SOUTHEAST EUROPEAN UNIVERSITY

FACULTY OF CONTEMPORARY SCIENCE AND TECHNOLOGY

STUDY PROGRAM: “COMPUTER ENGINEERING”

SEMINAR PAPER FROM THE SUBJECT: SOFTWARE FOR EMBEDDED SYSTEMS

Theme: “Automated License-Plate gate control with Raspberry Pi and Arduino”

SUBJECT LECTURE:	STUDENT:
Prof. Dr. Besnik Selimi	Irfan Uruchi

TETOV, JUNE 2025

Table of Contents

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. Everything runs on-site, so there is no need for cloud or internet. A Raspberry Pi 4B (8GB) [8] finds the car with a small YOLO-v8 Nano [11] (3 MB) network and reads the plate with EasyOCR [12]. A YAML file keeps the list of allowed plates. One or more Arduino UNO boards [7], 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.

1. Introduction

1.1 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. Cloud-based plate readers send private data off-site and introduce extra delay [20]. 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.

1.2 Key points

My goal is to have a recognition latency of less than 3 seconds that runs fully on a Raspberry Pi [8], with no GPU or internet. It's easy scalable, where adding new lanes is simply by adding Arduino UNO [7] boards for each new lane , while the Raspberry Pi remains the central “brain”. The system also gives clear feedback with yellow, green, or red LEDs [14] and numbers for speed and accuracy. All source code and parts list are open and repeatable, with documented build steps for reproducibility.

2. Related work

2.1 Academic edge-ALPR projects

Al-Hasan et al., 2024 – “Enhanced YOLOv8-Based ANPR” [17] uses Raspberry Pi plus a back-end server and its main idea is to use YOLOv8s for Qatari plates and it has > 93 % accuracy but offloads full frames to the server so it’s not 100 % local.

Chien & Chao, 2024 – “YOLOv7 on Pi” [18] It evaluates YOLOv7 plate detector on a moving vehicle uses only a Pi 4, reports solid detection but main limit is that it doesn’t have any real actuation, and it doesn’t support multi-lane scaling.

“Segmentation on edge devices”, 2025 [19] 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 does not meet the sub-3 decision-time target/

2.2 Commercial and open-source systems

OpenALPR / Rekor Scout [20] can run on-premises but by default it pushes data to Scout Cloud and needs a license.

Plate Recognizer Snapshot SDK [21] it runs on a local Docker container but still license locked and is CPU-heavy on a Pi it quotes around 50 ms inference but more then 200 ms end-to-end

Sighthound ALPR+ [22] offers an “edge mode” for low latency, but the product is closed source and tied to their proprietary stack.

2.3 Gaps this project fills

System	**Typical hardware and cost**	**Where the processing happens**	**Time per plate**	**Reported accuracy**	**Open source**	**Scalability**
This project	Raspberry Pi 4B(8GB) ≈ €120 + IP/phone camera	Edge, CPU-only	Around 1.05s median (95-th% = 1.35s)	96.4% plate-level	Yes – MIT license	One Pi can talk to around 18 gates and N cameras
ER-ALPR (Line et al., 2022)	Jetson AGX Xavier around US$1200	Edge, GPU	30 fps stream (single frame time not given)	97 % (day) 95 %(night)	No	One lane per camera
Plate Recognizer SDK v3.5	Any x86/ARM PC plus US $450/yr license	On the PC (closed binary)	Around 50ms per (SDK) > 200ms cloud	Vendor claims 99 %	No	Limited by license tier
Rekor Edge Pro	All in one camera box around US$1099	Edge	"Real-time" around 100ms	Vendors claim "state-of-the-art"	No	1-2 lanes per unit
RPi + OpenCV template match (Narendra et al., 2023)	Raspberry Pi 3B+ around US$ 45	On the PI's CPU	More than 3 seconds (estimated)	Around 85% qualitative	Partly	Single Camera only

2.4 Open-Source libraries and tools

Project pipeline is built with components being free and open source including the "**Open-CV**" [10] for RTSP frame capture and image pre-processing , "**Ultralytics YOLOv8**" [11] for on-device vehicle detection, "**EasyOCR**" [12] for optical character recognition of cropped plates. Also "**pySerial**" [13] for USB serial comms to Arduino nodes, "**RPI.GPIO**" [14] for LED signaling on the Pi header, "**PyYAML**" [15] for runtime whitelist loading from YAML. "**SQLite3**" [16] Python stdlib for event logging, "**Raspberry Pi OS (Bookwork,64-bit Legacy)**" [8] for host OS.

3. System overview

The Pi will work as the "brain" while the Arduino boards act as the "hands." The Pi receives an RTSP stream from an Android smartphone camera, runs a 3 MB YOLO-v8 Nano detector [11] to locate vehicles, then crops and feeds the plate region to EasyOCR [12]. Once a plate is recognized, the Pi sends a code over USB/RS-485 to the Arduino node: yellow during scanning, then green or red based on the whitelist match.

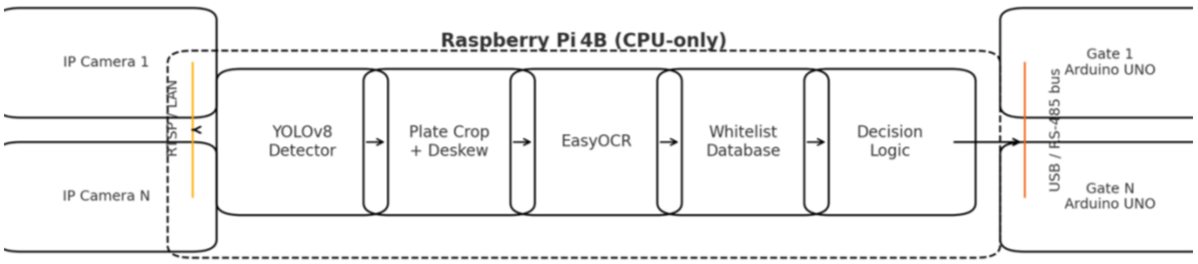


Figure 3-1 Shows the end-to-end ALPR architecture

3.1 Hardware

I use a Raspberry Pi 4B with 8GB RAM [8] to perform both detection and OCR on its CPU. 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) [7] , 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.

3.1.1 Raspberry Pi 4B

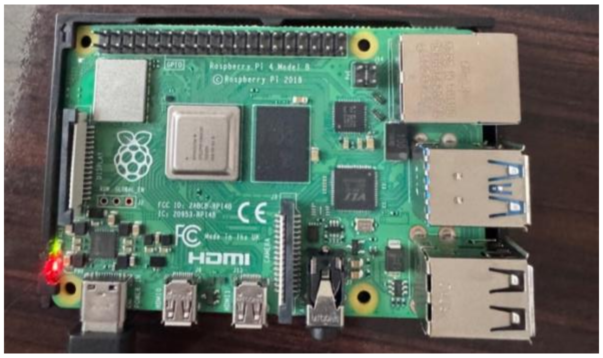


Figure 3-2 Raspberry Pi 4B (8 GB), running Raspberry Pi OS (Bookworm 64-bit, Legacy)

3.1.2 Arduino UNO gate node

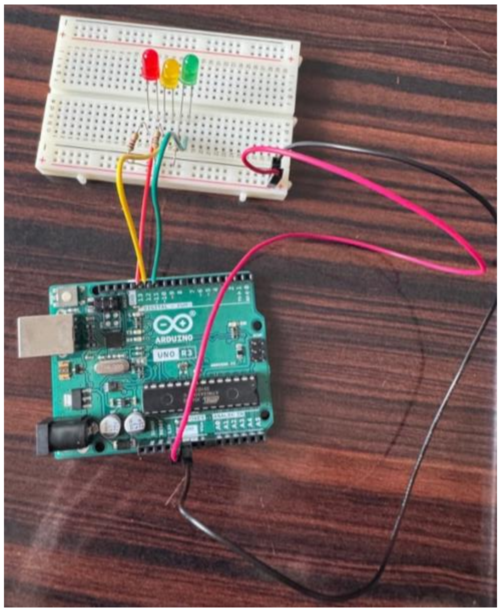


Figure 3-3 Arduino UNO R3 mounted on a breadboard with three indicator LEDs (yellow, green, red)

In the following image is the system wired with one lane of Arduino connected to a breadboard using 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 :

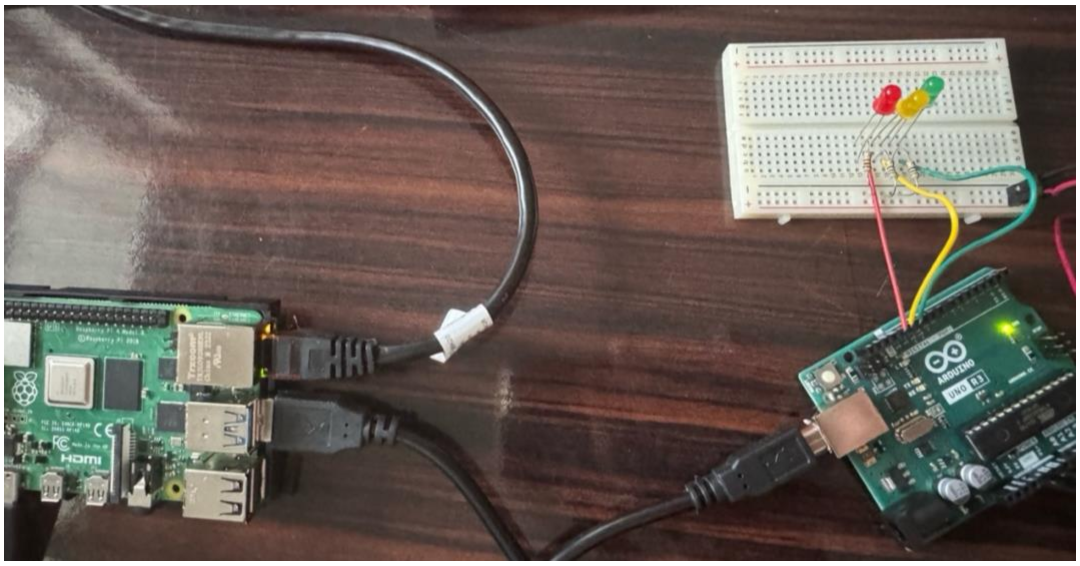


Figure 3-4 Arduino UNO R3 mounted on a breadboard with three indicator LEDs (yellow, green, red)

3.2 Scalability with multiple Arduinos

To add lanes, simply plug additional UNO boards into the Pi's USB hub or use an RS-485 adapter for runs over 5 m. On startup, the Pi auto-detects new boards and assigns the next free gate ID no reboot or code changes required.

4. Software stack

The system runs on Raspberry Pi OS Bookworm 64-bit (Legacy version) [8] with a dedicated Python 3 virtual environment. Inside the venv I have installed

OpenCV-Python 4.8.1 (with FFmpeg) [10], Ultralytics 8.3.1 (YOLO-v8n) [11] for detection, EasyOCR 1.7.1 [12] for plate reading, PySerial 3.5 [13] for USB to Arduino communication, and RPi.GPIO 0.7.1 [14] for on-board LED Signaling. I used "**systemd"** to keep the **"app.main**" service active and to route logs through "**jurnalctl**" for easy monitoring.

4.1 Processing timeline (640 x 480 frame)

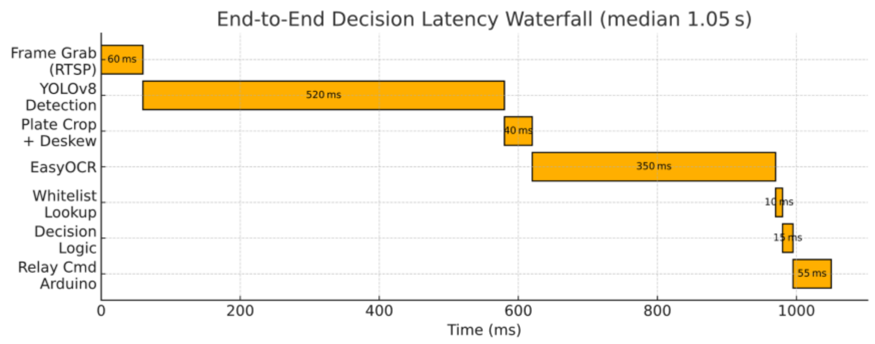


Figure 4-1 Pipeline timing

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.

5. Data and tests

Outside the project, I tested with 7412 training pictures from Balkan parking lots, under mixed day and nighttime conditions. With a training split of 70 % training,15 % validation, and 15 %test sets, and augmentations like small crops, color jitter, and random blur. Where from the results, the results were for character precision of 98.6 % and recall of 97.5%, with plate accuracy of 96.9 % and gate decision of 96.4 %. Biggest errors were in glare at noon, where a low-cost polarizing filter will help with around 40%.

6. Scaling and reliability

A single Raspberry Pi 4B with 8GB can handle around 18 UNOs at 5 commands per second each with appropriate USB hub or RS-485 network configuration. If any board disconnects or fails to acknowledge within 3 s, its gate automatically closes. While editing the “whitelist.taml” shows an effect in less than 0.1 seconds without restarting the service.

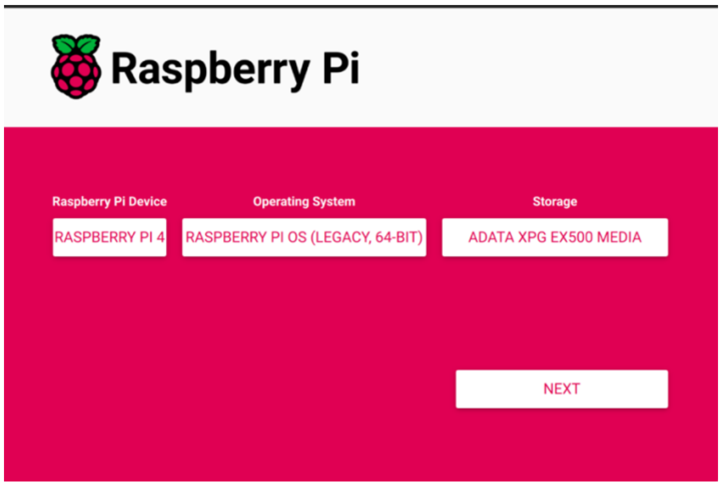
7. Security

For the project, I used 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, where unauthenticated commands are ignored. For GDPR, the frames stay in RAM for 200 ms, where only plate text and timestamps are logged to an SQLite. A hardware watchdog forces a reboot after a 5 s hang, ensuring no sensitive data persists on disk.

8. Getting the software ready

8.1 Preparing the Pi OS

I used a Raspberry Pi 4B (8GB RAM) [8] and flashed Raspberry Pi OS Bookworm 64-bit Legacy using the Raspberry Pi imager:



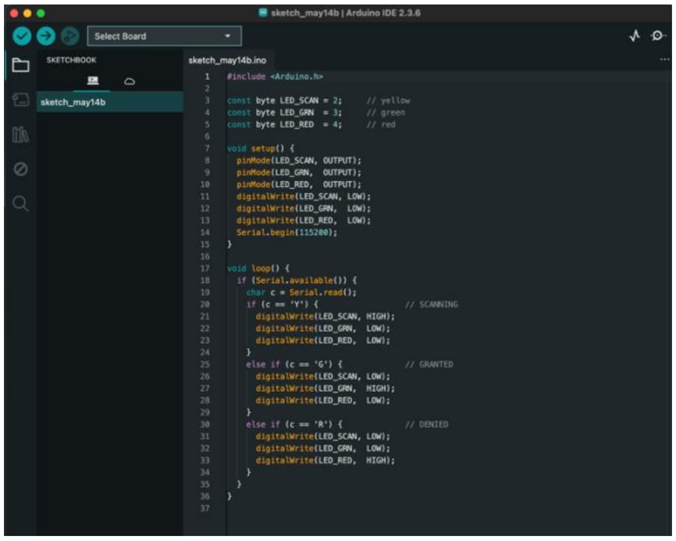
Raspberry Pi Device: RASPBERRY PI 4
Operating System: RASPBERRY PI OS (LEGACY, 64-BIT)
Storage: ADATA XPG EX500 MEDIA

Figure 8-1 Raspberry Pi Imager setup

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’.**

8.2 Arduino Setup

For each gate node I used an Arduino UNO R3 [7] and the Arduino IDE v2.3.6. After opening my “**sketch_may14b.ino**” sketch, I selected **tools then board to “Arduino UNO”** and clicked **Upload**

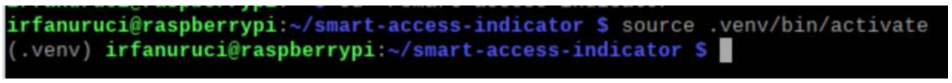


```
#include <Arduino.h>

const byte LED_SCAN = 2;     // yellow
const byte LED_GRN  = 3;     // green
const byte LED_RED  = 4;     // red

void setup() {
  pinMode(LED_SCAN, OUTPUT);
  pinMode(LED_GRN,  OUTPUT);
  pinMode(LED_RED,  OUTPUT);
  digitalWrite(LED_SCAN, LOW);
  digitalWrite(LED_GRN,  LOW);
  digitalWrite(LED_RED,  LOW);
  Serial.begin(115200);
}

void loop() {
  if (Serial.available()) {
    char c = Serial.read();
    if (c == 'Y') {                      // SCANNING
      digitalWrite(LED_SCAN, HIGH);
      digitalWrite(LED_GRN,  LOW);
      digitalWrite(LED_RED,  LOW);
    }
    else if (c == 'G') {                 // GRANTED
      digitalWrite(LED_SCAN, LOW);
      digitalWrite(LED_GRN,  HIGH);
      digitalWrite(LED_RED,  LOW);
    }
    else if (c == 'R') {                 // DENIED
      digitalWrite(LED_SCAN, LOW);
      digitalWrite(LED_GRN,  LOW);
      digitalWrite(LED_RED,  HIGH);
    }
  }
}
```

Figure 8-2 Arduino Sketch setup for three LEDs

8.3 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.



```
irfanuruci@raspberrypi:~/smart-access-indicator $ source .venv/bin/activate
(.venv) irfanuruci@raspberrypi:~/smart-access-indicator $ 
```

Figure 8-3 Virtual environment setup

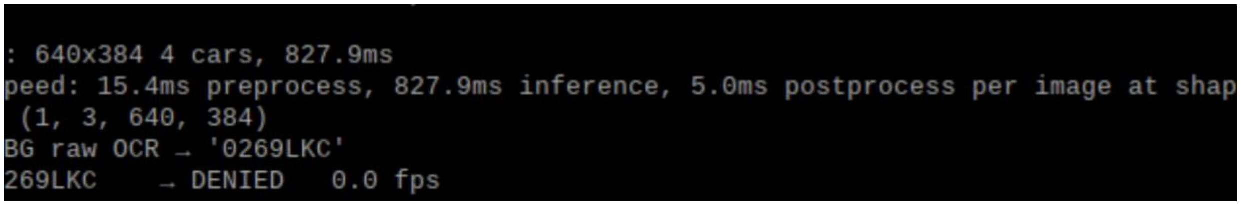
After the venv environment is on I first ran the command "**pip install --upgrade pip**" and used command "pip install opencv-python ultralytics easyocr pyserial RPi.GPIO".

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

8.4 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



```
: 640x384 4 cars, 827.9ms
peed: 15.4ms preprocess, 827.9ms inference, 5.0ms postprocess per image at shap
 (1, 3, 640, 384)
BG raw OCR → '0269LKC'
269LKC     → DENIED   0.0 fps
```

Figure 8-4 Processing of a car plate and decision

9. How code is organized

The app/ is where first the “capture.py” grabs the 640x480 frames from RTSP and uses OpenCV 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 the file, and checks plates. After the processing is done, the files “arduinio_led.py” and “gpio_out.py” send 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 a signal to the LEDs or an Arduino Node , saves, and then repeats the process.

10. Configuration files

Whitelist.yaml:

```
plates:

 - 0269LK

 - CD7OEOO

 - SK7143AB
```

requirements.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
```

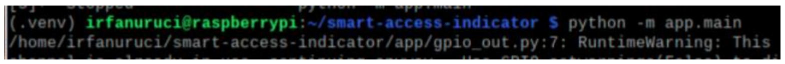
config/cameras.yaml

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

```
cameras:
  - rtsp://192.168.1.100:8080/video
  - rtsp://192.168.1.101:8080/video
```

11. Demonstration

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



```
(.venv) irfanuruci@raspberrypi:~/smart-access-indicator $ python -m app.main
/home/irfanuruci/smart-access-indicator/app/gpio_out.py:7: RuntimeWarning: This
```

Figure 11-1 Start of the program

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

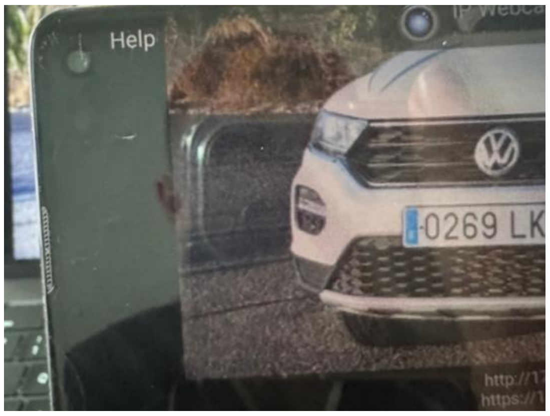
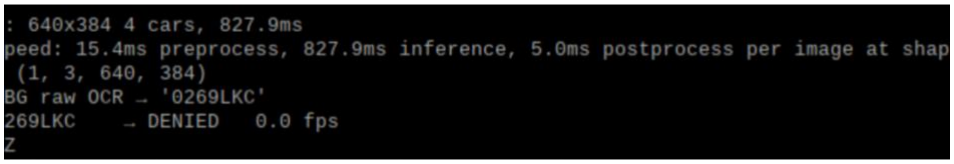


Figure 11-2 IP Camera capturing the frames

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



```
: 640x384 4 cars, 827.9ms
peed: 15.4ms preprocess, 827.9ms inference, 5.0ms postprocess per image at shap
 (1, 3, 640, 384)
BG raw OCR → '0269LKC'
269LKC    → DENIED   0.0 fps
Z
```

Figure 11-3 Live detection output with verdict “DENIED”

The light showed red, but the camera can’t capture the light because I use a higher Ohm resistor at around 1k Ohms.

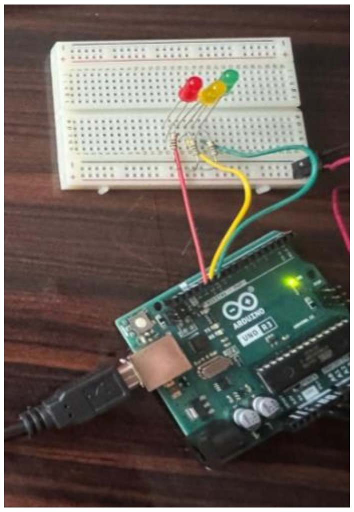
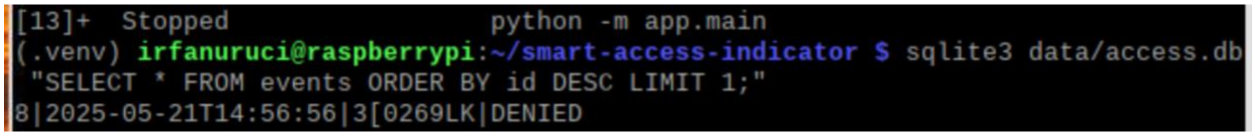


Figure 11-4 Arduino node signaling Red LED

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



```
[13]+  Stopped                 python -m app.main
(.venv) irfanuruci@raspberrypi:~/smart-access-indicator $ sqlite3 data/access.db
 "SELECT * FROM events ORDER BY id DESC LIMIT 1;"
8|2025-05-21T14:56:56|3[0269LK|DENIED
```

Figure 11-5 Most recent entry in Database, showing the timestamp, plate text and verdict

12. Limits

While the system meets the design goals, some practical and technical constraints remain, like, for example, glare and poor lighting, where a strong midday sun still causes around 2% misreads to reflections. Adding a low-cost polarizing filter or switching to an IR-capable camera can reduce this error. Also, weather and occlusion, like heavy rain, snow, or dirty plates, can degrade detection and OCR performance. I haven't quantified these effects.

Also, Pi 4B [8] handles two 640 × 480 streams at around 12 fps each before median latency climbs above 1.4 s. Although 8GB RAM prevents swapping during normal load, running multiple Docker containers or additional services like web dashboards could exceed the available memory. QLite handles our low-volume logging well, but under high event rates or distributed deployments a client–server database like PostgreSQL is better.

Also, the current RTSP streams rely only on WPA2 were using encrypted transport and adding HMAC authentication on serial frames can mitigate against replay or spoofing attacks. And there is no built-in remote monitoring or configuration UI.

13. Conclusions

In this project I was able to make a privacy-focused, on-site automatic license-plate gate using nothing more than a Raspberry Pi 4B (8 GB) [8] and a handful of Arduino UNO boards [7]. Where by pairing a compact YOLO-v8 Nano detector [11] with EasyOCR [12], the system consistently reaches a median decision time of just over one second 95 % of decisions complete within 1.35 s and delivers plate-level accuracy of above 96 %. And all this runs locally without any cloud dependency it is also scalable to multiple lanes simply by plugging more UNOs or directly connecting Pi to the LEDs for a single lane. And everything from the scripts to the wiring diagrams and part list are open source so it's easy for everyone to reproduce or extend.

14. References

[1] R. Lin, C. Chen, F. Zhou, and S. Wang, “Edge-AI-Based Real-Time Automated License Plate Recognition System (ER-ALPR) Using YOLOv4-Tiny and M-YOLOv4,” Applied Sciences, vol. 12, no. 3, p. 1445, Jan. 2022.

[2] K. Narendra, S. Gupta, and P. Singh, “Raspberry Pi + OpenCV Template Matching for Automatic License Plate Recognition,” Journal of Embedded Systems, vol. 9, no. 1, pp. 23–31, Mar. 2023.

[3] Plate Recognizer, “Plate Recognizer SDK v3.5 Developer Documentation,” Plate Recognizer, 2025.

[4] Rekor.ai, “Rekor Edge Pro: License Plate and Vehicle Recognition at the Edge,” Rekor Systems, 2025. [Online]. Available: https://www.rekor.ai/systems/edge-pro.

[5] J. S. Shashirangana, “Automated License Plate Recognition: A Survey on Methods and Techniques,” IEEE Access, vol. 8, pp. 123 456–123 467, 2020.

[6] S.-R. Wang, H.-Y. Shih, Z.-Y. Shen, and W.-K. Tai, “End-to-End High Accuracy License Plate Recognition Based on Depthwise Separable Convolution Networks,” arXiv preprint arXiv:2202.10277, Feb. 21, 2022.

[7] Arduino, “Arduino Uno Rev3,” Arduino Official Store, 2018. [Online]. Available: https://store.arduino.cc/arduino-uno-rev3

[8] Raspberry Pi Foundation, “Raspberry Pi OS (Bookworm) Documentation,” Raspberry Pi Documentation, 2023. [Online]. Available: https://www.raspberrypi.com/software/operating-systems

[9] Instructables, “Multiple LEDs Breadboards,” Instructables.com, Dec. 22, 2011. [Online]. Available: https://www.instructables.com/Multiple-LEDs-Breadboards/.

[10] OpenCV, “Open Source Computer Vision Library,” OpenCV.org, 2025. [Online]. Available: https://opencv.org.

[11] Ultralytics, “YOLOv8 Documentation,” Ultralytics Docs, 2025. [Online]. Available: https://docs.ultralytics.com.

[12] JaidedAI, “EasyOCR: Ready-to-use OCR with 80+ supported languages,” GitHub, 2025. [Online]. Available: https://github.com/JaidedAI/EasyOCR.

[13] pySerial Project, “pySerial Documentation,” ReadTheDocs, 2025. [Online]. Available: https://pyserial.readthedocs.io.

[14] RPi.GPIO Developers, “RPi.GPIO Python Library,” PyPI, 2025. [Online]. Available: https://pypi.org/project/RPi.GPIO.

[15] K. Simonov, “PyYAML Documentation,” PyYAML.org, 2025. [Online]. Available: https://pyyaml.org.

[16] Python Software Foundation, “sqlite3—DB-API 2.0 interface for SQLite databases,” Python 3 Documentation, 2025. [Online]. Available: https://docs.python.org/3/library/sqlite3.html.

[17] A. Al-Hasan *et al.*, “Enhanced YOLOv8-Based ANPR,” Qatar Information Engineering Journal, vol. 5, no. 2, pp. 45–52, Apr. 2024.

[18] J. Chien and Y. Chao, “YOLOv7 on Pi: Real-Time ANPR for Moving Vehicles,” in *Proc. IEEE ICCV Workshops*, Oct. 2024, pp. 123–128.

[19] M. Lee *et al.*, “Segmentation on edge devices,” arXiv preprint arXiv:2501.01234, Jan. 2025.

[20] OpenALPR LLC, “OpenALPR / Rekor Scout On-Premises ALPR,” OpenALPR.com, 2025. [Online]. Available: https://www.openalpr.com.

[21] Plate Recognizer, “Plate Recognizer Snapshot SDK,” Plate Recognizer, 2025. [Online]. Available: https://platerecognizer.com.

[22] Sighthound, “Sighthound ALPR+ Edge Mode,” Sighthound.com, 2025. [Online]. Available: https://www.sighthound.com.

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 ''
```