TU WIEN Informatics

Design and Implementation of an AI-Based Edge Device for Automated Traffic Counting

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Medical Informatics

by

Armin Kazda
Registration Number 11909468

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass. Mag.rer.soc.oec. Stefan Ohrhallinger, PhD

Vienna, October 27, 2025

Armin Kazda	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Armin Kazda

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 27. Oktober 2025

Armin Kazda

Danksagung

Ich möchte mich im Speziellen bei meinem Vater bedanken, der die Konstruktion des Prototyps durch Lötarbeiten zwecks Kompatibilität der Komponenten unterstützt hat.

Acknowledgements

I want to especially thank my father, who supported the construction of the prototype with soldering work to enable compatibility between the used components.

Kurzfassung

Diese Bachelorarbeit präsentiert den Entwurf, die Implementierung und die Bewertung eines kostengünstigen, autonomen Prototyps für die automatisierte Verkehrszählung und -analyse. Das System nutzt einen Raspberry Pi 5 mit einem HAILO-8-KI-Beschleuniger und einem Kameramodul, um Fahrzeugerkennung, Richtungserfassung und Datenprotokollierung in Echtzeit direkt auf dem Gerät durchzuführen. Der Prototyp wird über ein Solarpanel und eine Batterie mit Strom versorgt, arbeitet damit unabhängig von externen Stromquellen und ist für den flexiblen Einsatz im Freien konzipiert. Die Erkennungspipeline integriert ein für den HAILO-Chip optimiertes YOLOv8n-Modell und den ByteTrack-Algorithmus für die Verfolgung mehrerer Objekte, wodurch eine effiziente Fahrzeugerkennung und Bewegungsanalyse ermöglicht wird.

Zwei Prototypen wurden in Wien installiert und unter realen Bedingungen getestet. Die Auswertungsergebnisse zeigen eine hohe Erkennungsgenauigkeit mit einem Gesamt-F1-Score von 0,95 und einer Richtungserkennungsgenauigkeit von 88,8%. Die Temperaturaufzeichnung zeigt einen stabilen Betrieb mit Temperaturen unter 65 °C, selbst bei direkter Sonneneinstrahlung. Die Zuverlässigkeit litt unter der begrenzten Batteriekapazität und der Effizienz der Solarzelle, was die Laufzeit einschränkte. Eine spätere Hardware-Überarbeitung mit LTE-Konnektivität verbesserte die Zugänglichkeit.

Die Ergebnisse zeigen, dass die Kombination von Deep Learning und Edge Computing auf kompakter, energieautarker Hardware eine effektive, skalierbare Lösung für die Überwachung des städtischen Verkehrsflusses bieten kann. Zukünftige Arbeiten sollten sich auf die Optimierung des Energiemanagements, die Steigerung der Effizienz der Datenverarbeitung und die Verbesserung der Konfigurierbarkeit von Erkennungslinien und Ausgabefiltern konzentrieren.

Abstract

This bachelor's thesis presents the design, implementation, and evaluation of a low-cost, autonomous prototype for automated traffic counting and analysis. The system utilises a Raspberry Pi 5 with a HAILO-8 AI accelerator and camera module to perform real-time vehicle detection, direction tracking, and data logging directly on the device. Powered by a solar panel and battery, the prototype operates independently of external power sources and is designed for flexible outdoor deployment. The detection pipeline integrates a YOLOv8n model optimised for the HAILO chip and the ByteTrack algorithm for multi-object tracking, enabling efficient vehicle recognition and movement analysis.
Two prototypes were installed in Vienna and tested under real-world conditions. Evaluation results show a high detection accuracy, achieving an overall F1-score of 0.95 and direction-detection accuracy of 88.8%. Thermal monitoring confirmed stable operation below 65 °C even under direct sunlight. Reliability suffered of limited battery capacity and solar charging efficiency, constraining runtime. A later hardware revision adding LTE connectivity improved accessibility.
The results demonstrate that combining deep learning with edge computing on compact, energy-autonomous hardware can provide an effective, scalable solution for urban traffic flow monitoring. Future work should focus on optimizing power management, enhancing data processing efficiency, and improving configurability of detection lines and output filtering.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**

2 Related Work **5**
2.1 Miovision 5
2.2 Telraam 5
2.3 AI-based traffic counting systems 6
2.4 Other Contributors to this Project 6

3 The Prototype **7**
3.1 Hardware 7
3.2 Software 10
3.3 Mounting Locations 17

4 Evaluation **19**
4.1 Model Performance 19
4.2 Temperature 22
4.3 Reliability 24
4.4 Discussion 25

5 Conclusion and Future Work **31**

Overview of Generative AI Tools Used **33**

Übersicht verwendeter Hilfsmittel **35**

List of Figures **37**

List of Tables **39**

CHAPTER 1

Introduction

For urban planning and statistical analysis, counting and measuring traffic play a crucial role in understanding traffic flow. For example, this data is collected when redesigning a street to adapt it to the capacity necessary for the number of vehicles passing through. Despite the importance, counting the number of vehicles and tracking the direction is often still done with manual labour by hand-counting passing cars. This method is both time-consuming and expensive, as well as prone to errors, especially on busy roads.
The solution proposed in this bachelor's thesis aims to provide approaches for an automated, inexpensive design, capable of reliably analysing traffic flow. These devices should be easily deployable and as independent in operation as possible.
The prototype developed utilises a Raspberry Pi 5 with a camera module that, together with an AI accelerator module, conducts image analysis and statistical evaluation of traffic flow. Vehicles are counted, and their direction of travel is recorded. Together with analysis of the license plate (which is stored hashed for privacy reasons), multiple devices could make traffic flow analysis possible along streets or for measuring through-traffic in a district.

Automation is achieved by automatically launching the script and controlling the shutdown and boot times, minimising manual external input. Compared to other existing approaches to such automatic traffic-flow analysis devices, like ones of the company Miovision [Inc], the prototype constructed as part of this thesis aims to be as independent and easy to deploy as possible, with a solar panel and a battery as power source, eliminating the need of switching and recharging batteries, or having an external power source available at the mounting location. This makes the devices more flexible when it comes to choosing mounting locations. While other devices need to be easily accessible, this approach can be mounted higher up, reducing the possibility of theft and vandalism. This project was done in collaboration and is the basis for two other bachelor's theses [Har25][Tre25]. The construction and design of the prototype were done together, while the other thesis work puts the focus on specific image analysis operations inside the

detection pipeline. The focus of this thesis is the general construction of the pipeline, the design of the scripts, and the pipeline running on the Raspberry Pi, as well as the performance of the prototype and the general vehicle detection model. First, related work is presented. Then, the hardware and software of the prototype are presented, as well as the mounting locations of the two devices. Gathered data, like temperature curves and performance of the model, are analysed and discussed. The last section points out possibilities for further work on improving the design and construction of the prototype.

Figure 1.1: One of the prototype devices mounted on a light pole

CHAPTER 2

Related Work

2.1 Miovision

The company Miovision already provides similar types of devices, which can be rented to analyse various traffic situations. One such device offered is the battery-powered Scout Plus, which can be used to track and count traffic at intersections, streets, and roundabouts, and can also be used to measure pedestrian and bike traffic. Further use cases are speed data analysis and so-called "safety studies" [Inc].
These devices and the company were discovered later during the development process of the prototype and were unknown before; therefore, any similarities in design and use cases are a coincidence and not intended.

2.2 Telraam

On the contrary, the Belgian company Telraam developed small devices that are mountable on windows and then monitor the traffic outside [Telc]. These are marketed mainly towards private consumers who want to get insight into the traffic situation outside their home. The device can detect cars, buses, trucks, bikes, and pedestrians. Their approach was to create a small device that performs image analysis on the device and displays it on the built-in screen, as well as sending the count data to a server. The data is then not only available to the owners of the devices, but also published on a map [Telb]. However, Telraam devices don't have a built-in battery, but have to be connected to a power source via USB-C. The exact software and detection models used are unknown. In 2023, Telraam claimed to achieve over 90% weighted average for cars and bikes, 74% for trucks and larger vehicles, and 58% for pedestrians using the Telraam S2 device [Tela].

2.3 AI-based traffic counting systems

The paper "A YOLO-Based Traffic Counting System" by Lin et al (2018), published at the International Conference on Technologies and Applications of Artificial Intelligence (TAAI), proposed a similar approach to this prototype. The authors constructed an image analysis pipeline utilizing the YOLO framework as a base for traffic counting. However, the paper puts the focus on the software pipeline and achieving continuous tracking in a video stream by storing tracking data in a buffer. (Note: The software pipeline of our prototype utilizes an open-source pipeline published in 2021, see Chapter 3.2.3). The paper did not include the design and construction of a prototype, and does not mention any performance data of the hardware used, which could limit the overall performance of the model[LS18].

Another paper from 2021 proposed an AI-based traffic counting framework designed to run directly on small computers, addressing the latency and privacy challenges of cloud-based Intelligent Transportation Systems (ITS) [DNTL21]. The system by Dinh et al. integrates deep learning-based vehicle detection with tracking and a lightweight counting method, optimized for constrained hardware such as Nvidia Jetson Nano and Google Coral Dev Board. To support this, they created a Vietnamese Vehicle Detection Dataset (VDD), which reflects local traffic conditions dominated by motorcycles. Experimental results demonstrated real-time performance (26.8 FPS) with high accuracy (92.1%), showing the feasibility for deployment in smart city traffic management. Beyond traffic monitoring, the authors noted its applicability to other surveillance contexts, such as crowd monitoring and visitor counting in public spaces. This work highlights the effectiveness of combining deep learning with edge computing for scalable, low-cost, and accurate traffic flow analysis.

2.4 Other Contributors to this Project

Due to the size of this project, two other students have contributed to the construction of the prototype so far, putting the focus on different parts of the software and hardware. The bachelor's thesis of Nicholas Harisch puts the focus on speed calculation based on image analysis. They designed a speed calculation pipeline that uses different approaches to estimate the speed of a detected vehicle without the use of radar sensors, using the bounding boxes of the vehicles detected by the YOLO model[Har25].
Another contributor to the project is Stefan Trenovatz. Their work on the project consists of designing an extra software pipeline for license plate recognition. In their bachelor's thesis, they trained a YOLO model for detecting license plates in a given image, and then used Optical Character Recognition (OCR) algorithms to get the data of the license plate, and record it as a hashed value[Tre25].

CHAPTER 3

The Prototype

3.1 Hardware

The hardware of the prototypes consists of a Raspberry Pi with an AI accelerator module, a solar module, and a weatherproof housing for outdoor use. The components and the estimated price of one prototype are described in detail in this section.

3.1.1 Raspberry Pi

Raspberry Pi computers are a series of small single-board computers distributed by Raspberry Pi Ltd. The first Pi was released in 2012, with the model used in the prototype being the 5th version of the Raspberry Pi [Ltda].
The Raspberry Pi 5 is a single-board computer produced by Raspberry Pi Ltd. It is equipped with a Broadcom 2.4GHz quad-core processor with ARM architecture and an additional GPU. Further features are dual-band 802.11ac WiFi, Bluetooth 5.0, four USB ports, two MIPI camera transceivers, as well as a PCIe 2.0 x1 interface to connect external hardware to [Ltd25].
The version used in the prototype is equipped with 8GB of RAM. Additionally, the Raspberry Pi was equipped with the Active Cooler, the Raspberry Pi Camera Module 3 (12MP), and the obligatory microSD card (128GB), as well as the AI HAT+ accelerator module described below.

3.1.2 HAILO AI accelerator module

The HAILO-8 module is an AI accelerator module by Hailo Technologies Ltd. For this project, the Raspberry Pi AI HAT+ is used, which is equipped with the Hailo-8 accelerator module. It is capable of achieving 26 Tera-operations per second (TOPS). It is connected to the Raspberry Pi via the PCIe connector, and is used to process the image analysis operations in the software pipeline [Ltd24].

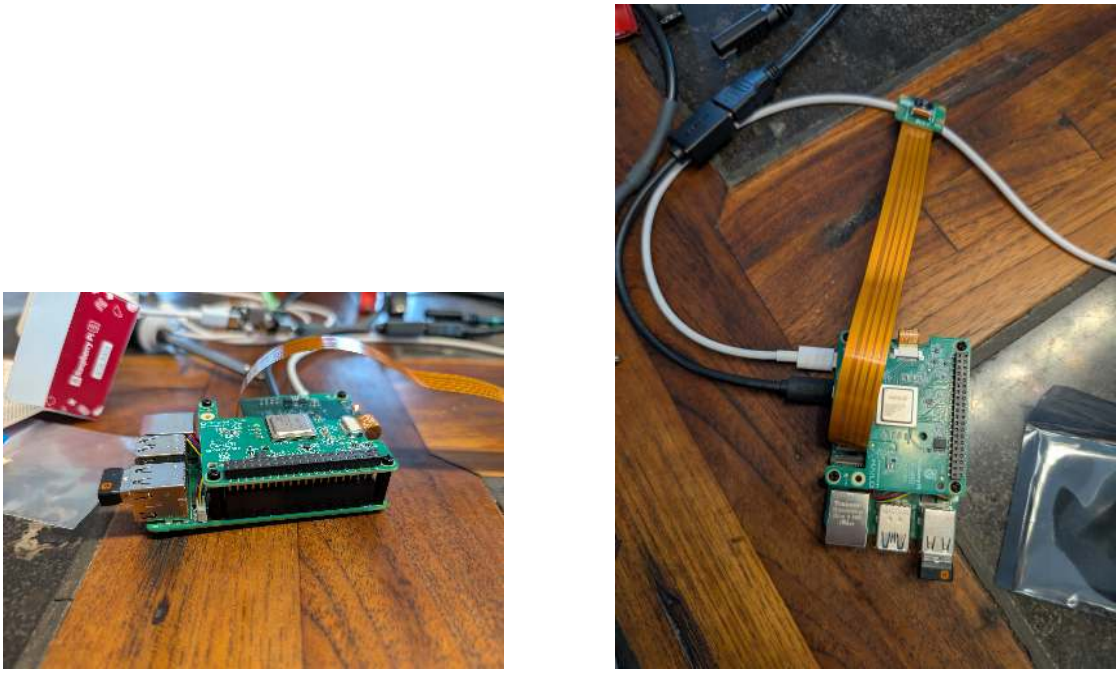


Figure 3.1: Raspberry Pi 5 with mounted AI HAT+ and Raspberry Pi Camera Module 3

3.1.3 Other components

To power the prototype independently from the power grid and make the prototype easier to mount on poles, a solar panel with a battery was used. The solar panel provides a maximum power of 25W, and, together with a dedicated controller, charges an 8Ah battery with 12V current. To power the Raspberry Pi, a converter from 12V to 5V USB and a USB-C cable were used. The converter is equipped with a coaxial power connector plug. To make the cable of the solar panel controller compatible, it was necessary to modify the cable with a compatible connector. Furthermore, the cable connecting the battery and the solar panel controller needed to be modified with different connectors due to space reasons. Figure 3.2 shows how the components are connected in a schematic drawing of the wiring of the prototype.

Additionally, waterproof housing for all components was needed. The Raspberry Pi with the camera and the HAILO module was fitted inside a surveillance camera-style housing together with the battery and the 12V to 5V converter. Due to space reasons, the solar panel controller needed to be fitted into a separate housing and was mounted onto the mount for the solar panel. For mounting, mountings to mount the components onto traffic/light poles were chosen individually to fit the pole.
In a first revision, an LTE-USB modem was added, connected to the Raspberry Pi with a short cable and, due to space reasons, a USB angle connector.

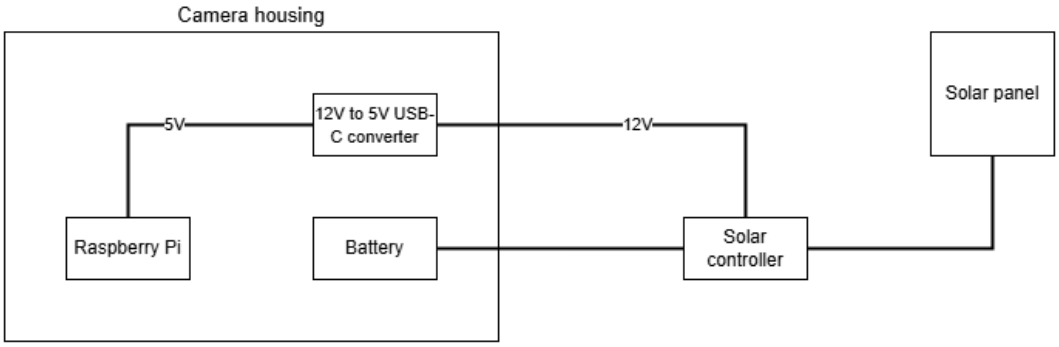


Figure 3.2: Diagram of the wiring setup connecting the solar panel, battery and the Raspberry Pi

Figure 3.3: One of the two prototypes mounted on a street light pole

3.1.4 Costs

The total cost of one prototype as of the end of August 2025 is around 533 €, without the mountings, which had to be chosen individually to fit the light/traffic pole they were mounted onto. These would add around 30€ more to the total cost.
Depicted below in Table 3.1 is the overall cost of one prototype as of the end of August 2025, including the addition of the USB-LTE module, added in the first revision. The price of the actual prototypes can be different, due to the parts being ordered between April and June 2025, and the parts of the first revision in August 2025.

Component	Notes	Price
Raspberry Pi 5	8GB RAM version	84.90 €
Raspberry Pi Camera Module 3	12 MP	28.90 €
Raspberry Pi AI HAT+	Hailo-8 module, 26 TOPS	122.90 €
Raspberry Pi 5 Active Cooler		5.80 €
SanDisk Ultra microSD Card	128GB, 140MB/s	11.50 €
Solar module + Battery	12V, 25W, 8Ah battery, including controller to regulate charging	100.83 €
Power Converter	12V to USB-C (with USB-PD standard)	3.00 €
Housing for camera, battery, etc.		27.99 €
Housing for Solar Controller	Waterproof housing for electric wiring	10.99 €
USB-C cable	With angled connector due to space reasons	46.96 €
DeLink USB-LTE module	DeLink DWM-222	63.90 €
USB extension with angle connector	Angle connector due to space reasons	23.48 €
SIM-card	from HoT (Hofer Telefon), data plan not included	1.99 €
Total		**533.14 €**

Table 3.1: Summary of the components used and their price

3.2 Software

This section explains the software used for the prototype, covering the operating system used and the technology behind the detection pipeline in detail. The software pipeline and the scripts are contained in a GitLab project [NH].

3.2.1 Raspberry Pi OS

To save processing power, it was necessary to choose an operating system that works best with the Raspberry Pi. Therefore, Raspberry Pi OS for the Raspberry Pi 5 was selected. It is a free OS, published by the Raspberry Foundation, and is optimised for operation on the Raspberry Pi computers [Ltdb]. Being a desktop operating system, it also has a full Graphical User Interface (GUI), providing easy usability with the option to have visual debugging, compared to terminal-based operating systems without any GUI. The most recent version of Raspberry Pi OS is based on Linux Debian Bookworm, which was released in June 2023 [SitPI23].

3.2.2 Scripts

To automate the prototype and start the tracking pipeline automatically, the use of shell scripts was necessary. In total, two Bash scripts ensure autonomous operation of the prototype. These are launched on startup by daemon services. These services are natively integrated into the operating system.
One script is responsible for automatic shutdown and boot procedures, and additionally launches a Wi-Fi hotspot to provide local remote access. Initially, the script was configured to boot the Raspberry Pi automatically at 6:00 in the morning and shut it down at 21:00 in the evening, resulting in a runtime of 15 hours per day. This timespan was selected to restrict system operation to daytime, as insufficient nighttime illumination could likely degrade the model's performance. Furthermore, limiting runtime to daylight hours

helped reduce energy consumption, since the system's battery could only be recharged during this period. However, boot and shutdown times were disabled after a week of operation, due to the clock running on the wrong date and time after a power loss due to a lack of battery (see Chapter 4.4). With the addition of the LTE-USB stick, permanent internet access was available, eliminating the problem of incorrect clock times. Therefore, automatic boot and shutdown times were re-activated in the script, setting the boot time of the prototype at 8:00 in the morning, and shutdown at 18:00, for a total runtime of 10 hours per day.
The second function of the script is to launch a local Wi-Fi hotspot, to enable local access to the prototype. When connected to the hotspot, the Raspberry Pi can be accessed over Virtual Network Computing (VNC) clients. VNC is an open-source protocol, enabling remote screen access to devices with an IP address. VNC is natively available in Raspberry Pi OS; connecting from a remote device requires the use of a VNC client.
Figure 3.4 shows the pipeline for how the scripts are launched after boot in their correct order, as well as the order of operations inside the script itself.

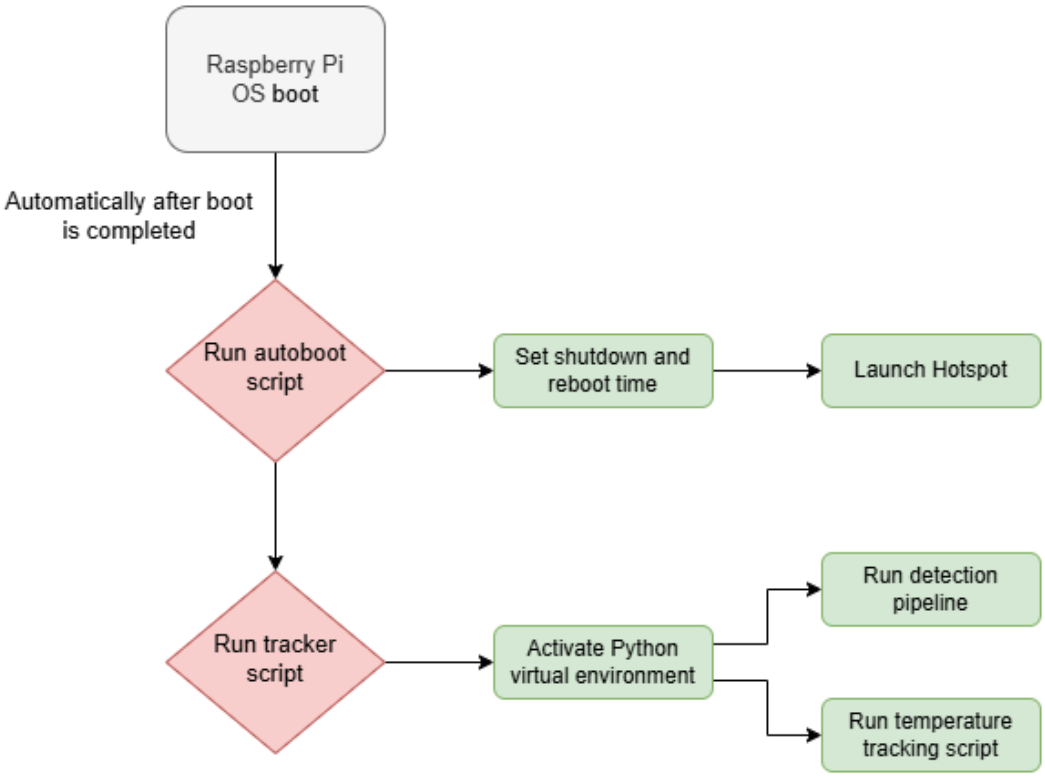


Figure 3.4: Order of launch of the scripts after the boot of the Raspberry Pi

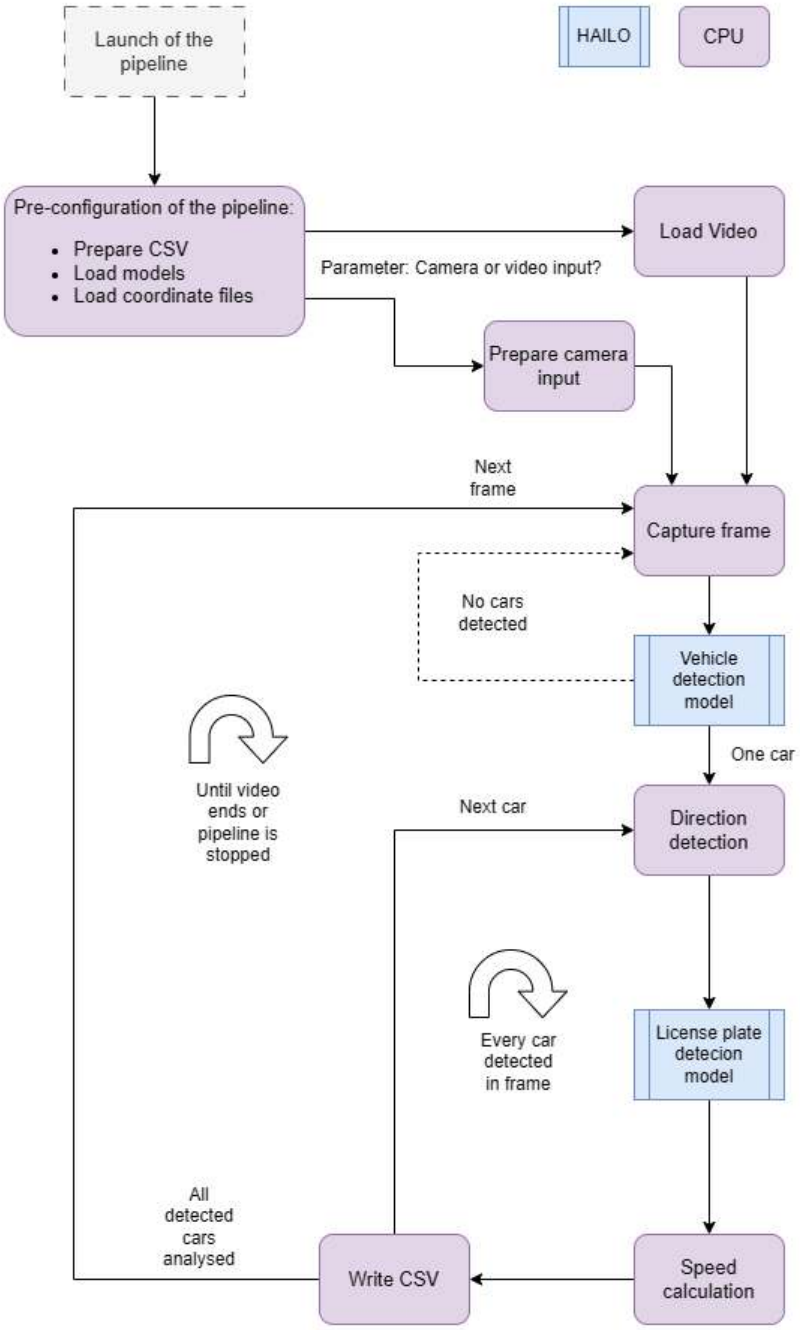


Figure 3.5: Tracking pipeline, showing how the input is handled by the Python script

3.2.3 Tracking Pipeline

Figure 3.5 shows a graphic representation of the tracking pipeline running on the Raspberry Pi. The tracking pipeline is implemented as a Python-based script that interacts with the HAILO module and manages all image processing operations. These operations include the vehicle speed calculation, license plate recognition, and direction detection. After the launch of the pipeline, which is usually automatic after boot and controlled by another script, the pipeline is initialised and pre-configured: The CSV file is prepared, the models are loaded, and the input source is configured. The pipeline allows either a live camera stream or a video file as input, providing ways to test the pipeline when the prototype is not mounted in place. Furthermore, an optional live visualisation mode can be activated for debugging purposes. However, this feature significantly impacts performance and is therefore unsuitable for regular operation.
Vehicle detection is performed continuously within a real-time processing loop. Each captured frame is transmitted to the HAILO module, where image analysis is done using a YOLOv8n model integrated through the DeGirum library (see Subsection YOLO for details). The detection model is optimized for compatibility with the HAILO hardware, and is limited to only detecting cars, excluding other types of vehicles. Detected vehicles are temporarily stored in a buffer and subsequently tracked across frames using the ByteTrack algorithm (see Subsection ByteTrack for details) to maintain object consistency. Direction detection is achieved by comparing the position of each tracked vehicle against predefined reference lines within the image frame. When a vehicle approaches a reference line, its entry and exit directions are determined and stored (for details, see section 'Direction detection' below). Subsequent processing stages include license plate recognition (as described in Trenovatz's thesis [Tre25]) and speed calculations (see Harisch's thesis for this [Har25]). Finally, all processed and measured data are recorded in a structured .csv file in the format below. For every frame, one line is written for each detected car, containing the timestamp, the internal car ID, and optional recorded data of license plate, direction, and speed detection. The .csv file is named with the timestamp of the pipeline's launch, placing each run in its own file to facilitate better comparison and statistical analysis later.

```
['timestamp', 'car_id', 'lp_hash', 'entered', 'exited',
'position_bottom_center', 'position_tp', 'speed']
```

YOLO

YOLO (You Only Look Once) is a detection architecture developed by Redmon et al. in 2016. Unlike earlier approaches, YOLO treats detection as a single regression problem, directly predicting bounding box coordinates and class probabilities from the input image in a single forward pass. This allowed the detection model to be fast and reliable, outperforming predecessors and competitors [RDGF15].
Since the original YOLO paper, multiple versions have been developed, improving accuracy and speed. Ultralytics, an independent research and development company, has been

a key contributor to YOLO's modern evolution—particularly with YOLOv5 (released mid-2020) and YOLOv8 (released in 2023).
Today, the most widespread models used are YOLOv5, YOLOv8, and YOLO11. Each of the models differs in features, image processing time, and accuracy. YOLOv5 only supports object detection, while newer models introduced new features like image classification and object tracking. Of the three models mentioned above, YOLOv8 achieves the lowest image processing time on some devices of 0.99ms when using the most lightweight model, while retaining a high mean average precision [Ltdc]. Default ultralytics YOLO models are pretrained with the COCO dataset, a dataset for object detection, containing information for 91 different object types, including cars, planes, trees, etc. [LMB+14]. However, YOLO models can also be custom-trained for different use cases. Due to the efficiency and the widespread availability of YOLOv8, with optimisations for multiple platforms and processors, it was chosen as a base model for the detection pipeline on the Raspberry Pi. Each of the YOLO models comes in different versions, scaling in size and accuracy at the cost of processing time. For models since YOLOv5, the versions nano (n), small (s), medium (m), large (l), and xlarge (x) are available. Since the power of the Raspberry Pi and the HAILO module is limited, we aimed for one of the lightweight models like YOLOv8n or YOLOv8s.

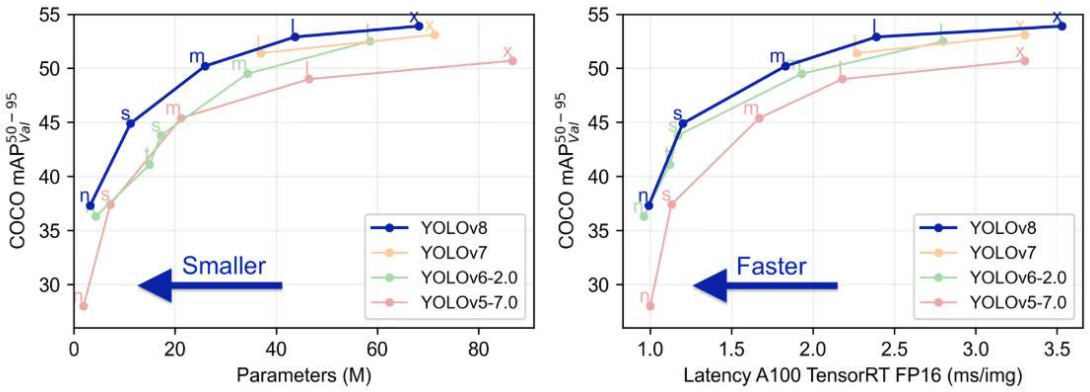


Figure 3.6: Diagrams comparing YOLO models to YOLOv8 regarding size/precision and latency/precision. Source: [Ltdc]

Several approaches are possible to access the HAILO module and provide it with the YOLO model to do the image processing for the detection. The challenge was the need to use two different models for one frame: the normal image detection model to detect the cars, and the license plate detection model trained by Trenovatz. Natively, the HAILO module only supports the use of one model for one processing task, so a third-party Python package was necessary to control the pipeline to the HAILO module and make the use of both models possible. Out of several approaches, the degirum library proved to be suitable and, while not being the most efficient one regarding performance, easy to use when using two models.

Degirum provides cloud-based image processing with models like YOLO through its "AI hub", but its Python package can also be used to access and run models on local hardware devices like the HAILO8 chip. Further, they provide pre-compiled models, like a YOLOv8n model optimised for car detection, in a format compatible with the HAILO8 chip. This model was used for the detection of cars in the detection pipeline of the prototype, and is limited to detecting cars only, excluding larger vehicles like trucks and buses, as well as bikes and other two-wheeled vehicles.

ByteTrack

ByteTrack, introduced in a paper by Zhang et al. in 2022, forms a novel approach to multi-object tracking (MOT) that focuses on maximizing the use of detection data rather than discarding low-confidence results. Compared to other MOT systems, which typically remove detections below a confidence threshold to avoid false positives, ByteTrack proposed a different tracking strategy that associates nearly all detection boxes, thereby improving trajectory continuity and reducing identity fragmentation [ZSJ+21].
The proposed method, called BYTE, employs a two-stage association process. In the first stage, high-confidence detections are matched with existing tracklets using motion or appearance similarity based on Intersection-over-Union (IoU) or Re-Identification (Re-ID) features. The second stage re-associates unmatched tracklets with low-confidence detections, using motion similarity alone to recover missing objects while filtering out background noise. This dual-stage design enables ByteTrack to recover occluded targets without significantly increasing false positives. A Kalman filter predicts object motion, and the Hungarian algorithm performs optimal matching, ensuring both computational efficiency and accuracy [ZSJ+21].
For implementation, ByteTrack uses the YOLOX detector as its backbone, achieving real-time tracking performance at approximately 30 frames per second on an NVIDIA V100 GPU. Extensive experiments on major MOT benchmarks, including MOT17, MOT20, HiEve, and BDD100K, demonstrate its effectiveness. ByteTrack achieved suitable results, with 80.3 MOTA and 77.3 IDF1 on MOT17 and 77.8 MOTA and 75.2 IDF1 on MOT20, significantly outperforming previous methods. The results indicate that considering low-confidence detections improves both identity preservation and tracking robustness, particularly in crowded or occluded environments. Figure 3.7 shows the performance of ByteTrack compared to other widespread trackers for multiple-object tracking [ZSJ+21]. The trackers are compared both in achieved frames per second as well as MOTA. MOTA is short for Multiple-object tracking accuracy, a metric used to evaluate the errors of object trackers [OEC].
Due to the low performance requirements, compatibility with the YOLO model pipelines, and open-source availability, ByteTrack proved to be suitable for use on a device with limited power like the Raspberry Pi. Furthermore, street environments tend to be more crowded with occluded objects, e.g., cars driving next to each other on two lanes, resulting in lower confidence scores at detection. Additionally, ByteTrack is natively implemented into the Ultralytics YOLO Python package for object tracking, therefore suggesting its

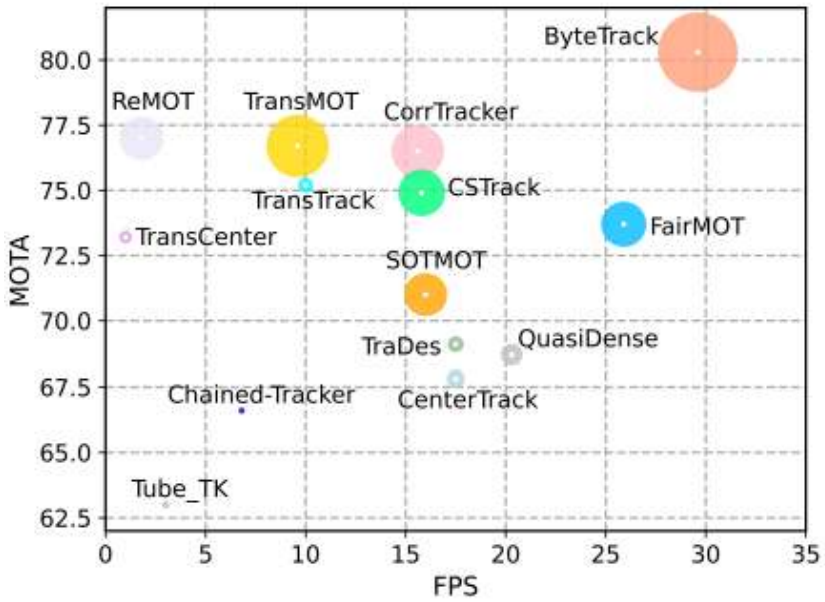


Figure 3.7: Performance of ByteTrack compared to other trackers [ZSJ+21].

use on the Raspberry Pi´s pipeline too.

Direction detection

For detecting the direction the cars are travelling through the frame, a lightweight algorithm was developed to save processing power for more resource-intensive tasks. The algorithm receives the coordinates of a point and compares its position to the position of a given line. The distance between the point and the line is calculated by interpolation. If the point is within the range of the line, defined by a threshold, the script returns a string that is given to the algorithm on call. If the point is outside the range, an empty string is returned. In the main pipeline, each line is one invocation of the algorithm. At invocation, the algorithm is invoked with the center of the bounding box of the detected car, the line, and the exit direction. If a non-empty string is returned, the direction of entering and exiting is put into the variables that are then written into the .csv file. For each pre-defined line, the algorithm has to be invoked separately, requiring manual re-configuration of the pipeline for each location the prototype is mounted at.
To define the lines, a separate Python tool can be used. This uses a given image and allows the user to define points on the image. These points are then saved into a .cfg text file, and can then be used in the main pipeline for the calls of the algorithm. One line is made up of one start and one endpoint.

3.3 Mounting Locations

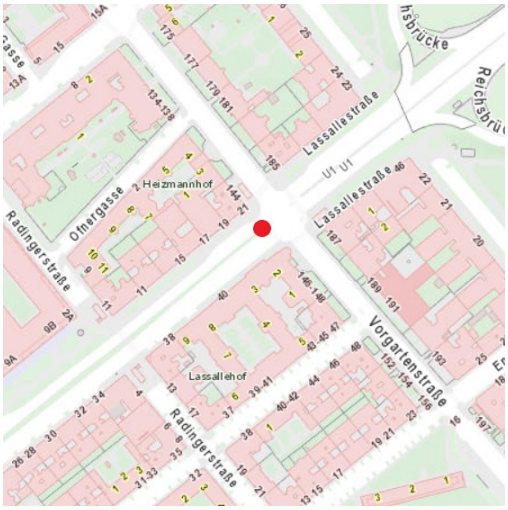
Two locations in the 2nd district in Vienna were chosen as suitable locations for mounting the prototypes. Both locations are on streets with an assumed high amount of through traffic. The prototypes were mounted in July 2025.
The first location is located at the intersection of Lassallestrasse and Vorgartenstrasse, which is equipped with a traffic light. The prototype is facing away from the crossing towards Praterstern in a southwestern direction. Tracking is possible in two directions:

1. Two lanes towards Praterstern in the southeastern direction
2. Four lanes towards the intersection, three for traffic continuing straight towards Reichsbrücke or Handelskai, one for turning right onto Vorgartenstrasse

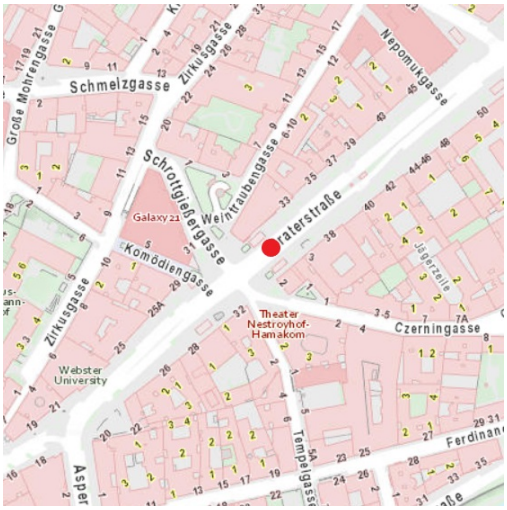
The second mounting location is located on Praterstrasse, next to Nestroyplatz. The camera is facing in a southwestern orientation towards a crossing with a traffic light. Tracking traffic is possible in three directions:

1. Towards Praterstern in the northeastern direction
2. Southeastern direction, turning left at Nestroyplatz
3. Turning right at Nestroyplatz, or staying on Praterstrasse, heading towards the inner city.

Figure 3.8 shows the mounting locations of both prototypes on a map. The locations are approximately 1.6km apart from each other, and connected by the Praterstrasse and the Lassallestrasse via the Praterstern, a major crossing in the 2nd district, formed by a large roundabout around a train and subway station, connecting several big streets in the 2nd district.
For location Lassallestrasse, a national traffic count from 2020 mentions an average of around 32,500 vehicles in a period of 24 hours, 31,600 of them being cars [BfK] (section 'Wien', count location 1075). Another source is counting data of the city of Vienna from 2015, which mentions ≈ 200 to ≈ 370 vehicles per 15 minutes towards Praterstern, and ≈ 200 to ≈ 440 vehicles per 15 minutes towards Reichsbrücke during the daytime [GMB].
For the Praterstrasse location, one article published in 2019 by Georg Scherer on wien-schauen.at could be found. The source claims an existing study of the Technical University of Vienna; however, the author does not provide a reference to the study, and the study could not be found anywhere. Therefore, it is unclear if the data is reliable. The article mentions that the study, conducted by Ulrich Leth and Harald Frey, speaks of 21,000 vehicles per day [Sch19].



(a) Location Lassallestrasse



(b) Location Praterstrasse

Figure 3.8: Locations where the prototypes are mounted. Source: https://www.wien.gv.at/stadtplan/

(a) Location Lassallestrasse

(b) Location Praterstrasse

Figure 3.9: View of the camera of both prototypes

CHAPTER 4

Evaluation

4.1 Model Performance

To evaluate the performance of the model at both mounting locations, two 15-minute-long videos were recorded on a Friday at around 16:00. These videos were then run through a similar software pipeline on a Windows computer running a YOLO model, which provides a video output as an end result. The same model running on the Raspberry Pi could not be used, since it is in a special format, compatible with and optimised to run on the HAILO module.
With this video, it was possible to analyse the accuracy of the model by hand-counting the cars and checking for false positives and negatives. Unfortunately, the video could not be run through the pipeline on the Raspberry Pi due to the pipeline only providing a live debug video view and no option to save a video after the pipeline finished analysing the video. Additionally, enabling the visual output of the pipeline has an impact on the overall performance, which could have an impact on the performance of the tracking model.
The videos were recorded with a resolution of 1280x720 pixels, at 30 frames per second, the same size and frame number that the pipeline receives in regular use. The model used to detect the cars in the video was the stock YOLOv8n model, configured with similar parameters to the software pipeline running on the Raspberry Pi. Therefore, only cars were detected, excluding larger vehicles and two-wheeled vehicles.

	True Positive (TP)	False Positive (FP)	False Negative (FN)
Left side	349	5	0
Right side	240	22	0
Total	589	27	0

Table 4.1: Results of the model at location Lassallestrasse

	True Positive (TP)	False Positive (FP)	False Negative (FN)
Left side	97	16	0
Right side	154	40	0
Total	251	56	0

Table 4.2: Results of the model at location Praterstrasse

The data in Table 4.1 and Table 4.2 show the results of the analysis of the YOLO model and the evaluation by hand counting the cars in the video. This data could be used to calculate precision and recall of the model, and therefore the F1-Score.
Counting the cars was done by individually counting the left and right sides/lanes for each location. This enabled the possibility to get a better view of where the False Positives are located, identify the weaknesses of the model, or possible improvements regarding camera angle and location. Furthermore, the F1-Score was calculated for both locations separately, as well as an overall F1-Score with the total numbers.

	Precision	Recall	F1-Score
Lassallestrasse	0,9562	1	0,9776
Praterstrasse	0,8176	1	0,8996
Total	0,9101	1	0,9529

Table 4.3: Precision, Recall and F1-Score of the detection results

The evaluation results of the detection method are shown in Table 4.3, which contains precision, recall, and F1-score for the two test locations as well as for the aggregated results.
For Lassallestrasse, the model achieved a precision of 0.9562, indicating that less than 5% of the detected instances were false positives. The recall value of 1 demonstrates that the system successfully identified all true instances, resulting in an F1-score of 0.9776. These results suggest that the method performed almost ideally at this location, with a negligible trade-off between precision and recall.
At Praterstrasse, the precision is lower at 0.8176, meaning that around 18% of the detections were false positives. However, the recall again reaches 1, showing that no relevant instances were missed. The corresponding F1-score of 0.8996 reflects this imbalance: While the method detects all relevant cases, it produced more erroneous detections at this location compared to the other one.
Combining the results, the system achieved an overall precision of 0.9101 and a recall of 1, leading to an F1-score of 0.9529. This indicates that the method is highly reliable in identifying all true cases at both locations, with only a small tendency towards false

positives.
The differences in precision are attributed to environmental factors, the higher presence of traffic light poles and street signs at Praterstrasse, leading to cars getting "split" and getting detected twice, resulting in false positives. Additionally, bigger vehicles like buses were sometimes identified as cars in a short timespan, with bigger bicycles and motorbikes sometimes also getting misidentified. This is likely a result of the low confidence value of 0.25 as a default threshold for detection; however, this represents the default confidence value.

Detection line performance

At the location Lassallestrasse, two detection lines were used, one for the lanes towards Reichsbrücke and the intersection on the left side of the frame, the other covering the lanes towards Praterstern on the right side. At Praterstrasse, three detection lines were created, one covering traffic towards Praterstern on the left side, one line for detecting traffic turning left at the intersection, and the last line for traffic going straight or turning right. It was not possible to create separate lines for traffic entering from the left or right side at the intersection, or coming from the inner City and not continuing on Praterstrasse towards Praterstern, due to the distance to the crossing (see Figure 3.9 for reference of the camera view at both locations).
The data from the detection lines can be compared to the data from the model performance above to obtain accuracy data of the detection line algorithm. The distance threshold for the detection lines was set to 10; therefore, a crossing of the line was recorded if the distance of a detected car to the line was less than 10 pixels.

	Detection line data	Model data (TP + FP)	Percentage
To Reichsbrücke	319	354	90,11%
To Praterstern	253	262	96.56%
Total	572	616	92.86%

Table 4.4: Accuracy of the detection lines at location Lassallestrasse

	Detection line data	Model data (TP + FP)	Percentage
To Praterstern	68	113	60.18%
To Intersection (left+straight/right)	180 (9+171)	194	92.78%
Total	248	307	80.78%

Table 4.5: Accuracy of the detection lines at location Praterstrasse

The detection line data presented in Tables 4.4 and 4.5 represent the number of unique tracking IDs intersecting the respective detection lines. When combining the datasets from both locations, an overall accuracy of 88.84% is achieved. False Positive detections were included in the analysis, as erroneously identified vehicles may intersect a detection line and thus be falsely counted. False Positives may also be detected by the model

without ever crossing a detection line, resulting in their exclusion from the line-based count. Additional sources of inaccuracy include correctly detected vehicles that do not enter the threshold of any detection line. This can occur either when vehicles pass through the frame without crossing a line or when they intersect with the line too rapidly for the center of the detection bounding box to fall within the predefined threshold.
The accuracy at Praterstrasse is notably lower than at other detection locations. This discrepancy is primarily attributed to vehicles approaching from the left side of the intersection but not turning toward Praterstern. These vehicles are detected within the frame but do not intersect any detection line, preventing the assignment of a travel direction. In contrast, the reduced accuracy observed at Lassallestrasse is likely due to vehicles moving too quickly for detection, even though all vehicles have to cross one of the lines. If only True Positive detections toward Praterstern were considered, the resulting accuracy would exceed 100%, suggesting that False Positive detections occurred in proximity to the detection line and were thus also included in the count.

4.2 Temperature

In a first test to estimate the CPU temperature of the Raspberry Pi during operation in the prototype, the solar panel setup and a stock Raspberry PI 5 with no additional devices and modifications was used, running a Python script that recorded the CPU Temperature every ten minutes. As seen in Figure 4.1 below, the components were put inside a dark grey plastic box and put in direct sunlight, to simulate the conditions the prototype would likely be exposed to.

The temperature monitoring started at 15:23, with an outside temperature of approximately 25 °C. The initial goal was to record the CPU temperature data over a period of 24 hours. However, the measurement ended at 01:43 due to the battery running out of power, which caused the Raspberry Pi to shut down. As Figure 4.2 shows, the CPU temperature was approximately 63 °C at the start of the recording, and remained around 65 °C during daylight, with a maximum recorded value of 68 °C. Following sunset, the temperature gradually declined, reaching a minimum of 50 °C. The final recorded value before shutdown was 52 °C.
To obtain continuous data of the temperature from the prototype, a monitoring script is running parallel to the tracking script, recording the temperatures of the CPU and the HAILO module at 30-second intervals. Compared to the data of the first test, the performance regarding temperature turned out better than expected. Despite operating under load and in a more enclosed space, the temperatures remained roughly similar to those recorded prior.

Figure 4.3 displays the temperature development of the Raspberry Pi CPU and the HAILO module at the location Praterstrasse under normal operations, running the tracking script. The measurements were recorded between 08:35 and 12:10. The y-axis

Figure 4.1: Test setup for temperature monitoring

represents the temperature in degrees Celsius (°C), while the x-axis represents the time of day. The red line is the temperature curve of the Raspberry Pi's CPU, the blue line shows the temperature of the HAILO module.
During the first 30 minutes of the measurement, both the CPU temperature and the temperature of the HAILO module increased from 26-28°C to around 50-55°C. After this initial rise, the temperature curves entered a slower growth phase, approaching a steady-state condition. The Raspberry Pi CPU temperature consistently remained higher than the HAILO temperature throughout the measurement, with an average offset of approximately 3-5°C.
Between 09:00 and 11:30, both temperatures stabilized in the range of 55-60°C, with the HAILO module temperature remaining slightly lower, fluctuating around 54-58°C. Both curves exhibit small fluctuations, likely due to variations in the overall workload. The small increase in the average temperature of both devices at around 10:40 could be explained by an increased exposure to sunlight. Toward the end of the observation period, the CPU temperature increased again, peaking at approximately 65°C, whereas the device temperature rose more moderately to around 59°C. This was likely caused by increased load due to remote access to the device using VNC, with the need to render the GUI and process input data.
Comparing the data of the prototype with the temperature curve of the first test, it shows that the mounted prototype turned out to be more temperature efficient than expected. The recorded peak temperature of the first test was higher than that of the

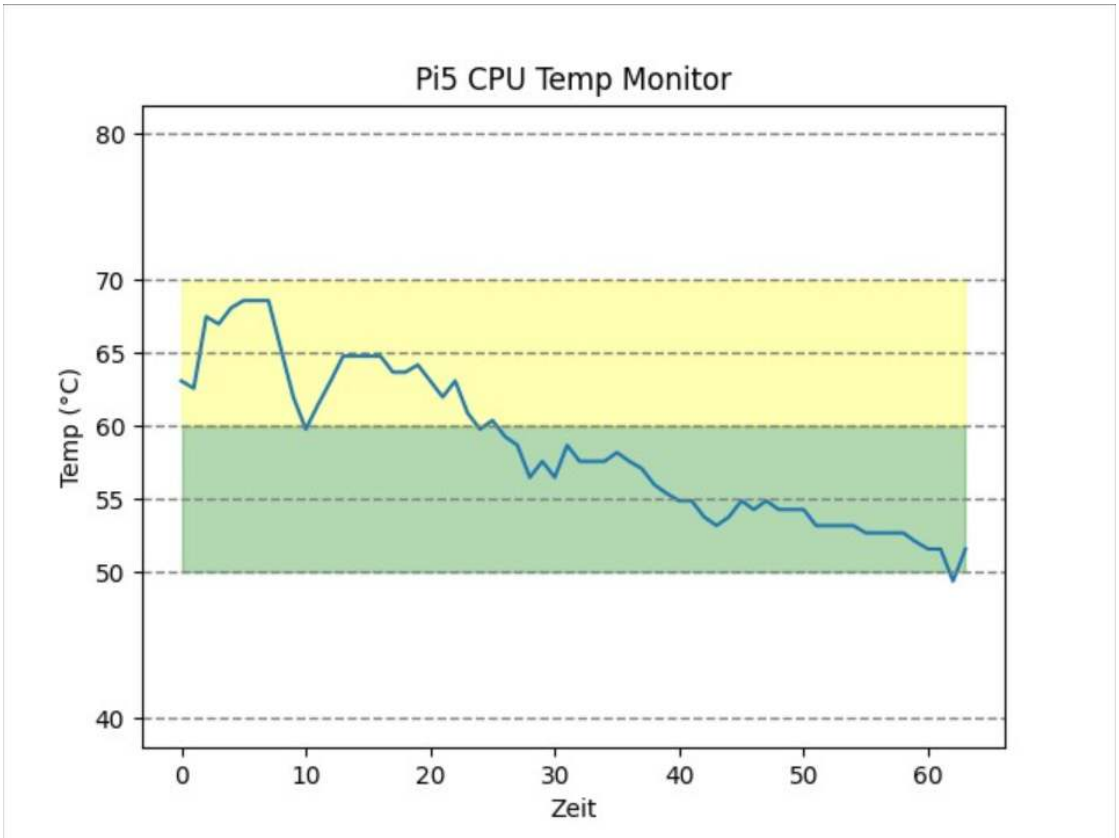


Figure 4.2: Plot of the temperature curve, 'Zeit' (Time) on the x-axis refers to the number of data points (one per ten minutes)

prototype, despite running in idle mode compared to running the tracking script. Of course, it is possible that the difference in daytime and outside temperature could make a difference. Both temperature curves were recorded on sunny days with temperatures ranging from 25 °C to 30 °C, with both devices being exposed to sunlight.

4.3 Reliability

Obtaining accurate data regarding runtime to measure the reliability was not implemented in the code. However, with the tracking pipeline and the temperature script both recording timestamps, an estimation regarding runtime can be made to get a view on the reliability of the prototypes.
Runtime proved to be very mixed throughout the first months of test operation. In the first week, with the prototypes configured to run from 6:00 to 21:00, power often cut out early, resulting in a shutdown due to the under-voltage protection of the Raspberry Pi or the battery voltage being too low to provide sufficient power to boot the system. With no power, the internal clock stopped, and due to a lack of internet, it could not be

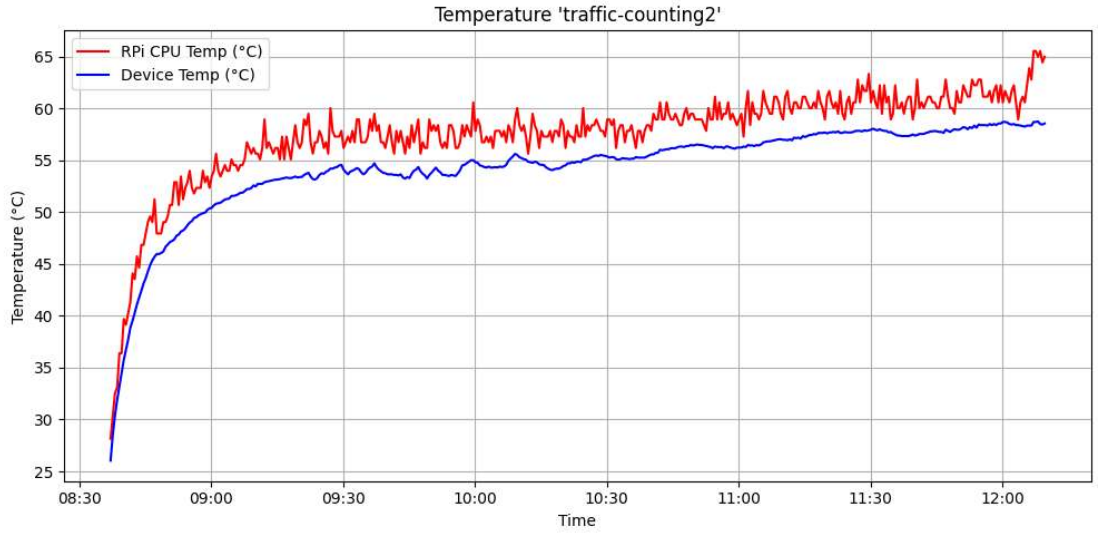


Figure 4.3: Plot of the temperature curve of both the CPU of the Raspberry Pi and the HAILO module while running the tracking script.

updated after booting, resulting in an offset of date and time, making accurate runtime timestamps impossible, and shifting the times of operation.
A first solution was to run the tracking script continuously until the prototype shut down due to lack of battery power. This resulted in data regarding total runtime; however, no information regarding the exact time of boot can be made, due to the date being shifted. Sometimes, the tracking script ran for several hours, in some instances even more than 12 hours, likely due to good weather charging the battery, making operation even at night possible. On other days, several short periods of the script running were recorded, with the script likely stopping due to lack of power, and the Raspberry Pi repeatedly trying to reboot and start tracking again.
Adding the USB-LTE-Sticks in the first revision eliminated the problem of inaccurate date and time on the prototypes. Establishing an internet connection on startup provided the OS with the possibility to synchronise the date and time, resulting in accurate timestamps when tracking. However, reliability data after the first revision is not in the scope of this thesis.

4.4 Discussion

The design and construction of the prototype provided valuable insights into the construction of such devices, and revealed flaws of the chosen components and further trade-offs. The development of the software pipeline showed the flexibility of the Raspberry Pi and its operating system, as well as the trade-offs that have to be made when conducting image analysis on lightweight devices with significantly lower performance compared to traditional computers.

The construction of the prototype faced several challenges, mainly related to making the chosen components properly compatible with each other. The converter converting the 12V provided by the battery and the solar controller to 5V uses a USB-C port on the 5V side and a standard coaxial power connector for the 12V input. The cable of the solar controller provided only small metal rings as connectors; therefore, the cable had to be modified with a connector to make it compatible with the converter, as shown in Figure 4.4.

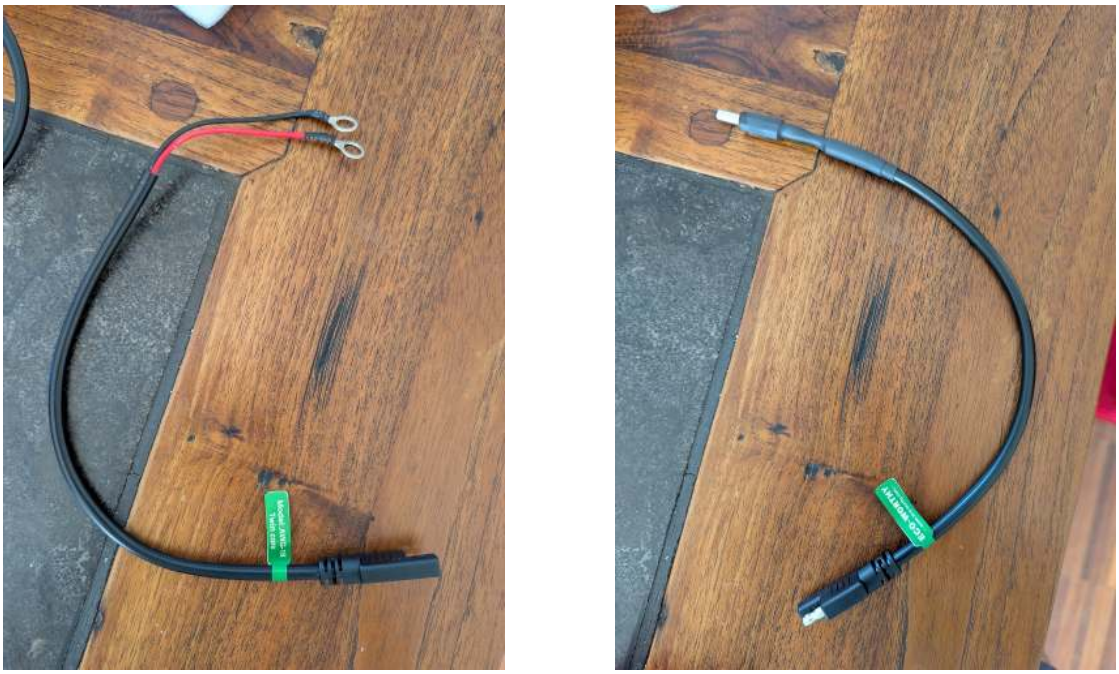


Figure 4.4: The cable connecting the solar controller with the 12V/5V converter in its unmodified and modified state.

Setting up the solar panel with the controller and the battery was easier than expected and close to self-explanatory. The biggest challenge with the construction of the prototype was the size of the hardware in relation to the size of the camera housing. Fitting the battery, the Raspberry Pi, the controller, and the converter into the housing was impossible due to the limited space; therefore, it was necessary to fit an additional housing for the solar controller onto the mount of the solar panel. Furthermore, the initial USB-C cables that were selected to power the Raspberry Pi did not fit the housing. When the cable was connected, it was not possible to close the lid of the housing, requiring different USB-C cables with an angled connector. Apart from the size, the housing proved to be more reliable than expected. Waterproofness was given as advertised, with the housing remaining dry on the inside, despite several days of rain during the testing period from July to September. The biggest concern of the prototype device was the temperature during operations, especially during summer in direct sunlight, and the fact that the

housing has a metal top. The first temperature test in a black plastic box raised this concern, with quite high temperatures even in idle, with no power-consuming pipeline running. However, during the test of the actual prototype when the detection pipeline was operating, the temperature was lower than expected, with a peak of around 60° C, even in direct sunlight, it remained lower than during the first test.
For the Raspberry Pi itself, valuable insights on how to automate the operation of the devices using scripts were achieved. Due to the Raspberry Pi 5 having a power button and the ability to maintain a shutdown mode as long as connected to a power source, it is the first Pi where it is possible to automatically shut down and reboot the system with just the unmodified Raspberry Pi and Raspberry Pi OS, without the need for a small buffer battery necessary on previous Pis. This was useful to save battery during nighttime when the Raspberry Pi was not running the detection pipeline, as the first temperature test also showed that running the device during nighttime would drain the battery so much that the system would run out of power during nighttime, and only power on again when the battery was charged enough. On the other hand, this power-saving state also caused some problems related to it, see section 'Problems' below.
The first approach to a software pipeline was done as a collaborative project on Gitlab [NH] before the prototype was built, therefore it contained a Python script based on Python libraries that address GPUs or regular x86 CPUs, like the Ultralytics Python package. Porting the code onto the Raspberry Pi showed that this package is incompatible with the Raspberry Pi and especially the HAILO module, requiring special libraries and model formats.
Installing the HAILO module and its required packages was very easy, and there is built-in support for running stock YOLO models on the HAILO in the basic packages. However, it was necessary to run two models for one software pipeline, due to the license plate recognition model being a specifically trained model for its use-case, and ByteTrack needed to be integrated too to buffer the detected vehicles between frames and track their path through the frames. The first two approaches, using a GStreamer pipeline and direct inference via the HAILO module's native tools, turned out to be particularly difficult to implement. Directly addressing the module would have been the most efficient approach, as there would be no library needed to convert the commands into the right format. But due to the poor documentation of the code, it was not possible to create a proper working pipeline in a reasonable time. Therefore, the DeGirum library was used, simplifying the pipeline and still achieving enough overall performance. Additionally, DeGirum provides models compatible with the accelerator module that do not require manually converting models into a HAILO-compatible format. This approach made the pipeline easily readable, simplifying later adaptations and extensions to the code, as well as changing the YOLO model.

Putting the prototypes to the real-world test provided the most valuable insights on the reliability and performance of both the Raspberry Pi itself, the design of the prototype in general, and the software pipeline. Besides some small problems when fixing the mountings onto the poles, the design of the prototype proved to be reliable and appealing regarding both design and functionality. The design looks like a regular CCTV camera

with a solar panel on top; therefore, it blends into the environment seamlessly, as such cameras are quite common in the context of streets and traffic. One design flaw of the mounting was the short distance between the solar panel and the housing, requiring the panel to be removed before it was possible to open the lid of the housing when doing maintenance. This is caused by the length of the cables connecting the solar controller with the battery and the Raspberry Pi.
The scripts responsible for booting, shutting down, launching the local hotspot, and launching the detection pipeline proved to be very reliable; there were no problems related to their functionality. On some occasions, the remote connection via VNC was slow and frequently froze with long reloading times. It is unclear if this was related to Raspberry Pi OS and its performance when running the detection pipeline, or to the device that made the connection to the prototype. Since the connection problems mainly appeared on one Windows device, and other devices like a phone with Android or an Apple MacBook experienced fewer problems, it is likely caused by the connecting device rather than the Raspberry Pi itself.
As the YOLO models themselves are well-trained and very reliable, there were no problems with detecting the cars; the statistical evaluation proved this assumption. Difficulties in configuring the prototypes appeared when setting up the camera, mainly finding the right focus to have a clear image. Due to the camera lens being mounted very close to the front window of the camera housing, the auto-focus struggled to find the right focus point, causing the image to be blurry permanently. This was no problem for the model when detecting the cars, but of course impacted the license plate recognition significantly. Therefore, the focus had to be set manually for each of the prototypes. Furthermore, the direction detection lines have to be implemented manually, not only by picking the points for the coordinates for a configuration file, but also by editing the code to fit the number of detection lines, as well as defining the directions the vehicles enter and leave the frame. The threshold that determined whether a car was crossing a line was configured a bit too small, so that faster cars could sometimes not be detected as they crossed the line, and its distance threshold between two frames.
The .csv files also come with some trade-offs, as they are not filtered, and therefore write a line for every detected car in every frame. This causes numerous lines to only contain the internal ID and the timestamp, as neither a license plate, a direction, nor its speed is recorded in the frame. The initial concept was to implement some sort of filtering during the operation of the pipeline; however, this was discarded to save processing power, making manual filtering and processing the files on an external device necessary for proper statistical analysis.

4.4.1 Problems

As the above section put the focus on the insights and trade-offs, this section aims to point out the major problems that mainly appeared throughout the real-world testing period and had a severe impact on the performance and reliability of the prototype.
The biggest problem during the testing period quickly turned out to be the battery capacity in connection with the total power consumption of the Raspberry Pi when

running the detection pipeline. After deployment, the pipeline was set to run from 6:00 until 21:00, and then shut down during the night. Due to the solar panel being the only power source to charge the battery, operating the prototype was heavily dependent on weather, and the overall performance of the solar panel was not sufficient to provide enough energy to both the Raspberry Pi and the battery, causing the Raspberry Pi to frequently shut down automatically due to a lack of power. These power outages would not cause problems themselves, as the Raspberry Pi automatically boots when being reconnected to a power source. But sometimes, the Raspberry Pi's low voltage protection activated before the battery power ran out, and it protected the system by shutting down into a low power mode without automatically rebooting.
Sometimes the low voltage protection would not shut down the system, but reduce power to external devices like the HAILO accelerator module. This caused the detection pipeline to crash, but the script controlling the start of the pipeline was configured to automatically restart the pipeline after crashes. This sometimes caused a loop, where the script started the pipeline, only to crash after a few seconds or minutes of running, as the power was cut off to the HAILO module again.
Another problem that was undiscovered and initially not seen as a problem was the lack of internet access on the Raspberry Pi. Having only local access to the device was a design flaw, as it required being in proximity of the prototype to connect to it and get detection data, and make modifications to the code. The problem manifested itself after the first power outages of the system, as the internal clock of the OS stops, and can´t be synchronised with the actual time and date after reboot. In the power saving mode, the clock continued to run, allowing the system to reboot at the right time. However, after every full power loss of the system, the clock and date were incorrect, which then caused the boot and shutdown times to be at the wrong time of day. In a first attempt to eliminate this problem, these planned times were removed, and the system simply ran until the battery power was too low. This caused more problems related to the under-voltage protection, as the system was frequently running on low power, and it likely also put stress on the battery itself, as the prototype booted on low power, only to quickly shut down again.
Therefore, the decision was made to add a USB-LTE stick with a SIM card in the first revision, as this would eliminate the problem of the clock being wrong, as it is synchronised on boot every time. Furthermore, it provided the possibility to remotely connect to the prototypes and upload and download files from external sources.

CHAPTER 5

Conclusion and Future Work

Conclusion

To conclude, the design, implementation, and testing process of the prototype was successful, considering it is the first iteration of such a prototype, and the lack of former experience regarding the construction and design of such prototypes. After finding a proper library to interface with the HAILO module, the software pipeline was easy to implement, and the overall performance was better than initially expected. Apart from the problems with the battery capacity and the performance issues causing reliability to decrease, the prototype proved to be durable and able to withstand the elements.
Gathering experience with working with image detection tools with YOLO was very valuable for future work, with them being notably more accurate and reliable than expected. The usage of a Raspberry Pi as a processing computer proved to be the right choice, combining both a compact size and a reasonable amount of processing power for the prototype.
Overall, the prototypes serve as a reliable proof-of-concept, despite suffering some initial problems and design flaws. Of course, there is still a lot of room to improve the concept, especially when addressing the battery capacity, and properly filtering and processing the .csv files.

Future Work

Looking into the future, several flaws and problems of the prototype can be addressed. The biggest room for improvement could be made by addressing the battery and solar module. Increasing the battery size would make it possible to run the detection pipeline for longer periods during the day, or even make test runs during nighttime. Especially during winter, when significantly less sunlight is available to charge the battery, maximizing battery capacity is crucial. Eventually, the solar module could be replaced with one with more charging power, enabling it to properly charge the battery and power the

Raspberry Pi at the same time.
Concerning the problem of the undervoltage protection of the OS, the pipeline could be halted before the protection activates, as the system emits an undervoltage warning before cutting off power to peripherals or shutting down the system. This could eliminate or reduce the downtime of the prototype in cases where there would be sufficient power available to power the Raspberry Pi without running under load, but not to run the detection pipeline. This approach could also reduce the stress on the battery, as there would be fewer shifts in total power draw.
Lastly, improvements can be made to the software pipeline. The design of the implementation of the detection lines could be reworked, for example properly modularising the number of lines and directions, allowing for full configuration of the lines with the point selection tool without making any adaptations to the pipeline itself, improving usability. Furthermore, the design of the .csv files could be improved, reducing the number of almost empty lines with respect to the processing power. This would decrease the size of the .csv files and make statistical analysis simpler and less time-consuming.

Overview of Generative AI Tools Used

ChatGPT was used to to rewrite some sentences to improve expression, as well as assistance for writing the Abstract.
Grammarly was used for finding grammar and spelling mistakes.
DeepL was used to translate the Abstract from English to German.

Übersicht verwendeter Hilfsmittel

ChatGPT wurde verwendet, um einige Sätze umzuschreiben und den Ausdruck zu verbessern, sowie Unterstützung beim Verfassen des Abstracts.
Grammarly wurde verwendet, um Grammatik- und Rechtschreibfehler zu identifizieren.
DeepL wurde verwendet, um den Abstract von Englisch nach Deutsch zu übersetzen.

List of Figures

1.1 One of the prototype devices mounted on a light pole 3

3.1 Raspberry Pi 5 with mounted AI HAT+ and Raspberry Pi Camera Module 3 8
3.2 Diagram of the wiring setup connecting the solar panel, battery and the Raspberry Pi 9
3.3 One of the two prototypes mounted on a street light pole 9
3.4 Order of launch of the scripts after the boot of the Raspberry Pi 11
3.5 Tracking pipeline, showing how the input is handled by the Python script 12
3.6 Diagrams comparing YOLO models to YOLOv8 regarding size/precision and latency/precision. Source: [Ltdc] 14
3.7 Performance of ByteTrack compared to other trackers [ZSJ+21]. 16
3.8 Locations where the prototypes are mounted. Source: https://www.wien.gv.at/stadtplan/ 18
3.9 View of the camera of both prototypes 18

4.1 Test setup for temperature monitoring 23
4.2 Plot of the temperature curve, 'Zeit' (Time) on the x-axis refers to the number of data points (one per ten minutes) 24
4.3 Plot of the temperature curve of both the CPU of the Raspberry Pi and the HAILO module while running the tracking script. 25
4.4 The cable connecting the solar controller with the 12V/5V converter in its unmodified and modified state. 26

List of Tables

3.1 Summary of the components used and their price 10

4.1 Results of the model at location Lassallestrasse 19
4.2 Results of the model at location Praterstrasse 20
4.3 Precision, Recall and F1-Score of the detection results 20
4.4 Accuracy of the detection lines at location Lassallestrasse 21
4.5 Accuracy of the detection lines at location Praterstrasse 21

Bibliography

[BfK] Energie Mobilität Innovation und Technologie Bundesministerium fuer Klimaschutz, Umwelt. https://www.bmimi.gv.at/dam/jcr:daeb6637-66e2-4060-b492-95c860ff8301/ece2020$_n$$ationalerbericht_t$$abellenteil.xlsx$, $accessed$24.10.2025.

[DNTL21] Duc-Liem Dinh, Hong-Nam Nguyen, Huy-Tan Thai, and Kim-Hung Le. Towards ai-based traffic counting system with edge computing. *Journal of Advanced Transportation*, 2021(1):5551976, 2021.

[GMB] SNIZEK + PARTNER VERKEHRSPLANUNGS GMBH. https://www.digital.wienbibliothek.at/wbrup/download/pdf/4429234, accessed 24.10.2025.

[Har25] Nicholas Harisch. Bachelor thesis. Technische Universität Wien, Sep 2025.

[Inc] Miovision Technologies Inc. https://miovision.com/scout-plus/, accessed 06.10.2025.

[LMB+14] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[LS18] Jia-Ping Lin and Min-Te Sun. A yolo-based traffic counting system. In *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 82–85, 2018.

[Ltda] Raspberry Pi Ltd. https://www.raspberrypi.com/about/, accessed 06.10.2025.

[Ltdb] Raspberry Pi Ltd. https://www.raspberrypi.com/documentation/computers/os.html, accessed 06.10.2025.

[Ltdc] Ultralytics Ltd. https://docs.ultralytics.com/models/yolov8/, accessed 15.10.2025.

[Ltd24] Raspberry Pi Ltd. https://datasheets.raspberrypi.com/ai-hat-plus/raspberry-pi-ai-hat-plus-product-brief.pdf, accessed 06.10.2025, Oct 2024.

[Ltd25] Raspberry Pi Ltd. https://datasheets.raspberrypi.com/rpi5/raspberry-pi-5-product-brief.pdf, accessed 06.10.2025, Jan 2025.

[NH] Stefan Trenovatz Nicholas Harisch, Armin Kazda. https://gitlab.cg.tuwien.ac.at/nharisch/traffic-counting/-/tree/22c9daa57722973b50dd913bdf87f67adc7c339e/, accessed 24.10.2025.

[OEC] OECD. https://oecd.ai/en/catalogue/metrics/multi-object-tracking-accuracy-mota, accessed 24.10.2025.

[RDGF15] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[Sch19] Georg Scherer. https://www.wienschauen.at/praterstrasse-vom-prachtboulevard-zur-stadtautobahn/, accessed 06.10.2025, Apr 2019.

[SitPI23] Inc (SPI) Software in the Public Interest. https://www.debian.org/news/2023/20230610.en.html, accessed 06.10.2025, Jun 2023.

[Tela] Telraam. https://github.com/telraam/telraam-s2/blob/main/count-performance-validation.md, accessed 24.10.2025.

[Telb] Telraam. https://telraam.net/, accessed 24.10.2025.

[Telc] Telraam. https://telraam.net/en/s2, accessed 24.10.2025.

[Tre25] Stefan Trenovatz. Lightweight license plate recognition for traffic flow analysis. Technische Universität Wien, Sep 2025.

[ZSJ+21] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. Bytetrack: Multi-object tracking by associating every detection box. *CoRR*, abs/2110.06864, 2021.