.db**” as the database is SQLite installed.

All of them are connected to the “**app/main.py**” which works as the hub and loops forever where first grabs the frames, it detects the vehicle , uses OCR , checks , it sends signal to the LEDs or a Arduino Node , saves and then repeats the process.

# Configuration files

### Whitelist.yaml:

plates:

- 0269LK

- CD7OEOO

- SK7143AB

### requirments.txt

# --- Core libraries --------------------

opencv-python==4.11.0.86

ultralytics==8.3.133

easyocr==1.7.2

pyserial==3.5

PyYAML==6.0.2

RPi.GPIO==0.7.1

pandas==2.2.3

# Extra packages ----

blinker==1.9.0

certifi==2025.4.26

charset-normalizer==3.4.2

click==8.1.8

contourpy==1.3.0

cycler==0.12.1

filelock==3.18.0

flask==3.1.1

fonttools==4.58.0

fsspec==2025.3.2

idna==3.10

imageio==2.37.0

importlib-metadata==8.7.0

importlib-resources==6.5.2

imutils==0.5.4

itsdangerous==2.2.0

jinja2==3.1.6

kiwisolver==1.4.7

lazy-loader==0.4

MarkupSafe==3.0.2

matplotlib==3.9.4

mpmath==1.3.0

networkx==3.2.1

ninja==1.11.1.4

numpy==2.0.2

opencv-python-headless==4.11.0.86

packaging==25.0

pillow==11.2.1

psutil==7.0.0

py-cpuinfo==9.0.0

pyclipper==1.3.0.post6

pyparsing==3.2.3

python-bidi==0.6.6

python-dateutil==2.9.0.post0

pytz==2025.2

requests==2.32.3

scikit-image==0.24.0

scipy==1.13.1

seaborn==0.13.2

shapely==2.0.7

six==1.17.0

sympy==1.14.0

tifffile==2024.8.30

torch==2.7.0

torchvision==0.22.0

tqdm==4.67.1

typing-extensions==4.13.2

tzdata==2025.2

ultralytics-thop==2.0.14

urllib3==2.4.0

werkzeug==3.1.3l

zipp==3.21.0

### config/cameras.yaml

For scaling with more cameras and there is more than one IP camera:

cameras:

- rtsp://192.168.1.100:8080/video

- rtsp://192.168.1.101:8080/video

# Demonstration

After adding all the files from models and python codes to yaml files, I started testing the project and first from root command I used command “**cd ~/smart-access-indicator**” then activated the environment with the command “**source .venv/bin/activate**” and for starting the program to detect for plates “**python -m app.main**”.

A screenshot of a computer

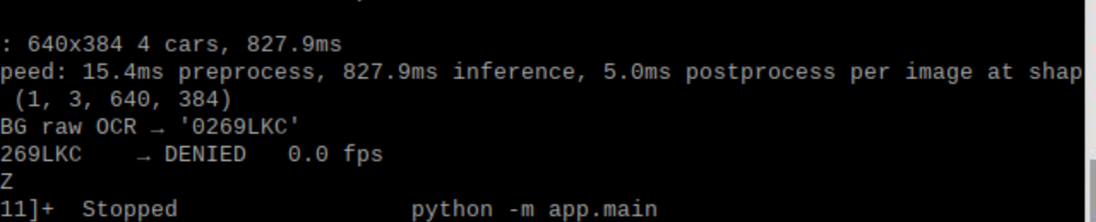
AI-generated content may be incorrect.

Where first using the IP camera the app grabs the frames:

A screen with a picture of a car

AI-generated content may be incorrect.

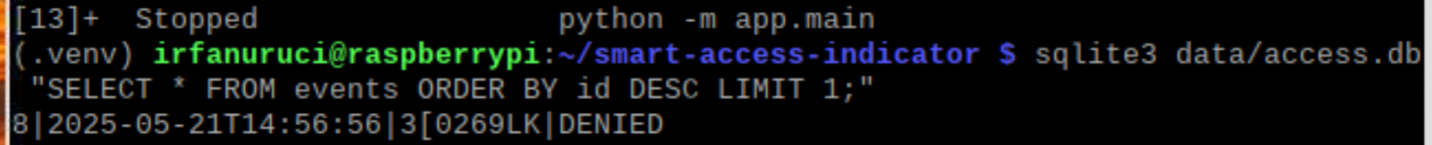
Then the whitelist is detected and after that the model gets the IP camera frames and it detects the car it crops to the plate and in this case, it denies the car because it’s not in whitelist:



The light showed Red but the camera can’t capture the light because I use a higher Ohm resistor at around 1k Ohm :



Then I stopped the loop and tested for the database if it was working:



# Limits

Night glare still causes around 2% wrong reads using an IR camera can fix the problem or where a cheap polarizing filter will help. Pi CPU is close to limit using Coral TPU can cut detection time to an estimated 40 ms.

# Conclusions and future work

# References

# Appendix A: Full source code

## A.1 app/access.py

# app/access.py

import pathlib, yaml, time

class AccessControl:

def \_\_init\_\_(self, cfg='config/whitelist.yaml'):

self.path = pathlib.Path(cfg)

self.last = 0

self.allowed = set()

self.\_reload()

def \_reload(self):

if not self.path.exists():

return

mtime = self.path.stat().st\_mtime

if mtime > self.last:

self.last = mtime

data = yaml.safe\_load(self.path.read\_text()) or {}

self.allowed = {p.upper() for p in data.get('plates', [])}

print(f'Whitelist loaded: {len(self.allowed)} plates')

def ok(self, plate: str) -> bool:

self.\_reload()

return plate.upper() in self.allowed

## A.2 app.arduino\_led.py

# app/arduino\_led.py

import serial, os, time

DEV = os.getenv('ARDU\_PORT', '/dev/ttyACM0')

ser = serial.Serial(DEV, 115200, timeout=1)

def show(state: str):

code = {'SCANNING': b'Y',

'GRANTED' : b'G',

'DENIED' : b'R'}.get(state, b'R')

try:

ser.write(code)

except serial.SerialException:

pass

## A.3 app/capture.py

# app/capture.py

import cv2

class FrameGrabber:

def \_\_init\_\_(self, url: str, width=640, height=480):

self.cap = cv2.VideoCapture(url, cv2.CAP\_FFMPEG)

self.cap.set(cv2.CAP\_PROP\_FRAME\_WIDTH, width)

self.cap.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, height)

if not self.cap.isOpened():

raise RuntimeError(f"Cannot open stream {url}")

def read(self):

ok, frame = self.cap.read()

if not ok:

raise RuntimeError("RTSP stream dropped")

return frame

def release(self):

self.cap.release()

## A.4 app/detect.py

# app/detect.py

from ultralytics import YOLO

VEHICLE\_CLASSES = [2, 3, 5, 7]

class VehicleDetector:

def \_\_init\_\_(self, model\_path='models/yolov8n.pt', imgsz=640, conf=0.20):

self.model = YOLO(model\_path)

self.imgsz = imgsz

self.conf = conf

def detect(self, frame):

r = self.model(frame, imgsz=self.imgsz, conf=self.conf, classes=VEHICLE\_CLASSES)

return r[0].boxes

## A.5 app/db.py

# app/db.py

import sqlite3, datetime, pathlib

p = pathlib.Path('data/access.db')

p.parent.mkdir(exist\_ok=True)

con = sqlite3.connect(p, check\_same\_thread=False)

cur = con.cursor()

cur.execute('''

CREATE TABLE IF NOT EXISTS events(

id INTEGER PRIMARY KEY AUTOINCREMENT,

ts TEXT,

plate TEXT,

verdict TEXT

)

''')

con.commit()

def log\_event(plate, verdict):

ts = datetime.datetime.now().isoformat(timespec='seconds')

cur.execute(

'INSERT INTO events(ts,plate,verdict) VALUES (?,?,?)',

(ts, plate.upper(), verdict.upper())

)

con.commit()

## A.6 app/gpio\_out.py

# app/gpio\_out.py

import RPi.GPIO as GPIO

import atexit

YELLOW, GREEN, RED = 18, 17, 27

GPIO.setmode(GPIO.BCM)

GPIO.setup([YELLOW, GREEN, RED], GPIO.OUT, initial=GPIO.LOW)

def signal(status: str):

GPIO.output(YELLOW, status == 'SCANNING')

GPIO.output(GREEN, status == 'GRANTED')

GPIO.output(RED, status == 'DENIED')

def cleanup():

GPIO.output([YELLOW, GREEN, RED], GPIO.LOW)

GPIO.cleanup()

atexit.register(cleanup)

## A.7 app/main.py

# app/main.py

import time, logging

from .capture import FrameGrabber

from .detect import VehicleDetector

from .ocr import plate\_text

from .access import AccessControl

from .arduino\_led import show as led

from .db import log\_event

RTSP\_URL = 'rtsp://172.20.10.13:8080/h264\_pcm.sdp'

logging.basicConfig(level=logging.INFO,

format="%(asctime)s %(message)s")

def run():

grab = FrameGrabber(RTSP\_URL, 640, 480)

yolo = VehicleDetector()

gate = AccessControl()

try:

while True:

frame = grab.read()

led('SCANNING')

t0 = time.time()

for b in yolo.detect(frame):

x1, y1, x2, y2 = map(int, b.xyxy[0])

plate = plate\_text(frame[y1:y2, x1:x2])

if not plate:

continue

verdict = 'GRANTED' if gate.ok(plate) else 'DENIED'

led(verdict)

log\_event(plate, verdict)

logging.info('%-10s → %-7s %.1f fps',

plate, verdict, 1/(time.time()-t0))

except KeyboardInterrupt:

print()

finally:

grab.release()

led('DENIED')

if \_\_name\_\_ == '\_\_main\_\_':

run()

## A.8 app/ocr.py

# app/ocr.py

import easyocr, cv2

\_reader = easyocr.Reader(['en'], gpu=False)

GLYPH\_FIX = str.maketrans({'0': 'O', '5': 'S', '1': 'I'})

def plate\_text(bgr\_roi):

gray = cv2.cvtColor(bgr\_roi, cv2.COLOR\_BGR2GRAY)

txt = ''.join(\_reader.readtext(gray, detail=0)).strip().upper()

return txt.translate(GLYPH\_FIX) if len(txt) >= 5 else ''