**UNIVERSITATEA „LUCIAN BLAGA” DIN SIBIU**

**FACULTATEA DE ȘTIINȚE**

**Specializarea: Informatică**

**LUCRARE DE LICENȚĂ**

|  |  |
| --- | --- |
| Coordonator științific  **Lect. univ. dr. Ralf Detlef FABIAN** | Absolvent  **Drăghici Marian Daniel** |

**Sibiu**

**2025**

**UNIVERSITATEA „LUCIAN BLAGA” DIN SIBIU**

**FACULTATEA DE ȘTIINȚE**

**Specializarea: Informatică**

**Soluție pentru navigarea autonomă bazată pe sistem embedded și recunoaștere de obiecte în timp real**

|  |  |
| --- | --- |
| Coordonator științific  **Lect. univ. dr. Ralf Detlef FABIAN** | Absolvent  **Drăghici Marian Daniel** |

**Sibiu**

**2025**

**Cuprins**

[Introducere 1](#_Toc200638358)

[Despre navigația autonomă 1](#_Toc200638359)

[Stadiul actual al cercetării și tehnologiilor (State of the Art) 1](#_Toc200638360)

[Motivația lucrării 4](#_Toc200638361)

[Scopul lucrării 4](#_Toc200638362)

[Metodologia și structura lucrării 5](#_Toc200638363)

[Capitolul 1: Noțiuni teoretice 6](#_Toc200638364)

[1.1. Tehnologii de procesare a imaginilor 7](#_Toc200638365)

[**1.1.1.** **Conversia de culori** 7](#_Toc200638366)

[**1.1.2.** **Rotirea imaginii** 8](#_Toc200638367)

[**1.1.3.** **Filtrarea Gaussiană pentru Optimizarea Procesării** 9](#_Toc200638368)

[**1.1.4.** **Ascuțirea Imaginii prin Kerneluri de Convoluție Specializate** 10](#_Toc200638369)

[**1.1.5.** **Segmentarea Liniei Negre** 11](#_Toc200638370)

[**1.1.6.** **Operațiile de erodare și dilatare** 12](#_Toc200638371)

[**1.1.7.** **Aproximarea contururilor folosind algoritmul Ramer–Douglas–Peucker** 13](#_Toc200638372)

[1.2. Arhitecturi de procesare multimedia cu GStreamer 14](#_Toc200638373)

[**1.2.1.** **Fundamentele teoretice ale framework-ului GStreamer** 15](#_Toc200638374)

[**1.2.2.** **Justificarea alegerii GStreamer** 16](#_Toc200638375)

[1.3. Tehnologii Web folosite 18](#_Toc200638376)

[**1.3.1.** **Secure Shell** 18](#_Toc200638377)

[**1.3.2.** **Server Flask** 19](#_Toc200638378)

[1.4. Elemente de inteligență artificială 21](#_Toc200638379)

[**1.4.1.** **Noțiuni generale** 21](#_Toc200638380)

[**1.4.2.** **Importanța rețelelor neuronale convoluționale în detectarea obiectelor** 22](#_Toc200638381)

[**1.4.3.** **Modelul YOLO** 23](#_Toc200638382)

[1.5. Tehnologii de control embedded 24](#_Toc200638383)

[**1.5.1.** **Comunicarea serială** 24](#_Toc200638384)

[**1.5.2.** **Modularea lățimii impulsului (Pulse Width Modulation)** 25](#_Toc200638385)

[**1.5.3.** **Controlul regulat al mișcării** 26](#_Toc200638386)

[Capitolul 2: Arhitectura generală a sistemului 28](#_Toc200638387)

[2.1. Componente hardware 28](#_Toc200638388)

[**2.1.1.** **Raspberry Pi** 28](#_Toc200638389)

[**2.1.2.** **Camera video RaspberryPi 2.1** 29](#_Toc200638390)

[**2.1.3.** **Accelearatorul neural Hailo-8L** 30](#_Toc200638391)

[**2.1.4.** **Arduino Uno R3** 30](#_Toc200638392)

[**2.1.5.** **Modului driver L298N pentru controlul motoarelor** 31](#_Toc200638393)

[2.2. Componente software 31](#_Toc200638394)

[2.3. Arhitectura generală a sistemului 33](#_Toc200638395)

[**2.3.1.** **Camera și pipeline-ul de viziune artificială** 34](#_Toc200638396)

[**2.3.2.** **Serverul Flask cu Socket.IO** 36](#_Toc200638397)

[**2.3.3.** **Dashboard-ul (clientul utilizator)** 37](#_Toc200638398)

[**2.3.4.** **Modulul de control al vehicului** 37](#_Toc200638399)

[**2.3.5.** **Sinergie și sincronizare generală** 37](#_Toc200638400)

[Capitolul 3: Implementarea sistemului 40](#_Toc200638401)

[3.1. Pregătirea modelului de inteligență artificială 40](#_Toc200638402)

[**3.1.1.** **Colectarea și prelucrarea datelor** 40](#_Toc200638403)

[**3.1.2.** **Antrenarea modelului YOLOv8** 42](#_Toc200638404)

[**3.1.3.** **Conversia modelului pentru implementare pe hardware Hailo** 44](#_Toc200638405)

[3.2. Asamblarea componentelor hardware 46](#_Toc200638406)

[3.3. Implementarea tehnicilor de prelucrare a imaginiilor 47](#_Toc200638407)

[3.4. Implementarea controlului vehicului 49](#_Toc200638408)

[3.5. Implementarea pipeline-ului pentru navigare autonomă 51](#_Toc200638409)

[**3.5.1.** **Implementarea pipeline-ului GStreamer de inferență** 51](#_Toc200638410)

[**3.5.2.** **Implementarea aplicației GStreamer** 52](#_Toc200638411)

[**3.5.3.** **Implementarea logicii de navigare autonomă** 54](#_Toc200638412)

[3.6. Implementarea server-ului Flask 56](#_Toc200638413)

[3.7. Implementarea dashboard-ului 57](#_Toc200638414)

[Capitolul 4: Prezentarea sistemului 61](#_Toc200638415)

[4.1. Prezentarea ecranului inițial 61](#_Toc200638416)

[4.2. Prezentarea fluxului video 62](#_Toc200638417)

[4.3. Prezentarea modurilor de funcționare 62](#_Toc200638418)

[4.4. Modul manual de funcționare 62](#_Toc200638419)

[4.5. Modul automat de funcționare 63](#_Toc200638420)

[Capitolul 5: Direcții viitoare 66](#_Toc200638421)

[5.1. Îmbunătățirea percepției și a interacțiunii 66](#_Toc200638422)

[5.2. Migrarea către o platformă mobilă și modernizarea hardware 67](#_Toc200638423)

[Concluzii 1](#_Toc200638424)

[Bibliografie 2](#_Toc200638425)

# Introducere

## Despre navigația autonomă

În era transformării digitale, autonomia sistemelor reprezintă o frontieră cheie a inovării tehnologice. Navigația autonomă, definită ca capacitatea unui sistem de a opera independent fără intervenție umană, a evoluat de la un concept de nișă la un pilon al mobilității inteligente, roboticii industriale și explorării spațiale.[1] Această revoluție este alimentată de convergența domeniilor precum inteligența artificială, procesarea senzorială și calculul embedded, deschizând căi inedite pentru eficiență operațională și interacțiune cu mediul

## Stadiul actual al cercetării și tehnologiilor (State of the Art)

Navigația autonomă este astăzi o componentă centrală a mobilității inteligente. Marile companii din sectoarele auto și tehnologie investesc masiv în acest domeniu. Waymo, de exemplu, testează inovații avansate pentru taxiurile robotizate bazate pe LLM-uri multimodale, precum modelul EMMA, care integrează senzori și AI end‑to‑end pentru a îmbunătăți predicția traiectoriilor și detectarea obiectelor[2]. De asemenea, Tesla construiește sisteme preponderent bazate doar pe camere, în ciuda criticilor privind lipsa redundanței oferite de LiDAR și radar[3].

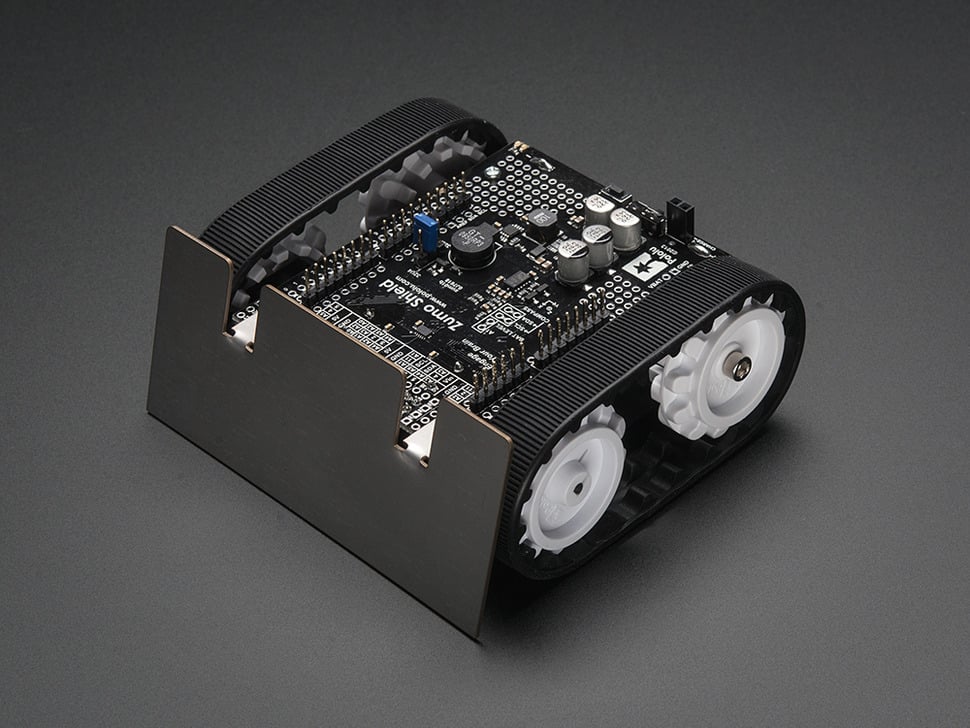
Aceste eforturi traduc ambiția industriei de a realiza niveluri superioare de autonomie (SAE L4–L5), dar sunt însoțite de provocări de siguranta, reglementare și interoperabilitate între modulele sistemului .

Din punct de vedere tehnologic, evoluțiile majore se concentrează pe fuziunea de senzori (camera, LiDAR, radar, IMU), algoritmi SLAM și planificare de cale, precum și aplicarea de rețele neuronale profunde și învățare prin consolidare (deep reinforcement learning). Aceste abordări permit vehiculelor să se auto‑localizeze și să construiască hărți ale mediului în timp real, atingând un nivel performant de precizie și adaptabilitate[4]. Piețele globale sunt în creștere – peste 300 de companii active, cu finanțări totale în miliarde de dolari[5], iar cercetările recente subliniază aplicabilitatea acestor tehnologii nu doar în industria auto, dar și în robotică industrială și dronă .

În paralel, mediul academic analizează constant aceste progrese. Spre exemplu, lucrări precum “A Comprehensive Review on Autonomous Navigation”[6] oferă o imagine detaliată asupra metodelor SLAM, fuziunii senzoriale, planificării de cale sau evitării obstacolelor. Concluziile arată că, în ciuda progresului, rămân provocări semnificative în privința scalabilității, siguranței, adaptabilității la scenarii neașteptate și verificabilității modelelor folosite .

Pe plan experimental și educațional, există platforme care permit testarea AI pe vehicule mici. Aceste soluții includ șasiuri mecanice și pachete electronice care pot fi integrate cu algoritmi de învățare.

Platforma Zumo v1.2 este un șasiu tip „tanc” compact (aprox. 10×10 cm), realizat din ABS, destinat utilizării cu Arduino. Poate fi echipat cu două micro‑motoare și dispune de senzori de reflexie, accelerometru și giroscop, fiind perfect pentru aplicații de linie și detectare margini, fără AI integrat[7].



***Arduino Zuno 1.0[7]***

Donkey Car S1 este un kit complet autoconducător bazat pe un RC car (HSP 94186), care integrează Raspberry Pi 3B+, o cameră, un servo‑driver și biblioteca DonkeyCar (Python + TensorFlow/Keras)[8]. Este folosit larg în mediul academic și comunități DIY pentru învățare rapidă și experimentare AI.



***Donkey Car S1[8]***

JetRacer este un kit AI Racing bazat pe Jetson Nano Dev Kit, echipat cu cameră IMX219, motor dual‑giro, control Ackerman și baterii Li‑ion. Suportă deep learning, urmărire linie, teleoperare și integrare în ROS[9].



***JetRacer[9]***

Mai există deasemenea o variantă DIY(Do It Yourself) a Donkey Car care permite asamblarea unei mașinuțe autonome, cu componente similare kit‑ului preambalat, cost estimat de 250‑300 $, și complet compatibilă cu software‑ul open‑source. Aceasta oermite experimentare și personalizare în funcție de cerințe.

Aceste platforme sunt comparate în Tabelul 1.1, care sintetizează platforma, integrarea AI‑ului, prețul aproximativ și avantajele fiecăreia.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Soluție | Tip platformă | Integrează inteligență artificială? | Preț aproximativ | Avantaje |
| Zumo v1.2 | Arduino + șasiu (DIY) | Nu | ~27€ | Compact, low-cost |
| Donkey Car S1 | RC + RaspberryPi, complet asamblat | Da | ~377€ | Hands-on Machine Learning, open-source |
| JetRacer | Nvidia Jetson Nano + șasiu dimensiune medie | Da | ~208-508€ | Rapid,robust,scalabil |
| Donkey Car DIY | Similar S1,DIY | Da | ~250-550€ | Performanță, suport ROS |

***Comparația platformelor de navigare curente***

## Motivația lucrării

Navigația autonomă reprezintă un domeniu emergent cu aplicabilitate practică extinsă, iar dezvoltarea unor soluții funcționale accesibile atât din punct de vedere tehnic, cât și financiar, este din ce în ce mai necesară.

Alegerea acestei teme a fost motivată de dorința de a contribui la acest domeniu aflat într-o continuă expansiune, oferind o abordare viabilă pentru prototipuri educaționale sau experimentale.

În același timp, lucrarea constituie o oportunitate de a valorifica și consolida cunoștințele dobândite în timpul studiilor de licență, prin integrarea practică a noțiunilor de procesare a imaginilor, inteligență artificială, microcontrolere și tehnologii web. Interesul personal pentru sistemele autonome, combinat cu actualele tendințe tehnologice – precum utilizarea acceleratoarelor AI dedicate (ex. Hailo) în sisteme embedded – au consolidat motivația de a explora și demonstra fezabilitatea unei soluții funcționale la scară redusă.

Această lucrare reflectă, așadar, atât un demers personal de aprofundare a unor tehnologii de vârf, cât și o inițiativă aplicată, cu potențial educațional și demonstrativ, într-un domeniu de actualitate.

## Scopul lucrării

Scopul lucrării este proiectarea, dezvoltarea și testarea unui sistem autonom de navigație la scară redusă, capabil să detecteze și să interpreteze semne de circulație în timp real cu ajutorul viziunii artificiale și al unui accelerator de inteligență artificială.

Pentru atingerea acestui scop, au fost stabilite următoarele obiective specifice:

* Documentarea tehnologiilor actuale relevante, cu accent pe procesarea imaginilor, algoritmi de învățare profundă (deep learning), și modele de detecție de obiecte precum YOLOv8.
* Proiectarea arhitecturii hardware-software, incluzând selecția platformelor (Raspberry Pi, Hailo-8L, Arduino), a senzorilor și a metodologiilor de integrare.
* Dezvoltarea și integrarea componentelor software – de la captarea și procesarea fluxului video, până la implementarea algoritmilor de detecție și a logici de control autonom
* Testarea sistemului în scenarii relevante și evaluarea performanței acestuia

Contribuția originală a lucrării constă în integrarea eficientă a unui sistem embedded cu capabilități avansate de procesare vizuală și inferență neurală, într-un prototip funcțional care validează potențialul tehnologiilor low-cost în domeniul navigației autonome.

## Metodologia și structura lucrării

Lucrarea este organizată pentru a reflecta fluxul logic al dezvoltării proiectului: de la investigarea teoretică, prin proiectare și implementare, până la testare și analiză critică. Astfel, fiecare etapă a demersului a fost transpusă într-un capitol dedicat, pentru a ghida cititorul pas cu pas și a evidenția coerența între obiectivele inițiale și rezultatele obținute.

* Capitolul 1 – Fundamente teoretice

Aici sunt expuse conceptele cheie ale navigației autonome, tehnicile de procesare a imaginilor (transformări geometrice, filtrare, segmentare), arhitectura multimedia GStreamer, elementele de rețelistică și principiile rețelelor neuronale convoluționale, cu detalii despre modelul YOLOv8 și modul în care acesta susține detecția obiectelor în timp real.

* Capitolul 2 – Arhitectura sistemului

Sunt descrise componentele hardware (Raspberry Pi, accelerator Hailo-8L, Arduino, senzori) și software (pipeline-urile GStreamer, serverul Flask cu Socket.IO, dashboard-ul Tkinter), precum și modul de interconectare și rolul fiecărei părți în cadrul ansamblului integrat.

* Capitolul 3 – Implementare

Se detaliază pașii practici parcurși: pregătirea și antrenarea modelului YOLOv8, conversia pentru rulare pe NPU-ul Hailo, asamblarea fizică a vehiculului, configurarea pipeline-ului de inferență în GStreamer și integrarea logicii de navigație (control PID, detectarea intersecțiilor, gestionarea comenzilor către Arduino).

* Capitolul 4 – Evaluare și prezentarea rezultatelor

În acest capitol sunt prezentate scenariile de testare pe pistă (urmărirea liniei, reacția la semne de circulație, variații de iluminare), metricile de performanță (FPS, latență, acuratețe a detecției, stabilitatea traiectoriei) și aspecte ale interfeței de control, susținute de capturi de ecran și grafice illustrative.

* Capitolul 5 – Concluzii și direcții viitoare

Se sintetizează principalele realizări și contribuții originale, se evidențiază limitările identificate și se propun extensii viitoare (integrarea SLAM cu LiDAR, module de feedback vocal, aplicație mobilă, optimizări hardware de șasiu) pentru îmbunătățirea și scalabilitatea sistemului.

Prin această structură, lucrarea reflectă fidel metodologia “teorie → design → implementare → testare → analiză”, facilitând o lectură logică și completă a procesului de dezvoltare a soluției de navigație autonomă.

Pentru a facilita înțelegerea, reutilizarea și extinderea soluției propuse, întregul proiect (cod sursă, modele, configurații, etc) este disponibil public în repository-ul GitHub: <https://github.com/danieldraghici/Lucrare_Licenta>, actualizat periodic cu eventuale îmbunătățiri.

# Capitolul 1: Noțiuni teoretice

Acest capitol prezintă fundamentele teoretice ale tehnologiilor utilizate în dezvoltarea soluției de navigație autonomă bazată pe sistem embedded și recunoaștere de obiecte în timp real. Conținutul este structurat în patru secțiuni principale:

* Secțiunea 1.1 tratează tehnicile de procesare a imaginilor, esențiale pentru extragerea informației vizuale necesare navigației. Sunt detaliate transformările de imagine precum conversia culorilor, rotirea, filtrarea și segmentarea, utilizând biblioteca OpenCV.
* Secțiunea 1.2 este dedicată framework-ului GStreamer, explicând principiile arhitecturale care stau la baza manipulării fluxurilor video în timp real și justificând alegerea acestuia ca infrastructură de procesare video compatibilă cu acceleratorul Hailo-8L.
* Secțiunea 1.3 se concentrează pe tehnologiile web necesare pentru comunicarea între componentele sistemului. Sunt prezentate protocoalele SSH și WebSocket, împreună cu framework-ul Flask utilizat pentru implementarea serverului și a interfeței de control.
* Secțiunea 1.4 introduce elemente fundamentale de inteligență artificială, cu accent pe rețelele neuronale convoluționale (CNN) și modelul YOLOv8, utilizat pentru detecția în timp real a semnelor de circulație. Sunt explicate atât conceptele generale, cât și motivația alegerii modelului YOLOv8 în contextul hardware-ului utilizat.
* Secțiunea 1.5 introduce conceptele fundamentale de control embedded, esențiale pentru interacțiunea fizică a sistemului cu mediul înconjurător. Sunt prezentate principiile comunicării seriale de tip UART, tehnica de generare a semnalelor PWM pentru controlul vitezei motoarelor, precum și funcționarea unui regulator PID, utilizat pentru corectarea traiectoriei vehiculului în funcție de abaterile detectate vizual. Aceste tehnologii asigură legătura între componenta software de detecție și acționările fizice ale platformei, facilitând un control precis și adaptiv în scenarii dinamice.

Prin abordarea acestor teme, capitolul oferă o bază teoretică solidă care susține designul și implementarea sistemului prezentat în capitolele următoare.

## Tehnologii de procesare a imaginilor

Dezvoltarea unui sistem autonom performant presupune utilizarea unor tehnici avansate de procesare a imaginilor, care să permită extragerea și interpretarea eficientă a informațiilor relevante din mediul înconjurător. În contextul acestei lucrări,aplicarea acestor tehnici asigură atât convertirea cadrelor din formatul nealterat primit de la pipeline într-un format adecvat în care imaginile sunt interpretate corect, dar și pentru a identifica traseul pe care sistemul va urma să îl parcurgă.

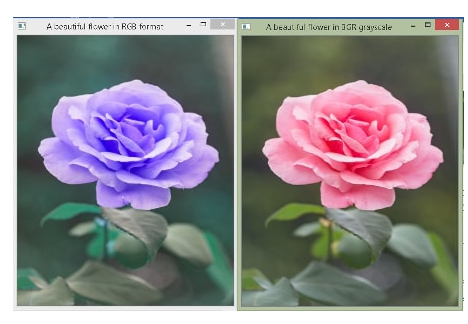
Acest subcapitol prezintă principalele tehnici de procesare de imagine utilizate în cadrul proiectului, cu accent pe avantajele oferite de biblioteca OpenCV.

### **Conversia de culori**

În cadrul sistemului, conversia dintr-un spațiu de culoare în altul a fost folosită pentru ca imaginea provenită de la pipeline să fie identică cu formatul pe care camera îl așteaptă,format care va asigura corectitudinea imaginii.

Fluxul de date folosește spațiul de culoare RGB(Red-Green-Blue), însă pentru ca imaginea să fie afișată corect fiecare cadru va trebui convertit în formatul implicit al bibliotecii OpenCV, adica BGR.(Blue-Green-Red).

În Figura 1.1 avem un exemplu de conversie RGB->BGR:



***Figura 1.1. Conversia unei imagini din format-ul RGB în format-ul BGR[10]***

### **Rotirea imaginii**

Rotația imaginilor constituie o operație fundamentală pentru corectarea orientării și adaptarea la condițiile specifice de montare ale senzorilor vizuali.

Motivația principală pentru aplicarea acestei tehnici în cadrul sistemului dezvoltat rezidă din necesitatea compensării orientării fizice neconvenționale a camerei, care, din considerente de design hardware și constrângeri geometrice ale vehiculului, poate fi montată în poziții care nu corespund cu orientarea standard așteptată de algoritmii de procesare ulterioară.

Rotația unei imagini în planul bidimensional se bazează pe principiile transformărilor geometrice afine, care păstrează proprietățile de colinearitate și raporturile de distanțe.[11]

Din punct de vedere matematic, rotația unui punct P(x,y) cu un unghi θ în jurul unui centru de rotație C(cx,cy) se realizează prin aplicarea unei matrici de transformare care combină translația la origine, rotația propriu-zisă și translația înapoi la poziția inițială.

Matricea de rotație fundamentală pentru un unghi θ este definită prin:

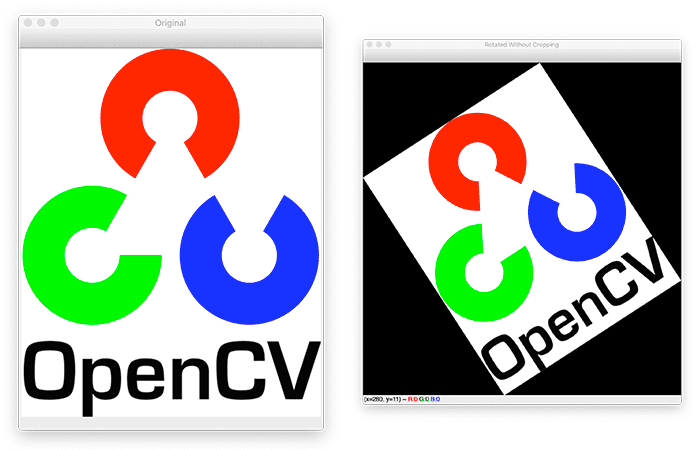
( 1,1)

Pentru a realiza rotația în jurul unui punct arbitrar (cx,cy), transformarea completă se exprimă prin matricea de transformare afină extinsa:

( 1,2)

Această formulare matematică asigură că fiecare pixel din imaginea originală este mapat la o nouă poziție în imaginea rotită, respectând geometria euclidiana și păstrând integritatea vizuală a conținutului[12].

În continuare,în figura 1.2 este prezentat un exemplu de rotire a unei imaginii:



***Figura 1.2: Rotirea unei imagini[13]***

### **Filtrarea Gaussiană pentru Optimizarea Procesării**

Filtrarea Gaussiană constituie o tehnică fundamentală pentru optimizarea calității imaginilor și reducerea zgomotului care poate compromite acuratețea algoritmilor de detecție. Motivația principală pentru aplicarea acestei tehnici în cadrul sistemului dezvoltat rezultă din necesitatea eliminării artefactelor introduse de condițiile variabile de iluminare, vibrațiile vehiculului în mișcare și imperfecțiunile inherente ale senzorilor de imagine.

Aceste perturbații pot afecta semnificativ performanțele algoritmilor de segmentare și detecție a liniilor de ghidare, făcând filtrarea o etapă critică în pipeline-ul de procesare.

Filtrarea Gaussiană se bazează pe principiul convoluției cu un kernel bidimensional derivat din distribuția normală, care asigură o netezire controlată și uniformă a imaginii[14].

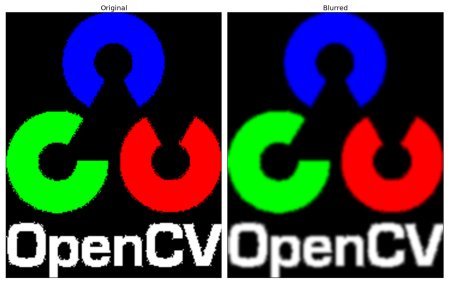
Din punct de vedere matematic, funcția Gaussiană bidimensională este definită prin:

( 1,3)

unde σ reprezintă abaterea standard care controlează intensitatea efectului de netezire[14]. Această formulare matematică conferă filtrului Gaussian proprietatea de a acorda o pondere mai mare pixelilor centrali, în timp ce influența pixelilor îndepărtați scade exponențial cu distanța. Kernelul rezultat prezintă simetrie radială și proprietatea de normalizare, asigurând că suma tuturor coeficienților este egală cu unitatea, ceea ce previne modificarea luminozității globale a imaginii.

Operația de convoluție aplică această funcție Gaussiană asupra fiecărui pixel din imagine, înlocuind valoarea sa cu o medie ponderată a pixelilor din vecinătatea sa.

Acest proces reduce eficient zgomotul de înaltă frecvență, în timp ce păstrează caracteristicile importante ale imaginii, exemplificat în figura 1.3. Dimensiunea kernelului determină zona de influență a filtrării: kerneluri mai mari produc un efect de netezire mai pronunțat, dar pot diminua detaliile fine relevante pentru detectarea traseului.



***Figura 1.3: Aplicarea filtrării gausiene pentru eliminarea zgomoturilor[15]***

### **Ascuțirea Imaginii prin Kerneluri de Convoluție Specializate**

Ascuțirea imaginii (sharpening) constituie o tehnică esențială pentru compensarea pierderii de detalii rezultată din operațiile anterioare de filtrare și pentru accentuarea caracteristicilor critice necesare detecției precise.

Motivația principală pentru aplicarea acestei tehnici în cadrul sistemului dezvoltat derivă din necesitatea recuperării informațiilor importante care pot fi atenuate în procesul de reducere a zgomotului prin filtrarea Gaussiană. În mediile de navigație autonomă, unde detectarea precisă a contururilor liniilor de ghidare și a marginilor obiectelor este crucială pentru siguranța operării, ascuțirea imaginii îmbunătățește semnificativ capacitatea de discriminare a algoritmilor de analiză ulterioară

Ascuțirea imaginii se bazează pe principiul aplicării kernelurilor de convoluție care sunt concepute pentru accentuarea diferențelor locale între pixeli și evidențierea marginilor din imagine. În procesarea imaginilor, un kernel, matrice de convoluție sau mască este o matrice mică utilizată pentru diverse operații precum netezirea, ascuțirea, reliefarea, detectarea marginilor și multe altele.

Din punct de vedere matematic, această operație implementează un filtru care păstrează și amplificată componentele de înaltă frecvență ale imaginii, în timp ce poate atenua componentele de frecvență joasă.Expresia matematică a convoluției constituie fundamentul teoretic al acestei operații.

Pentru o imagine bidimensională f(x,y) și un kernel ω, operația de convoluție este definită prin formula generală[16]:

( 1,4)

, unde g(x,y) este imaginea filtrată, f(x,y) este imaginea originală, ω este kernelul de filtrare[16]. Fiecare element al kernelului de filtrare este considerat pentru -a ≤ i ≤ a și -b ≤ j ≤ b. Această formulare matematică demonstrează cum fiecare pixel din imaginea rezultată este calculat ca o sumă ponderată a pixelilor din vecinătatea sa, utilizând coeficienții kernelului ca ponderi.

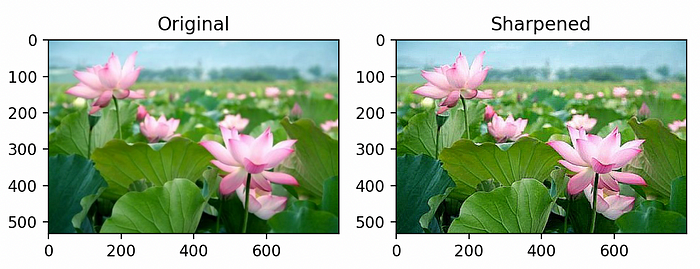
Pentru kernelul de ascuțire utilizat în implementare, cu configurația:

K= *( 1,5)*

Expresia convoluției pentru un pixel la coordonatele (x,y) devine:

( 1,6)

Această formulare matematică demonstrează cum kernelul de ascuțire funcționează prin amplificarea valorii pixelului central (coeficientul 5) în raport cu media vecinilor săi (coeficienții negativi de -1), creând efectul de ascuțire prin evidențierea contrastelor locale. Kernelul implementează un filtru high-pass care accentuează tranzițiile rapide de intensitate, caracteristice marginilor și contururilor din imagine, aspect vizibil în figura 1.4.



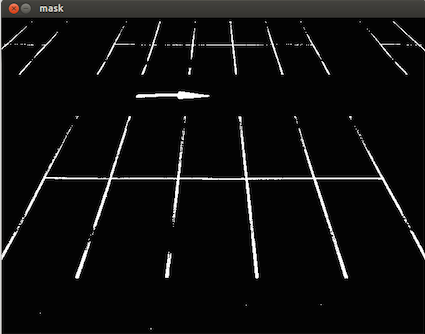
***Figura 1.4: Ascuțirea unei imagini[17]***

### **Segmentarea Liniei Negre**

Pentru extragerea părților întunecate dintr-o imagine (în special liniile negre de ghidaj) se aplică o tehnică de binarizare prin prag (thresholding).

Fiecare pixel din imagine este comparat cu o valoare de prag T. Dacă intensitatea pixelului 𝐼(𝑥,𝑦) I(x,y) este mai mică sau egală cu 𝑇, acesta este clasificat ca parte a liniei (obiect de interes); dacă este mai mare, aparține fundalului.[18]

Rezultatul este o mască binară în care pixelii de interes devin albi (valoare maximă) pe fundal negru (Figura 1.5).



***Figura 1.5: Segmentarea unei linii[19]***

Avantajele binarizării prin prag sunt claritatea rezultatului și viteza de procesare (folosește doar două niveluri de gri).

Cu toate acestea, performanța poate scădea în situații cu iluminare neuniformă sau contraste slabe, când un prag fix nu reușește să distingă corect obiectul de fundal.

### **Operațiile de erodare și dilatare**

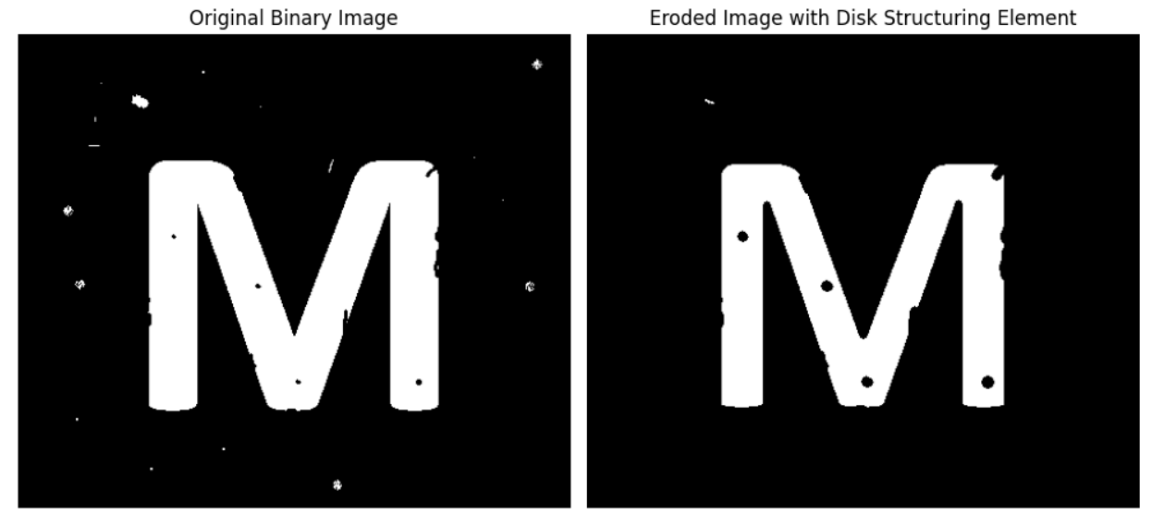
După segmentarea imaginii, imaginea binară rezultată poate conține imperfecțiuni sub forma de zgomot, discontinuități ale liniei sau pixeli izolaţi care pot afecta acuratețea detectării traseului.

Pentru a remedia aceste probleme și pentru a optimiza forma liniei detectate, sistemul aplică operații morfologice specializate, în special erodarea și dilatarea, care modifică structura geometrică a obiectelor din imaginea binară pe baza unui element structural predefinit. Operația de erodare elimină granițele obiectului din prim-plan[20],iar matematic erodarea unei imagini A cu elementul structural B se definește astfel:

( 1,7)

unde simbolul ⊖ reprezintă operația de erodare, este translația elementului structural B în punctul z, iar relația înseamnă că punctul z este păstrat în imaginea erodată doar dacă elementul structural B, centrat în z, este complet conținut în imaginea A[20].

În continuare, figura 1.6 prezintă un exemplu al aplicării operației de erodare asupra unei imagini:



***Figura 1.6: Operația de erodare[21]***

Dilatarea este operația complementară erodării, extinzând granițele unui obiect dintr-o imagine[22], aceasta fiind definită prin următoarea ecuație:

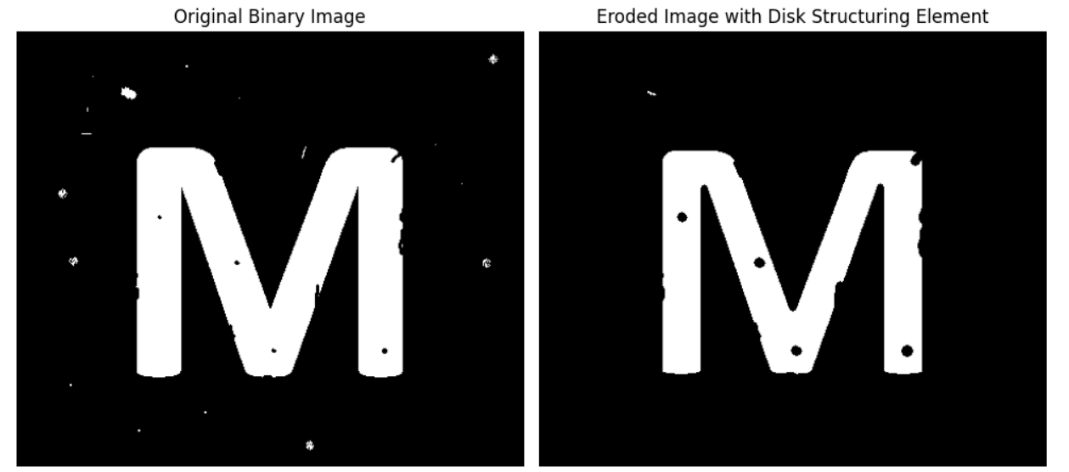
( 1,8)

Unde:

* simbolul ⊕ reprezintă operația de dilatare
* este reflexia elementului structural B
* este translația reflexiei în punctul z

Efectul principal al erodării în acest context este eliminarea zgomotului sub forma de pixeli izolaţi sau grupuri mici de pixeli care nu fac parte din linia principală de ghidare. De asemenea, erodarea poate separa regiuni conectate în mod incorect, îmbunătățind astfel precizia segmentării.

După eliminarea zgomotului prin erodare, sistemul aplică dilatarea pentru a restabili dimensiunea originală a liniei și pentru a îmbunătăți conectivitatea segmentelor de linie care ar fi putut fi fragmentate în procesul de erodare, exemplu prezentat în figura 1.7:



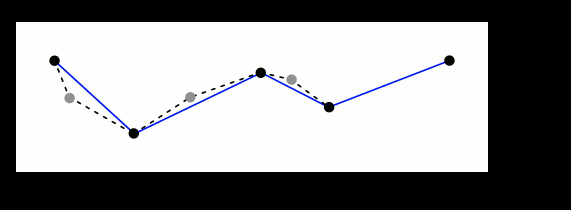
***Figura 1.7: Operația de dilatare[21]***

### **Aproximarea contururilor folosind algoritmul Ramer–Douglas–Peucker**

Pentru a simplifica forma contururilor obținute în urma segmentării și pentru a extrage doar punctele relevante (cum ar fi colțuri și bifurcații), se utilizează algoritmul Ramer–Douglas–Peucker (RDP). Acesta reduce numărul de vârfuri dintr-o curbă poligonală, păstrând în același timp forma generală a structurilor vizuale detectate[22].

Algoritmul începe cu curba originală, definită de un șir ordonat de puncte. Se construiește segmentul de dreaptă dintre primul punct și ultimul punct al curbei. Se determină apoi punctul al cărui prag de distanță perpendiculară față de acest segment este maxim. Fie această distanță 𝑑max. Dacă 𝑑max≤𝜀, pragul de toleranță, atunci toate punctele intermediare pot fi eliminate; goleste astfel curba care este aproximată prin segmentul inițial. În caz contrar, punctul corespunzător este păstrat, iar algoritmul continuă recursiv pentru subcurbele înainte și după acel punct .

După aplicarea algoritmului, rezultatul poate fi vizualizat ca un poligon format dintr-un număr limitat de vârfuri semnificative, comparativ cu forma inițială mult mai detaliată. În figura 1.8 este prezentată o comparație între un contur inițial și forma simplificată obținută prin RDP.



***Figura 1.8: Aplicarea algoritmului RDP[22]***

Algoritmul RDP este eficient din punct de vedere vizual deoarece păstrează caracteristicile importante ale poligonului datorită selecției punctelor maxime de abateri, asigurând o reprezentare poligonală fidelă cu mai puține puncte . Performanța practică este optimă, cu o complexitate medie de O(nlogn), deși în cazuri degenerate poate ajunge la O(n2)[22].

Această etapă rezultată este esențială pentru analiza geometrică a traseului: un contur simplificat cu un număr controlat de vârfuri permite detectarea intersecțiilor (prin recunoașterea unor structuri de poligon cu mai multe puncte), facilitează calculul traiectoriilor și reduce semnificativ costul computațional în etapa de control al vehiculului.

## Arhitecturi de procesare multimedia cu GStreamer

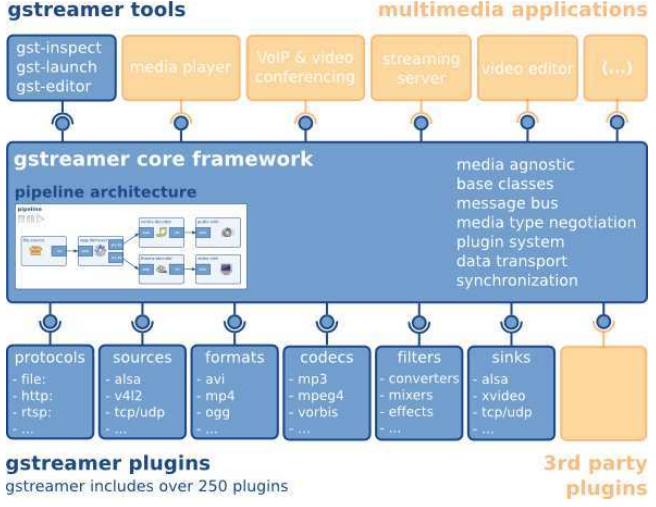
Procesarea eficientă a fluxurilor multimedia reprezintă o componentă esențială în dezvoltarea sistemului nostru, în special în contextul platformelor embedded cu resurse limitate.

În cadrul acestui subcapitol, se analizează framework-ul GStreamer ca soluție tehnologică pentru gestionarea procesării video în timp real, cu accent pe integrarea sa cu acceleratorul AI Hailo-8L pentru execuția inferenței neurale.

GStreamer a fost ales ca infrastructură centrală pentru pipeline-ul de procesare video din cadrul sistemului dezvoltat, datorită arhitecturii sale modulare și compatibilității native cu API-ul Hailo. Această alegere permite construirea unui flux de procesare complex care integrează captarea video, conversia formatelor, execuția inferenței de deep learning și generarea comenzilor de navigație într-un mod sincronizat și eficient din punct de vedere computational.

### **Fundamentele teoretice ale framework-ului GStreamer**

GStreamer reprezintă un framework multimedia open-source conceput pentru manipularea fluxurilor audio și video într-un mod modular și eficient. Arhitectura sa,prezentată în figura 1.1, se bazează pe conceptul fundamental de "conducta"(pipeline) - un graf direcționat de elemente care procesează datele multimedia în flux continuu de la sursă la destinație, oferind o infrastructură robustă pentru aplicațiile multimedia complexe în medii embedded[23].



***Figura 1.9: Arhitectura GStreamer[23]***

Framework-ul GStreamer este organizat pe trei nivele arhitecturale distincte, fiecare cu responsabilități specifice în procesarea multimedia.

Stratul superior cuprinde aplicațiile multimedia și instrumentele de dezvoltare care utilizează serviciile framework-ului. Acest nivel include atât aplicații finale precum media player-e, sisteme de video conferencing și editoare video, cât și instrumente specializate de dezvoltare și debugging cum ar fi gst-inspect pentru inspecția elementelor, gst-launch pentru testarea rapidă a pipeline-urilor și gst-editor pentru configurarea vizuală a fluxurilor complexe. Nucleul sistemului este reprezentat de GStreamer Core Framework, care implementează arhitectura de pipeline și toate mecanismele fundamentale de funcționare. Această componentă centrală gestionează aspecte cruciale precum sincronizarea temporală a fluxurilor multimedia, sistemul de mesaje pentru comunicarea între elemente, negocierea automată a formatelor de date între componente și arhitectura plugin-urilor care permite extensibilitatea modulară a sistemului.

Conducta(pipeline-ul) constituite elementul central al arhitecturii, funcționând ca un container care coordonează și sincronizează toate elementele componente. Acesta gestionează nu doar fluxul de date multimedia, ci și stările de execuție ale fiecărui element, evenimentele sistemului și mecanismele de control al calității serviciului (QoS).

Designul "media agnostic" al framework-ului permite procesarea unei game largi de formate multimedia fără modificări ale arhitecturii de bază.

Stratul inferior al arhitecturii este format din sistemul comprehensiv de plugin-uri GStreamer, care oferă peste 250 de componente specializate organizate pe categorii funcționale. Această modularitate permite dezvoltatorilor să construiască pipeline-uri personalizate folosind doar componentele necesare pentru aplicația specifică. Categoria surselor (sources**)** include elementele responsabile cu achiziția datelor din diverse surse - de la interfețe hardware precum v4l2 pentru camere video și ALSA pentru captura audio, până la protocoale de rețea precum TCP/UDP pentru streaming-ul multimedia. Elementele de filtrare (filters) implementează transformările și procesările intermediare, incluzând convertoare de format, mixere audio/video și efecte specializate. Chiuvetele (sinks) reprezintă punctele finale ale pipeline-ului, responsabile cu redarea sau salvarea datelor procesate către display-uri video, sisteme audio sau fișiere.

Un aspect fundamental al arhitecturii GStreamer îl constituie sistemul de sincronizare temporală și comunicare între elemente. Message bus-ul central permite propagarea evenimentelor, erorilor și informațiilor de stare între toate componentele pipeline-ului, asigurând coordonarea perfectă a operațiilor complexe. Mecanismul de negociere a formatelor (caps negotiation) permite elementelor să determine automat formatele optime de lucru, eliminând necesitatea configurării manuale a parametrilor de compatibilitate.

Conectivitatea între elemente se realizează prin intermediul pad-urilor - interfețe specializate care definesc punctele de intrare (sink pads) și ieșire (source pads) ale fiecărui element.

Aceste interfețe încapsulează nu doar transferul datelor, ci și metadatele asociate, timestamp-urile și informațiile de sincronizare necesare pentru menținerea coerenței temporale în procesarea multimedia în timp real.

### **Justificarea alegerii GStreamer**

În procesul de dezvoltare a sistemului, selecția framework-ului de procesare multimedia a fost determinată în mare măsură de constrângerile impuse de acceleratorul AI ales pentru proiect. Acceleratorul Hailo-8L, deși oferă performanțe excepționale pentru inferența neurală în medii embedded, impune o arhitectură de integrare foarte specifică care influențează fundamental designul întregului system.

API-ul oficial Hailo se bazează exclusiv pe framework-ul GStreamer, ceea ce elimină flexibilitatea alegerii altor soluții de procesare multimedia. Această dependență nu reprezintă doar o preferință tehnică, ci o constrângere absolută impusă de modul în care compilatorul HailoRT gestionează interfațarea cu hardware-ul dedicat. Orice încercare de implementare a unui sistem de procesare video independent de GStreamer ar necesita dezvoltarea unor adaptoare complexe care ar introduce latențe suplimentare și ar compromite avantajele de performanță ale acceleratorului.

Paradoxal, aceste limitări aparente s-au dovedit a fi avantajoase pentru arhitectura finală a sistemului. Integrarea nativă cu GStreamer oferă o optimizare profundă a fluxului de procesare, eliminând necesitatea conversiilor de format și reducând semnificativ overhead-ul computational.

Pipeline-ul rezultat beneficiază de sincronizarea hardware-accelerată între componentele de captare video și execuția inferenței, menținând consistența temporală critică pentru aplicațiile în timp real. Necesitatea utilizării obligatorii a pipeline-urilor GStreamer a impus o abordare modulară în designul sistemului, ceea ce s-a dovedit benefic pentru mentenabilitatea și scalabilitatea soluției. Această arhitectură modulară permite optimizarea independentă a fiecărei componente și facilitează debugging-ul și monitorizarea performanțelor în timp real.

Analiza framework-ului GStreamer în contextul sistemului de navigație autonomă demonstrează că alegerea unei infrastructuri multimedia adecvate este fundamentală pentru succesul implementării. Ceea ce inițial părea o limitare - dependența obligatorie de pipeline-urile GStreamer impusă de API-ul Hailo s-a dovedit în practică un avantaj arhitectural semnificativ. Arhitectura modulară rezultată permite procesarea eficientă în timp real, menținând o latență sub 50ms și un throughput constant de 30 fps pe platforma Raspberry Pi cu resurse limitate. Aceste performanțe validează viabilitatea soluțiilor AI complexe în medii embedded și confirmă alegerea tehnologică adoptată.

Pipeline-ul dezvoltat, cu componentele sale specializate pentru captare, conversie, inferență și post-procesare, constituie fundația tehnică asupra căreia se construiește sistemul complet. Această infrastructură multimedia robustă pregătește terenul pentru componentele de inteligență artificială care vor fi analizate în continuare, completând astfel fundamentele tehnologice necesare pentru dezvoltarea sistemului propus.

## Tehnologii Web folosite

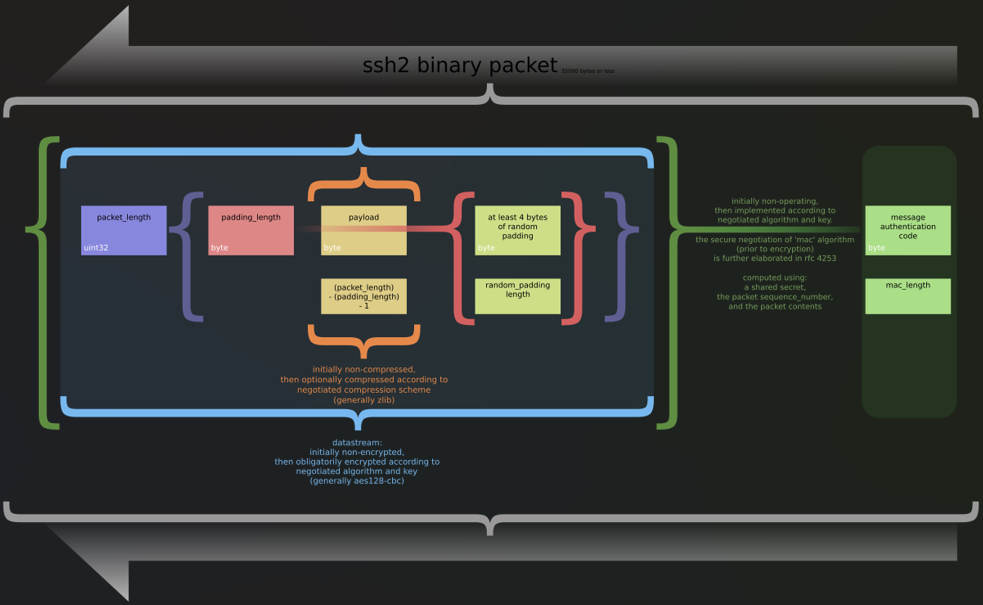
În cadrul unei soluției noastre, comunicarea securizată dintre componente este esențială. Două tehnologii populare care asigură această comunicare sunt protocolul SSH (Secure Shell) și serverele web ușoare construite cu ajutorul bibliotecii Flask din Python. Acest subcapitol oferă o scurtă prezentare a fiecăreia dintre aceste tehnologii, evidențiind modul în care contribuie la transferul sigur de date și comenzi între nodurile unui sistem distribuit.

### **Secure Shell**

Secure Shell (SSH) este un protocol criptografic de rețea utilizat pentru a opera servicii de rețea în mod securizat peste o rețea nesigură. Cele mai comune aplicații ale sale sunt conectarea la distanță și execuția de comenzi în linia de comandă, fiind adesea utilizat pentru administrarea sistemelor embedded sau Linux-based[24].

SSH a fost dezvoltat ca o alternativă securizată la protocoale mai vechi precum Telnet sau rlogin, care trimiteau datele (inclusiv parolele) în clar. În schimb, SSH utilizează criptare simetrică și asimetrică pentru a proteja comunicarea dintre client și server, asigurând astfel confidențialitatea și integritatea datelor transmise[24].

Arhitectura sa,dupa cum este prezentată în figura 1.3, este una de tip client–server: clientul inițiază conexiunea către server (care ascultă de regulă pe portul 22). După negocierea algoritmilor de criptare și autentificare, se stabilește un canal criptat prin care utilizatorul poate executa comenzi, transfera fișiere sau chiar redirecționa porturi.



***Figura 1.10:Arhitectura SSH[24]***

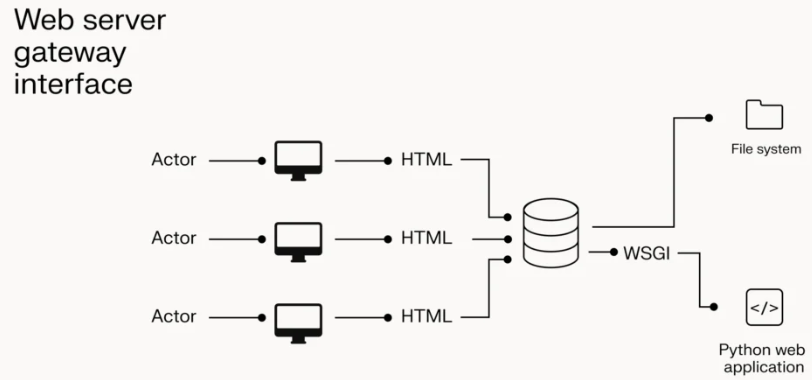
Mecanismul de autentificare poate folosi fie o parolă, fie un sistem mai sigur bazat pe criptografie cu chei publice. În scenarii automatizate sau embedded, autentificarea cu chei este preferată, deoarece elimină nevoia intervenției umane și reduce riscul interceptării parolelor.

În plus față de accesul interactiv, SSH suportă și transferul de fișiere securizat prin SFTP (SSH File Transfer Protocol) și montarea de sisteme de fișiere la distanță prin SSHFS (SSH File System), utile în contextul comunicării cu dispozitive edge sau senzori.

### **Server Flask**

Flask este un micro-framework web scris în Python, conceput pentru a fi simplu și extensibil. Acesta permite dezvoltatorilor să creeze aplicații web rapide și scalabile, fiind ideal pentru proiecte de dimensiuni mici și medii[25].

Arhitectura unei aplicații Flask este bazată pe modelul WSGI (Web Server Gateway Interface), care facilitează comunicarea între serverul web și aplicația Python. WSGI acționează ca o interfață standard între serverele web și aplicațiile Python,după cum e descris în figura 1.11, permițând o separare clară între logica aplicației și serverul care gestionează cererile HTTP[26].



***Figura 1.11: Arhitectura WSGI[26]***

Într-un scenariu tipic, un server web precum Nginx sau Apache primește cererile HTTP de la clienți și le transmite către un server WSGI, cum ar fi Gunicorn sau uWSGI. Acest server WSGI prelucrează cererile și le transmite aplicației Flask, care generează răspunsurile corespunzătoare. Răspunsurile sunt apoi trimise înapoi prin serverul WSGI către serverul web și, în final, către client.

În mod tradițional, Flask utilizează un sistem de decoratoare pentru a lega funcțiile de rutele HTTP definite. Aceste decoratoare definesc acțiunea pe care serverul trebuie să o execute atunci când primește o anumită cerere de la client. Pe lângă decoratoarele clasice pentru HTTP (eg: @app.route()), Flask permite extinderea comportamentului său prin alte tipuri de decoratoare, cum ar fi cele oferite de Flask-SocketIO pentru evenimente în timp real.

Pentru a permite o comunicare bidirecțională și în timp real între sistemul embedded și server, a fost utilizat modulul Flask-SocketIO, care extinde funcționalitatea Flask prin suportul pentru protocolul WebSocket.

Alegerea acestei tehnologii în locul clasicelor cereri HTTP a fost motivată de nevoia unui canal de comunicare continuu și reactiv, în care serverul și clientul pot schimba date în mod dinamic, fără să fie necesară inițierea repetitivă a conexiunii la fiecare interacțiune. Cererile HTTP funcționează într-un model request-response strict, ceea ce introduce o întârziere semnificativă atunci când este necesară transmiterea frecventă a datelor sau reacții imediate la evenimente externe.

În schimb, WebSocket permite menținerea unei conexiuni persistente, reducând latențele și utilizarea resurselor, fiind astfel mult mai eficient pentru sisteme embedded care necesită transmisii de comenzi, imagini sau alerte în timp real.

Această soluție a fost esențială pentru buna funcționare a comunicației între vehiculul autonom și interfața de control, permițând reacții rapide la schimbările din mediu și asigurând o coordonare fluidă între componentele sistemului.

## Elemente de inteligență artificială

### **Noțiuni generale**

Inteligența artificială (IA) se ocupă de crearea de algoritmi și modele care permit mașinilor să își „perceapă” mediul și să ia decizii pentru a-și atinge obiectivele[27].

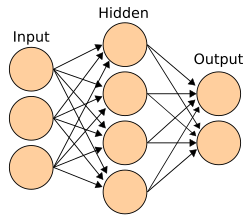
În acest context, procesarea vizuală este esențială: sistemele autonome trebuie să interpreteze imagini sau fluxuri video pentru a recunoaște obiecte și a naviga.

IA se bazează pe rețele neuronale – structuri inspirate de creierul uman – în care unitățile de bază sunt neuroni artificiali.

O rețea neurală este un ansamblu de noduri interconectate, similar cu rețeaua de neuroni biologici. Fiecare neuron artificial primește intrări ponderate de la alți neuroni și, dacă suma ponderată depășește un prag, „votează” pentru activare[27].Învățarea constă în ajustarea acestor ponderi pe baza datelor de antrenament, astfel încât rețeaua să recunoască modele complexe.

Neuronul artificial este astfel componenta de bază care permite rețelelor să învețe reprezentări utile ale datelor vizuale.

Rețelele neuronale clasice (feedforward, complet conectate),precum cea exemplificată în figura 1.12, întâmpină dificultăți majore în prelucrarea directă a imaginilor.



***Figura 1.12: Rețea neuronală[27]***

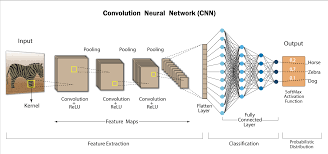
În primul rând, numărul de conexiuni și parametri crește exponențial cu dimensiunea imaginii,spre exemplu, un strat complet conectat ce primește o imagine color de 224×224 pixeli ar necesita peste 51 milioane de parametri. Acești parametri suplimentari solicită foarte mult resurse de calcul și memorie. În plus, straturile complet conectate tratează fiecare pixel independent, ignorând structura spațială a imaginii; astfel, ele nu pot exploata eficient faptul că pixeli vecini sunt corelați.

În consecință, aceste rețele sunt ineficiente pentru date vizuale, ducând la costuri computaționale mari și risc ridicat de overfitting.

### **Importanța rețelelor neuronale convoluționale în detectarea obiectelor**

Rețelele neuronale convoluționale (CNN) au fost concepute tocmai pentru a depăși aceste limitări.

O CNN tipică include trei tipuri principale de straturi: convoluție, pooling și fully connected, după cum sunt evidențiate în figura 1.13.



***Figura 1.13: Structura unui CNN***

Stratul de convoluție aplică filtre (kerneluri) locale pe imagine, partajând aceleași greutăți pe întreaga suprafață (partajarea greutăților). Acest mecanism permite detectarea acelorași tipare locale oriunde ar apărea în imagine, oferind invarianță la translații și reducând drastic numărul de parametri comparativ cu un strat complet conectat. De exemplu, weight-sharing-ul în convoluție asigură că același filtru învață un anumit tipar (contur, textură) în toate pozițiile imaginii, diminuând considerabil complexitatea modelului. Stratul de pooling (de obicei max pooling) realizează subeșantionarea matricei de caracteristici, reducând dimensiunile spațiale și numărul de parametri de procesat; astfel se obține robustețe la mici deplasări și zgomot visual.

În final, unul sau mai multe straturi complet conectate iau caracteristicile extrase și realizează sarcina de clasificare. În ansamblu, arhitectura CNN – conv, pooling și straturi complet conectate – permite extragerea automată a trăsăturilor relevante din imagin**i** și clasificarea lor, fără a necesita conectivitate globală între toți pixelii.

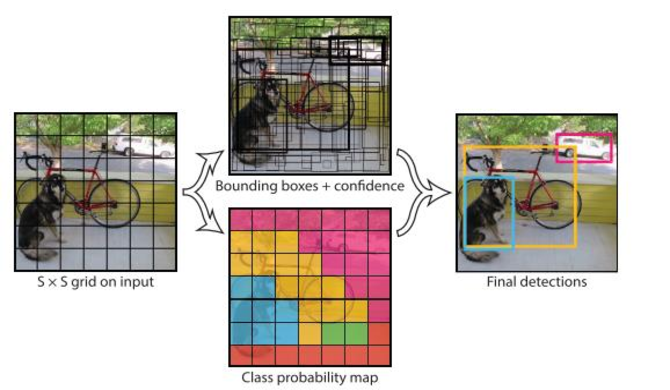
În acest fel, CNN-urile pot procesa imagini de dimensiuni mari mult mai eficient decât rețelele neuronale clasice, economisind resurse și crescând performanța în sarcini de viziune computațională.

### **Modelul YOLO**

O aplicație practică importantă a CNN-urilor în navigație autonomă este detecția de obiecte în timp real, unde este utilizat pe scară largă algoritmul YOLO (You Only Look Once). YOLO, propus în lucrarea "You Only Look Once: Unified, Real-Time Object Detection"[29],care realizează detecția obiectelor într-o singură trecere a rețelei, spre deosebire de metodele convenționale bazate pe ferestre glisante sau propuneri de regiuni.

Procesul de detectare al unui obiect dintr-o imagine, evidențiat în figura 1.14, funcționează astfel:

* Imaginea de intrare este împărțită într-un grid de celule, iar pentru fiecare celulă rețeaua prezice simultan mai multe casete delimitatoare (bounding boxes) și scoruri de încredere pentru posibilele clase de obiecte.
* Fiecare casetă prezice cinci valori (coordonatele centrului relativ la celulă, lățimea, înălțimea și scorul de încredere).
* Toate predicțiile sunt apoi filtrate (prin Non-Maximum Suppression) pentru a obține box-urile finale.



***Figura 1.14: Modul de funcționare al modelului YOLO[29]***

Metoda YOLO permite astfel detectarea rapidă și eficientă a obiectelor în imagine, conform filosofiei “o singură privire” (un singur pas în rețea).

Iterația YOLO folosită în cazul nostru, versiunea YOLOv8, reprezintă un punct de maturitate în dezvoltarea arhitecturilor pentru detecția obiectelor în timp real.

Lansat de Ultralytics în 2023, YOLOv8 aduce o arhitectură eficientă și modulară, cu un backbone modernizat (C2f) și un head complet decuplat, anchor-free, ceea ce simplifică procesul de antrenare și crește acuratețea localizării obiectelor[30].

De asemenea, YOLOv8 oferă suport extins pentru sarcini multiple, precum segmentare semantică și estimare de poziție, ceea ce îl transformă într-o platformă unificată pentru viziune computațională în timp real.

Cu o precizie ridicată și performanțe excelente pe o gamă variată de dispozitive, YOLOv8 se poziționează drept una dintre cele mai flexibile și robuste soluții de detecție vizuală disponibile în prezent.

## Tehnologii de control embedded

### **Comunicarea serială**

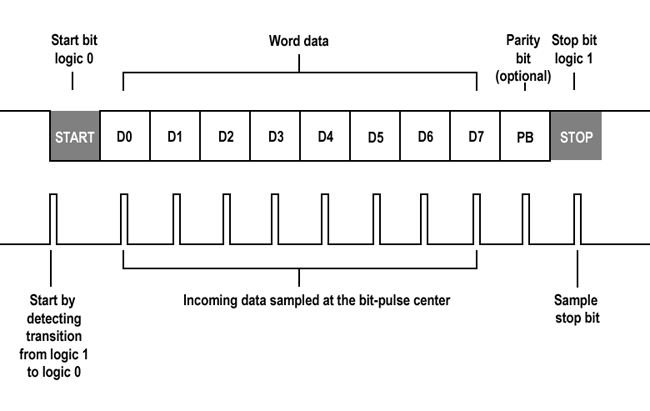
Comunicarea serială asincronă reprezintă un mod eficient de transfer al datelor între dispozitive, în care biții sunt transmise secvențial, bit cu bit, pe o linie fizică dedicate[36].

În universul embedded, există diverse protocoale seriale dedicate, precum SPI, I²C, RS‑232 sau UART, însă microcontroller-ele folosite în această lucrare utilizează protocolul UART (Universal Asynchronous Receiver–Transmitter), datorită simplității, eficienței și absenței necesității unui semnal de ceas comun[36].

În esență, UART este un dispozitiv hardware integrat în microcontrolere sau implementat ca circuit auxiliar. Acesta convertește datele în format paralel în șiruri seriale de biți și inversează procesul la destinație.

Comunicarea se realizează pe două linii,după sincronizarea parametrilor inițiali, TX și RX, fără a utiliza vreun semnal de ceas extern.

În figura 1.15 este ilustrată arhitectura generală a unui canal UART, care include componentele transmitter și receiver, fiecare având un registru de tip shift, un registru temporar ,logica de control și un generator de baud‑rate:



***Figura 1.15: Principiu de funcționare UART[35]***

Protocolul UART structurează fiecare cadru de date astfel: prin transmiterea unui bit de start (logic 0), urmat de 5 până la 9 biți de date, opțional un bit de paritate, și unul sau doi biți de stop (logic 1).

Bitul de start identifică începutul unui cadru, în timp ce bitul de stop permite resincornizarea receptorului în absența unui semnal de ceas.

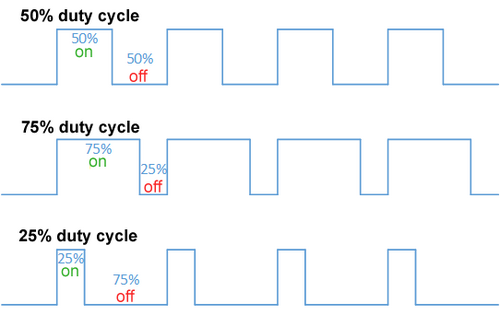
Acest protocol este ideal pentru comunicații de tip full‑duplex cu eficiență moderată, necesitând doar două conductoare fizice și fiind comun în aplicații embedded, pentru interconectarea microcontrolerelor cu senzori, module de comunicații sau alte dispozitive externe

### **Modularea lățimii impulsului (Pulse Width Modulation)**

În cadrul unui vehicul autonom, tehnica modulării lățimii impulsurilor (PWM) este esențială pentru controlul eficient al vitezei motoarelor și al cuplului, contribuind la stabilitate și manevrabilitate.

Această tehnica constă în aplicarea unui semnal digital cu o frecvență constantă, dar cu o durată variabilă a impulsurilor (duty cycle), ceea ce modifică tensiunea medie aplicată motorului și, implicit, viteza acestuia[37].

Într-un exemplu ce poate fi observat și în figura 1.16, pentru un duty cycle de 50% înseamnă că semnalul este activ (HIGH) pentru jumătate din perioada totală, rezultând o tensiune medie aplicată motorului de aproximativ 50% din tensiunea de alimentare. Prin creșterea duty cycle-ului, tensiunea medie și viteza motorului cresc, iar prin scăderea acestuia, viteza motorului scade.



***Figura 2.2:Variația impulsurilor [37]***

Prin urmare, PWM reprezintă o metodă fiabilă, economică și controlabilă digital pentru gestionarea puterii aplicate motoarelor unui vehicul autonom, asigurând ajustări fine ale vitezei și performanțe optime din punct de vedere energetic.

### **Controlul regulat al mișcării**

Regulatorul PID (Proporțional–Integral–Derivativ) este un mecanism de control în buclă închisă, folosit pe scară largă în sisteme care necesită reglaje continue și automate. Acesta compară constant valoarea dorită (setpoint) cu valoarea măsurată (process variable), iar diferența dintre acestea — numită eroare — este utilizată pentru generarea semnalului de control.

Funcționarea PID se bazează pe combinația a trei acțiuni:

* **Componenta proporțională (P)** produce o corecție direct proporțională cu eroarea curentă. Astfel, cu cât vehiculul se abate mai mult de la traiectoria dorită, cu atât controlul aplicat este mai intens.
* **Componenta integrală (I)** acumulează erorile anterioare pentru a corecta deviațiile persistente care nu sunt eliminate de componenta P, asigurând compensarea reziduală pe termen lung .
* **Componenta derivativă (D)** anticipează tendințele erorii prin estimarea ratei de schimbare a acesteia, contribuind la atenuarea oscilațiilor și sporind stabilitatea sistemului .

Ecuatia generală a unui regulator PID este:

( 1,9)

unde u(t) este semnalul de control generat pentru sistem, iar e(t) este eroarea dintre setpoint și variabila măsurată.

Astfel, regulatorul PID oferă un control fin al mișcării, cu ajustări dinamice ce echilibrează reacția rapidă, precizia pe termen lung și stabilitatea în fața perturbărilor. Această abordare este optimă pentru sisteme autonome care operează în medii variabile, unde erorile de traiectorie pot apărea din fracturi de suprafață, variații de viteză sau schimbări bruște de direcție.

Capitolul de față a prezentat fundamentele teoretice esențiale pentru înțelegerea și proiectarea unui sistem autonom de navigație bazat pe tehnologii embedded și inteligență artificială. Au fost analizate tehnicile de procesare a imaginilor necesare interpretării vizuale a mediului, arhitectura GStreamer utilizată pentru manipularea fluxurilor video, precum și tehnologiile web care facilitează comunicarea între componentele distribuite ale sistemului.

În continuare, au fost introduse conceptele de bază din domeniul inteligenței artificiale, cu accent pe rețelele neuronale convoluționale și modelul YOLOv8, care constituie motorul de detecție vizuală al platformei autonome. De asemenea, au fost expuse principiile tehnologiilor de control embedded, incluzând comunicarea serială UART, generarea semnalelor PWM și utilizarea unui regulator PID pentru menținerea traiectoriei.

Prin această sinteză teoretică, se fundamentează componentele care vor fi dezvoltate și integrate practic în capitolele următoare, asigurând o înțelegere clară a mecanismelor care guvernează funcționarea unui sistem embedded inteligent, capabil de percepție și acțiune în timp real.

# Capitolul 2: Arhitectura generală a sistemului

Acest capitol oferă o prezentare detaliată a arhitecturii generale a sistemului embedded dezvoltat pentru navigație autonomă, punând în evidență modul în care componentele hardware și software colaborează pentru a permite funcționarea în timp real a vehiculului.

Sunt descrise principalele blocuri funcționale, fluxurile de date și interacțiunile dintre modulele sistemului, oferind o imagine completă a modului în care teoria este aplicată în practică.

În primul subcapitol sunt analizate componentele hardware esențiale, precum Raspberry Pi 5, acceleratorul Hailo-8L, microcontrolerul Arduino și modulul de control al motoarelor L298N, precum și modul în care acestea sunt conectate și alimentate pentru a forma o platformă embedded funcțională.

Al doilea subcapitol detaliază componentele software implicate în sistem, de la sistemul de operare și aplicațiile de control la distanță (precum RealVNC), până la mediile de dezvoltare utilizate (VSCode și PyCharm), limbajele de programare alese (Python și C++) și bibliotecile folosite (OpenCV, Flask, Socket.IO etc.), subliniind distribuția logică a aplicației între dispozitive.

În final este prezentată arhitectura sistemului din punctul de vedere al fluxului de date. Se analizează pipeline-ul de prelucrare video implementat cu GStreamer, modul în care modelul YOLOv8 este integrat și accelerat hardware cu Hailo-8L, structura serverului Flask și a dashboard-ului grafic, precum și comunicarea cu Arduino pentru controlul motoarelor prin regulatorul PID. Este evidențiată interacțiunea coerentă dintre aceste module și sincronizarea necesară pentru a asigura un comportament autonom robust.

Prin această organizare, capitolul fundamentează arhitectura practică a sistemului, făcând legătura directă între conceptele teoretice prezentate anterior și implementarea lor într-un prototip funcțional de vehicul autonom.

## Componente hardware

### **Raspberry Pi**

Raspberry Pi 5 este placa de bază de tip „single-board computer” utilizată ca unitate centrală de calcul în sistem. Aceasta dispune de un procesor 64-biţ quad-core ARM Cortex-A76 la 2,4 GHz, oferind un salt de 2–3× în performanţa CPU faţă de Raspberry Pi 4. Conexiunile USB (două USB 3.0 și două USB 2.0), Ethernet Gigabit şi interfaţa PCI Express 2.0 îi permit transfer rapid de date și conectivitate extinsă[31]. Memoria de lucru LPDDR4X 8GB şi slotul pentru card microSD asigură capacitate de stocare şi rulare a sistemului de operare Linux, precum şi a aplicaţiilor de calcul intensiv.

Această alegere este justificată de necesitatea unui calculator puternic care să ruleze sistemul de operare şi să coordoneze procesarea imaginii cu YOLOv8 în timp real.

Placa Raspberry Pi 5 oferă suficientă performanţă de calcul pentru a gestiona interfaţele hardware (USB, CSI, I2C etc.), conexiunea cu acceleratorul neural şi transferul de date către aplicatia de detectie de obiecte.

Suportul oficial pentru accesorii (precum camera CSI) şi comunitatea largă de utilizatori fac din Raspberry Pi 5 o platformă potrivită pentru prototipuri de vehicule autonome. Performanţele CPU şi GPU ridicate, împreună cu interfaţa PCIe integrată, permit integrarea facilă a acceleratorului neural Hailo şi exploatarea capacităţii de procesare paralelă necesare aplicaţiilor de viziune artificială avansată.

### **Camera video RaspberryPi 2.1**

Camera Raspberry Pi v2.1 este o placă de cameră dedicată, proiectată special pentru platformele Raspberry Pi. Ea folosește senzorul Sony IMX219, de 8 megapixeli[32].

Camera suportă captură de imagini statice de înaltă rezoluție și video până la 1080p cu 30 cadre pe secundă (și moduri mai rapide la rezoluții reduse). Conexiunea la Raspberry Pi se realizează prin interfața CSI (MIPI CSI-2), oferind bandă de date mare pentru transmiterea fluxului video brut direct în memoria dispozitivului.

Fiind un modul oficial, beneficiază de suport software în ecosistemul Raspberry Pi, eliminând incompatibilități și simplificând procesul de calibrare și reglare.

De asemenea, forma compactă și consumul redus (încărcabilă de la sursa comună) corespund cerințelor unui vehicul autonom de dimensiuni mici spre medii. Astfel, Camera v2.1 asigură captarea imaginii de la care se pornește analiza vizuală în sistem, conectându-se direct la placa Raspberry Pi 5 prin magistrala dedicată.

În cadrul sistemului, camera funcționează ca sursă de intrare video pentru pipeline-ul GStreamer. Inițial, camera captează la rezoluția maximă disponibilă de 1080p însă pentru procesul de inferență realizat de modulul Hailo, imaginea este redimensionată la 640×640 pixeli, aceasta fiind rezoluția acceptată de acceleratorul Hailo. Ulterior, pipeline-ul care transmite fluxul video utilizează tot rezoluția de 640×640 pixeli, însă imaginea este doar scalată, evitând astfel un zoom excesiv care ar putea afecta calitatea vizuală a stream-ului. Această abordare asigură un echilibru între calitatea imaginii și cerințele de procesare ale sistemului, permițând o inferență eficientă fără a compromite experiența vizuală a utilizatorului.

### **Accelearatorul neural Hailo-8L**

Hailo-8L este un accelerator hardware dedicat inferenţei reţelelor neurale, conceput pentru aplicaţii edge cu cerințe moderate. Acesta oferă până la 13 TOPS (operaţiuni trillioane pe secundă) la un consum redus (≈1,5 W)[33]. Datorită designului său optimizat, Hailo-8L nu necesită memorie externă şi oferă latenţă foarte mică, fiind capabil să proceseze fluxuri multiple video simultan şi mai multe modele de reţea în paralel.

Suportă framework-uri standard (TensorFlow, PyTorch, ONNX etc.) şi este compatibil din punct de vedere software cu suita Hailo, permițând portarea rapidă a modelelor de detectare obiecte (de exemplu YOLO) pentru inferenţă.

Acceleratorul Hailo-8L a fost ales pentru că YOLOv8 este un model complex de detecție de obiecte care necesită resurse de calcul semnificative. Procesarea în timp real a fluxului video capturat de cameră ar încărca foarte mult processorul Raspberry Pi-ului, astfel că Hailo-8L oferă prin intermediul interfeței PCIe un suport hardware specializat pentru inferenţă rapidă.

Prin utilizarea Hailo-8L, sistemul poate atinge viteze de procesare de nivel server în condiții de consum scăzut, satisfăcând cerinţele de latenţă şi eficienţă ale aplicaţiei autonome. Caracterul său entry-level și prețul competitiv îl fac adecvat pentru proiecte integrate de robotică și mobilitate autonomă unde este nevoie de accelerație AI eficientă

### **Arduino Uno R3**

Arduino Uno R3 este o placă de dezvoltare microcontrolată bazată pe microcontrolerul ATmega328P. Ea oferă 14 pini digitali de intrare/ieșire (dintre care 6 pot genera semnale PWM), 6 intrări analogice şi un oscilator de 16 MHz[34]. Dispune de interfaţă USB, conector de alimentare, buton de reset şi header ICSP, fiind proiectată să fie robustă şi uşor de utilizat. Arduino Uno este bine documentat și foarte utilizat în proiecte didactice şi industriale, ceea ce permite dezvoltarea rapidă a codului de control şi accesul la o multitudine de biblioteci software.

În arhitectura sistemului, Arduino Uno R3 este utilizat pentru interfaţele de control la nivel scăzut şi pentru gestionarea motoarelor. De exemplu, Arduino poate genera semnale PWM şi comenzi logice către driverul de motoare L298N, precum și gestionarea comunicării cu componenta de Raspberry Pi, prin comunicarea serială.

Alegerea sa se datorează costului redus, consumului mic și portabilității – microcontrolerul ATmega328P fiind ușor de înlocuit în caz de avarie.

În plus, documentația bogată și comunitatea Arduino ușurează integrarea în sistem și prototiparea unor funcții de control esențiale, neimplicând executarea directă a sarcinilor mai complexe, acestea fiind realizate de către Raspberry Pi.

### **Modului driver L298N pentru controlul motoarelor**

Driverul L298N este un modul dual H-Bridge proiectat pentru a controla motoare DC și motoare pas-cu-pas. El conține două circuite integrate L298N care permit controlul independent al a două canale de motor, fiecare canal suportând un curent de până la 2 A (maxim) la tensiuni de până la 46 V[42].

Modulul include un regulator de tensiune de 5 V pentru logica internă și câteva LED-uri de stare.

Semnalele de intrare (IN1–IN4, ENA/ENB) controlează direcția și turația motoarelor prin semnale logice și PWM, iar ieșirile (OUT1–OUT4) se conectează la bornele motoarelor.

## Componente software

Arhitectura software a sistemului embedded a fost construită pentru a susține un flux de dezvoltare eficient, interacțiune de la distanță și procesare distribuită între mai multe platforme.

O componentă importantă a acestei arhitecturi este capacitatea de a accesa interfața grafică a Raspberry Pi 5 fără a necesita un monitor extern. În acest scop, a fost utilizată aplicația RealVNC Viewer, care permite conectarea remote la interfața desktop a Raspberry Pi-ului prin protocol VNC (Virtual Network Computing). Conectarea s-a realizat prin IP-ul local al dispozitivului, accesat prin hostname-ul raspberrypi.local, fără a necesita configurări suplimentare de rețea sau cabluri video. Această soluție oferă control complet asupra interfeței grafice, fiind utilă atât în faza de dezvoltare, cât și în cea de testare.

În această configurație, microcontroller-ul Arduino este conectat fizic la Raspberry Pi prin portul USB, iar codul este scris, încărcat și monitorizat direct din Arduino IDE instalat pe Pi. Acest flux de lucru este practic, deoarece permite editarea și programarea controlerului direct de pe platforma embedded, fără a implica un PC de dezvoltare separat pentru programarea microcontrolerului.

Figura 2.1 exemplifică utilizarea RealVNC pentru accesarea grafică a sistemului de operare instalat pe Raspberry Pi, rulând aplicația Arduino IDE direct în mediul desktop:



***Figura 2.1: Rulare Arduino IDE remote,folosind RealVNC***

Pentru dezvoltarea codului Python care rulează pe Raspberry Pi, a fost utilizat editorul Visual Studio Code (VSCode), instalat pe PC-ul de dezvoltare.

VSCode este o platformă puternică și extensibilă, cu suport pentru limbaje multiple, debugging integrat și extensii utile. Un avantaj major îl constituie posibilitatea conectării la dispozitive remote prin protocol SSH, folosind extensia „Remote – SSH”. Astfel, codul sursă a fost editat și executat direct pe Raspberry Pi fără a fi necesar transferul manual de fișiere, iar logurile și ieșirile sistemului puteau fi monitorizate direct din terminalul integrat al IDE-ului.

De asemenea, prin această metodă de conectare, a fost posibilă deschiderea și monitorizarea ușoară a porturilor esențiale pentru comunicație în sistem, cum ar fi portul 4956 pentru streamul video și portul 5000 utilizat de serverul Flask pentru controlul și interacțiunea cu interfața de tip dashboard.

Pe lângă VSCode, un alt mediu de dezvoltare utilizat a fost PyCharm, instalat tot pe PC-ul de dezvoltare. Acesta a fost folosit pentru dezvoltarea interfeței grafice de control (dashboard) care comunică cu serverul Flask rulat pe Raspberry Pi. În plus, PyCharm a fost utilizat pentru antrenarea modelului YOLOv8, cu ajutorul librăriei ultralytics, în medii locale sau cloud.

După antrenare, modelul a fost convertit în formatul intermediar ONNX (Open Neural Network Exchange), un standard deschis compatibil cu multiple framework-uri. Acest format a fost apoi introdus în Hailo Model Compiler, care a generat fișierul final cu extensia .hef necesar pentru rularea modelului pe acceleratorul Hailo-8L.

Alegerea PyCharm s-a datorat nevoii de a menține separația între sarcinile de dezvoltare: evitarea deschiderii a două instanțe VSCode care pot genera confuzie și folosirea unui IDE mai robust, cu suport extins pentru debugging, profiling și gestionarea mediilor virtuale. PyCharm consumă mai multe resurse față de VSCode, dar oferă o integrare mai avansată cu librării complexe și o ergonomie mai potrivită pentru proiecte de dimensiune medie spre mare, ceea ce l-a făcut potrivit pentru antrenare și dezvoltarea aplicației de interfață.

Limbajul de programare principal utilizat dezvoltare a fost Python, datorită compatibilității excelente cu API-ul Hailo și framework-ul GStreamer. Suportul oferit de biblioteca oficială hailo\_platform este nativ în Python, facilitând astfel integrarea modelului YOLOv8 în pipeline-ul video. Totodată, limbajul Python oferă acces la o multitudine de biblioteci utile în contextul sistemelor embedded, precum OpenCV pentru prelucrarea imaginilor, NumPy pentru manipulări vectoriale și Flask pentru comunicația intermodule.

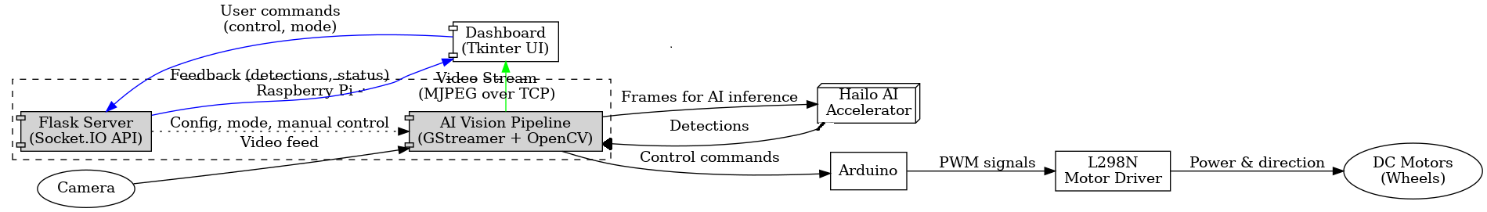
Controlul motoarelor și aplicarea regulatorului PID sunt implementate pe microcontrolerul Arduino Uno R3, pentru care a fost folosit mediul de dezvoltare Arduino IDE. Acest IDE oficial oferă o interfață simplă și robustă pentru scrierea, compilarea și încărcarea codului C/C++ pe plăci Arduino, permițând totodată și monitorizarea comunicației seriale. Codul pentru regulatorul PID este structurat modular, având în vedere execuția în timp real și reacția rapidă la comenzile venite de la Raspberry Pi.

Împreună, aceste componente software oferă o infrastructură coerentă și scalabilă pentru sistemul de navigație autonomă propus. Controlul la distanță prin RealVNC, dezvoltarea asistată prin SSH cu VSCode, limbajul Python optim pentru API-ul Hailo și GStreamer, precum și programarea low-level cu Arduino IDE contribuie fiecare la integrarea eficientă între componenta de percepție vizuală și cea de control al mișcării.

## Arhitectura generală a sistemului

Navigația autonomă a vehiculului se bazează pe o arhitectură integrată care îmbină componente hardware și software pentru a asigura captarea, procesarea și interpretarea datelor vizuale, precum și controlul precis al mișcării.

Figura 2.4 ilustrează această arhitectură modulară, evidențiind fluxurile de date și interacțiunile dintre componentele principale: camera video, pipeline-ul de viziune artificială, acceleratorul Hailo-8L, serverul Flask cu Socket.IO, dashboard-ul utilizatorului și modulul de control al motoarelor.



***Figura 2.4: Arhitectura generală a sistemului***

### **Camera și pipeline-ul de viziune artificială**

Aplicația GStreamer principală rulează pe Raspberry Pi și reprezintă inima sistemului nostru autonom.

Raspberry Pi gestionează prelucrarea imaginilor și logica decizională, în timp ce Arduino execută comenzile primite, folosind un controler PID pentru ajustarea vitezelor motoarelor.

La inițializare, aplicația pornește un pipeline GStreamer care captează și preprocesează fluxul video, redimensionând imaginea și pregătind-o pentru detecția obiectelor și segmentarea liniei. Totodată, este lansat și serverul Flask, responsabil cu recepționarea comenzilor și parametrilor în timp real.

Pentru fiecare cadru capturat, aplicația verifică dacă sistemul se află în mod manual.

În acest caz, comenzile de control primite de la interfața grafică sunt transmise direct către Arduino în formatul "H,viteza\_stânga,viteza\_dreapta".

În mod autonom, aplicația aplică un lanț de preprocesare imagistică, înainte de a detecta linia de ghidare. Acest lanț include rotirea imaginii cu 180°, aplicarea unui filtru Gaussian și a unui kernel de ascuțire. Detecția liniei negre este realizată prin thresholding adaptiv, urmată de operații morfologice de erodare și dilatare, concepte prezentate pe larg în capitolul 1. Coordonata centrului liniei este extrasă prin calculul centroidului conturului principal, iar eroarea de poziționare se obține ca diferență între această coordonată și centrul imaginii.

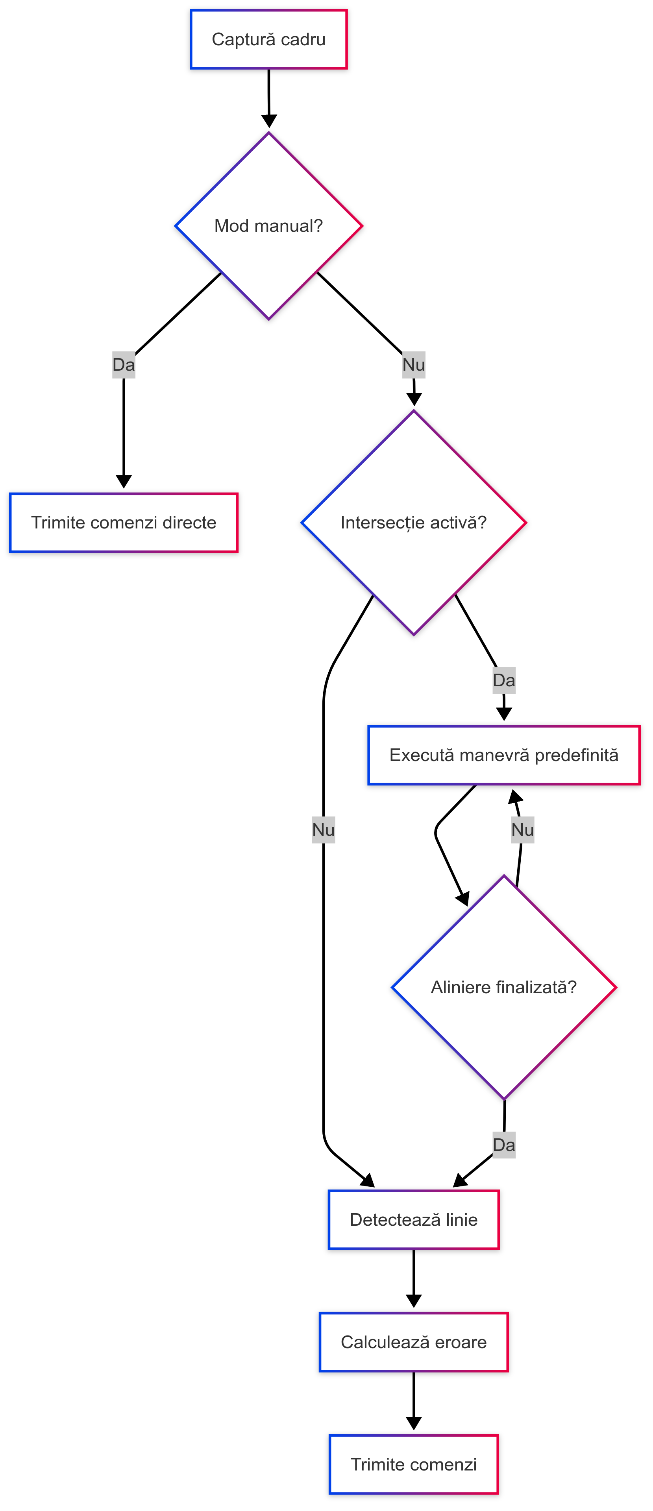
Dacă vehiculul nu se află într-o intersecție activă, Raspberry Pi transmite periodic către Arduino comenzi de tip "E,eroare\_calculată". Arduino ajustează viteza motoarelor în funcție de această eroare, folosind un algoritm PID.

Detecția intersecțiilor se realizează prin aproximarea poligonală a conturului liniei,folosind o implementare practica a algoritmului Douglas-Peucker,algoritm prezentat pe larg in capitolul 1, iar semnele de circulație sunt identificate în zona inferioară a cadrului folosind acceleratorul Hailo și modelul YOLOv8. În cazul în care se detectează o intersecție cu semn activ, vehiculul oprește pentru un număr de cadre (ex. stop\_frames), apoi execută o manevră predefinită (de exemplu, viraj dreapta) pentru un interval fixat (direction\_frames). La final, un mecanism de realiniere asigură revenirea corectă pe traseu.

Arduino operează în trei moduri: control PID, execuție de comenzi directe sau modul de siguranță (activat automat dacă nu primește comenzi într-un interval de 200 ms).

Comenzile sunt validate și transmise în timp real, cu constrângeri aplicate pe vitezele PWM (60–255), pentru a menține o operare sigură și stabilă.

Întregul flux decizional este ilustrat în Figura 2.3 și se repetă ciclic: după fiecare manevră sau ajustare de traiectorie, sistemul revine automat la starea de detecție și urmărire a liniei.



***Figura 2.3: Diagrama fluxului decizional al aplicației autonome***

Prin integrarea camerei Raspberry Pi cu un pipeline GStreamer personalizat, sistemul reușește să capteze, prelucreze și analizeze în timp real fluxul video necesar navigației autonome. GStreamer nu doar facilitează manipularea eficientă a cadrelor video, ci și permite conectarea directă cu acceleratorul Hailo pentru inferență hardware, reducând latența și maximizând performanța sistemului.

Combinația dintre procesarea imaginii, analiza decizională bazată pe contextul rutier și capacitatea de comunicare rapidă cu microcontrolerul Arduino asigură un comportament adaptiv și robust al vehiculului, atât în scenarii liniare, cât și în situații complexe precum intersecțiile cu semne de circulație.

Această infrastructură de prelucrare vizuală constituie astfel coloana vertebrală a sistemului, reprezentând punctul de intersecție între percepția vizuală și acțiunea fizică asupra mediului, în cadrul unui flux decizional complet automatizat și controlabil în timp real.

### **Serverul Flask cu Socket.IO**

Serverul Flask reprezintă componenta intermediară esențială care coordonează interacțiunea în timp real dintre interfața grafică (dashboard) și pipeline-ul de procesare video. Acesta este responsabil de primirea comenzilor de la utilizator, propagarea modificărilor către pipeline și transmiterea datelor de stare și detecție înapoi către interfață.

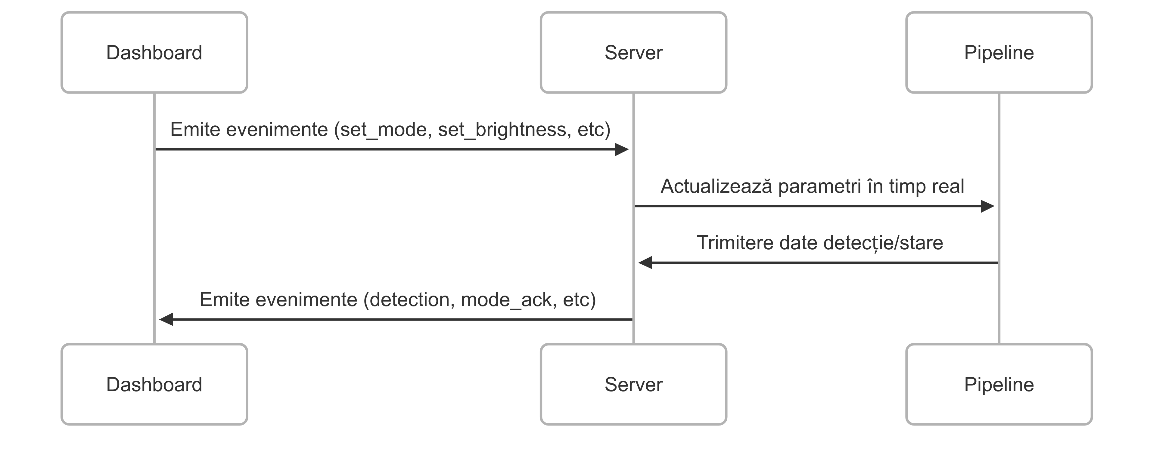
Pentru a permite o comunicare bidirecțională, rapidă și asincronă între aceste module, serverul este construit folosind microframework-ul Flask în combinație cu extensia Flask-SocketIO, care oferă suport complet pentru protocolul WebSocket.

După lansarea aplicației, dashboard-ul emite periodic sau la cerere evenimente precum set\_mode, set\_brightness, set\_pid sau set\_speed, care sunt recepționate de serverul Flask. Acesta actualizează instantaneu pipeline-ul activ, trimițând noii parametri printr-un canal de control.

De asemenea, atunci când pipeline-ul detectează un semn de circulație, o intersecție sau o abatere de la traseu, acesta trimite înapoi date de detecție, eroare și stare operațională. Acestea sunt convertite de server în evenimente de tip detection, status, mode\_ack, etc., ce sunt apoi transmise către dashboard pentru afișare sau reacție vizuală.

Întregul mecanism este asincron și permite modificarea în timp real a funcționării sistemului, fără a fi necesară oprirea sau repornirea pipeline-ului.

Figura 2.4 evidențiază schimbul bidirecțional de evenimente între componentele software. Comenzile utilizatorului sunt propagate prin server către pipeline, iar rezultatele procesării sunt retransmise către dashboard.



***Figura 2.4: Fluxul de comunicare în timp real între Dashboard, Server și Pipeline***

Această arhitectură permite utilizatorului să modifice dinamica sistemului în mod reactiv, să primească feedback vizual instantaneu și să mențină un control fin asupra parametrilor de funcționare.

Serverul Flask devine astfel un punct central al ecosistemului software, asigurând coerență între componente și menținând sincronizarea între percepție, decizie și interfață.

### **Dashboard-ul (clientul utilizator)**

Dashboard-ul este o aplicație desktop dezvoltată în Python, utilizând biblioteca Tkinter pentru interfața grafică. Tkinter oferă un set de widget-uri pentru construirea interfețelor grafice în Python și este inclus în distribuția standard a limbajului.

Dashboard-ul permite utilizatorului să:

* Vizualizeze fluxul video în timp real, cu detecțiile suprapuse.
* Trimită comenzi de control către vehicul, cum ar fi pornirea sau oprirea motoarelor, schimbarea vitezei sau comutarea între modurile de navigație.
* Ajusteze parametrii de detecție, cum ar fi pragurile de încredere sau modurile de scanare.

Comunicarea între dashboard și serverul Flask prezentată anterior asigură o conexiune WebSocket permanentă pentru transmiterea rapidă a datelor.

### **Modulul de control al vehicului**

Comenzile de control primite de la dashboard sunt transmise către un microcontroler Arduino Uno R3 prin interfață serială USB. Arduino utilizează un algoritm PID (Proporțional-Integral-Derivat) pentru a controla viteza și direcția motoarelor, asigurând o mișcare precisă și stabilă a vehiculului.

Biblioteca PID pentru Arduino permite implementarea acestui algoritm, oferind funcționalități pentru ajustarea parametrilor și calculul semnalelor de control.

Semnalele generate de Arduino sunt transmise către un driver de motoare L298N, care amplifică și adaptează curentul pentru a alimenta motoarele DC ale vehiculului.

### **Sinergie și sincronizare generală**

Funcționarea sistemului autonom presupune coordonarea în timp real a mai multor subsisteme eterogene, fiecare cu specificații, frecvențe și latențe distincte. Sinergia acestor componente este asigurată printr-o arhitectură distribuită, unde sincronizarea se realizează prin canale de comunicație asincrone și prin constrângeri temporale impuse la nivel de aplicație.

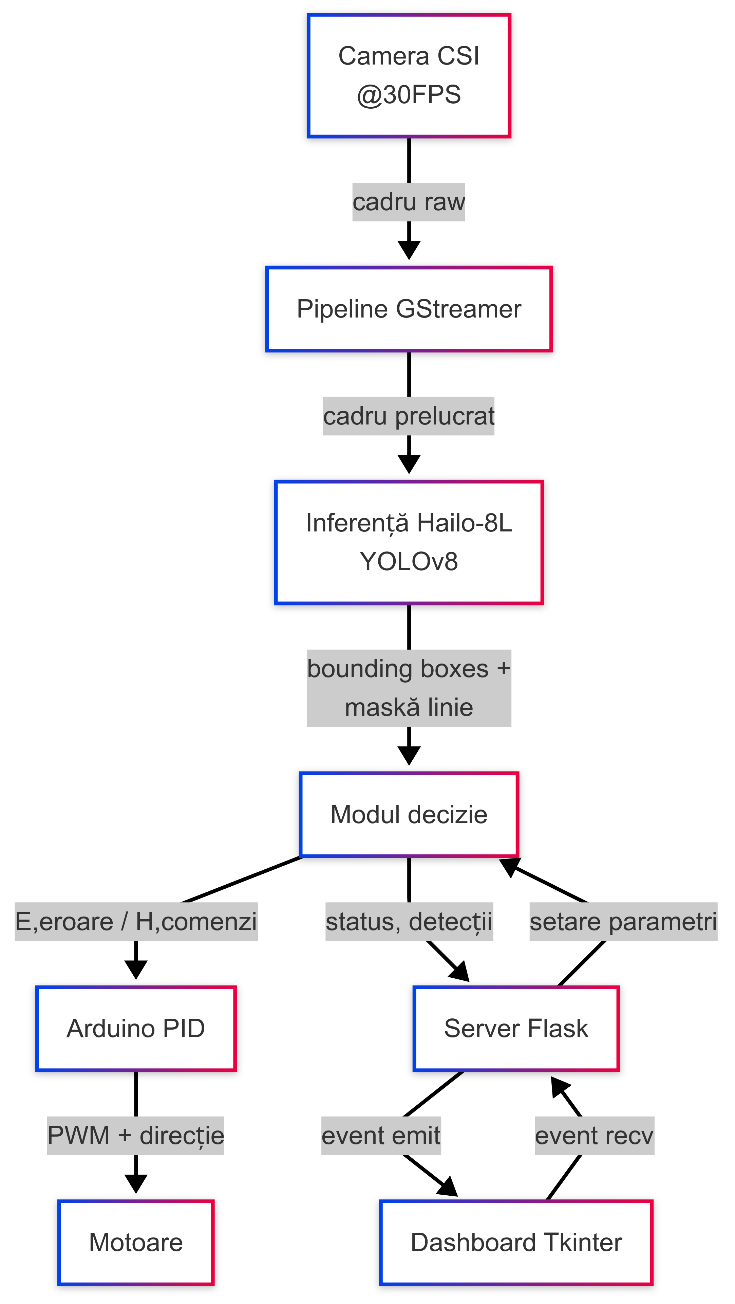
Captura și procesarea imaginii sunt gestionate de un pipeline GStreamer care funcționează în regim continuu, generând cadre la o frecvență fixată (de exemplu 30 FPS). Fiecare cadru este supus unui lanț de preprocesare, inferență neuronală (pe acceleratorul Hailo-8L), și analiză geometrică (pentru urmărirea liniei sau detecția de intersecții). Pipeline-ul acționează ca sursă principală de evenimente pentru luarea deciziilor.

Serverul Flask acționează ca un broker de comunicații între nivelul de control și interfața utilizatorului. Utilizând protocolul WebSocket prin extensia Flask-SocketIO, acesta primește și transmite mesaje în regim full-duplex, fără blocaj, permițând reglarea parametrilor PID, a vitezei motoarelor și schimbarea modului de funcționare (manual/autonom) în timpul execuției.

Controlerul Arduino funcționează într-o buclă de control de tip event-driven, în care mesajele de tip "E,eroare" sau "H,comenzi\_directe" sunt procesate imediat ce sunt recepționate. Timpul de reacție este controlat printr-un mecanism de timeout (200 ms), asigurând robustețe la pierderi temporare de pachete sau întârzieri.

Sincronizarea globală este realizată implicit prin frecvența de procesare a cadrelor și prin coerența mesajelor transmise pe canalele WebSocket și UART. Timpul total de la captură până la aplicarea comenzii motoarelor este constrâns sub 100 ms, ceea ce permite sistemului să reacționeze la variații de traseu sau la semne rutiere în regim near real-time.

Fluxul complet de date, sincronizare și control între subsisteme este ilustrat în Figura 2.6, unde sunt evidențiate principalele canale de comunicație și ordinea procesării de la percepție la acțiune.



***Figura 2.6: Sincronizarea fluxului de procesare între subsistemele sistemului autonom***

Capitolul de față a prezentat în mod detaliat structura hardware și software a sistemului autonom propus, evidențiind rolul fiecărei componente și modul în care acestea colaborează pentru a permite funcționarea în timp real a vehiculului.

De la selecția platformelor embedded (Raspberry Pi 5, Hailo-8L, Arduino Uno) până la integrarea aplicațiilor software (GStreamer, Flask, Socket.IO, dashboard-ul grafic), arhitectura sistemului a fost concepută pentru a asigura un echilibru între performanță, modularitate și scalabilitate.

Fiecare subsistem a fost analizat individual, dar și în contextul unui flux global de date și decizie, în care sincronizarea și comunicarea eficientă sunt esențiale.

Utilizarea GStreamer pentru procesarea video, combinată cu inferența accelerată pe Hailo și bucla de control PID pe Arduino, asigură o reacție rapidă la modificările din mediu, în timp ce interfața grafică permite o interacțiune intuitivă și ajustarea parametrilor în timp real.

Ultima secțiune a consolidat aceste elemente într-o perspectivă integratoare, subliniind importanța sinergiei între module și a unei sincronizări riguroase pentru menținerea unui comportament autonom robust. Arhitectura propusă oferă astfel o fundație solidă pentru implementarea și testarea unui sistem funcțional, capabil să îndeplinească cerințele unei navigații autonome reactive, în condiții controlate.

În capitolul următor vor fi detaliate etapele concrete de implementare ale acestei arhitecturi, incluzând dezvoltarea software-ului, integrarea hardware și testarea sistemului în scenarii reale de operare.

# Capitolul 3: Implementarea sistemului

Capitolul 3 prezintă implementarea sistemului de recunoaștere a semnelor de circulație, acoperind etapele esențiale ale dezvoltării acestuia.

Se începe cu pregătirea setului de date, urmată de antrenarea și conversia modelului pentru a fi compatibil cu hardware-ul specific.

Ulterior, se detaliază asamblarea componentelor hardware necesare funcționării sistemului. Implementarea în cod a tehnicilor de prelucrare a imaginilor, discutate în capitolul 1, este apoi abordată, evidențiind adaptările necesare pentru integrarea eficientă.

De asemenea, se analizează personalizarea pipeline-ului de detecție prin suprascrierea componentelor implicite ale GStreamer-ului din hailo-api, precum și logica serverului care gestionează comunicarea între componente. În final, se descrie dezvoltarea dashboard-ului interactiv și modul în care acesta se conectează la server pentru a oferi o interfață intuitivă de monitorizare și control al sistemului.

## Pregătirea modelului de inteligență artificială

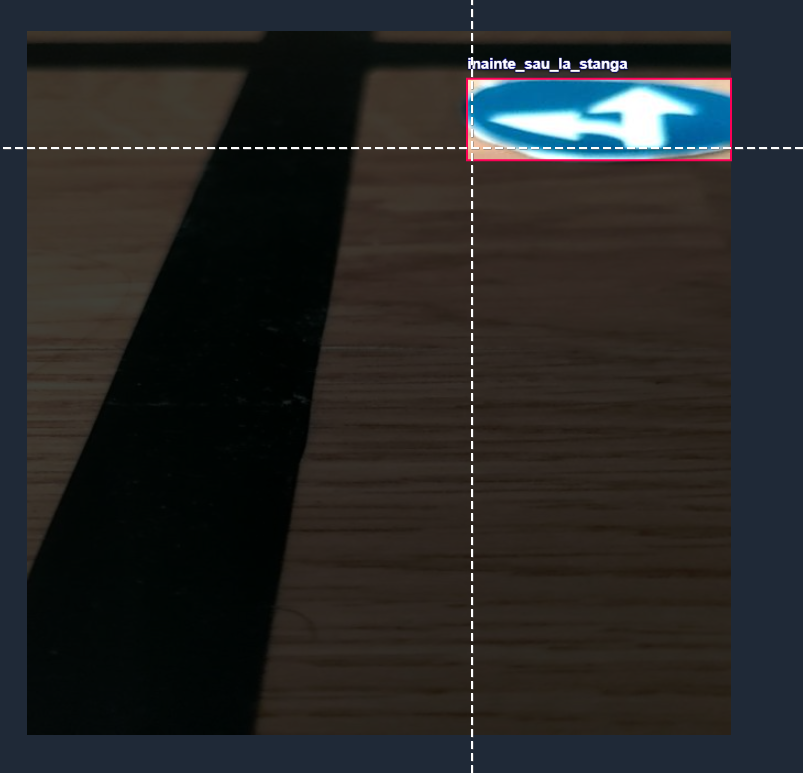
### **Colectarea și prelucrarea datelor**

Pentru antrenarea unui model de detecție a semnelor de circulație, s-a optat pentru construirea unui set de date personalizat, adaptat specificului mediului de operare al vehiculului autonom. În acest scop, au fost realizate fotografii și înregistrări video cu semne de circulație tipărite pe hârtie și amplasate pe podea, simulând scenarii de navigație realistă.

Imaginile au fost capturate din unghiuri variate și în condiții diferite de iluminare, pentru a reflecta cât mai bine posibilele contexte de rulare. Videoclipurile rezultate au fost apoi încărcate pe platforma Roboflow, unde s-a realizat procesul de extragere a cadrelor (frames) relevante, în special acelea în care semnele sunt clar vizibile.

Etichetarea s-a efectuat manual (exemplu prezentat în figura 3.1), direct în interfața grafică a Roboflow, prin trasarea de bounding boxes și atribuirea clasei semnului (ex: „la\_stanga”, „accesul\_interzis”, „stop” etc.).

Procesul a fost migălos, dar esențial pentru acuratețea finală a modelului. Platforma Roboflow a fost aleasă pentru interfața sa intuitivă, dar și pentru opțiunile avansate de augmentare și export în formate compatibile cu YOLOv8.

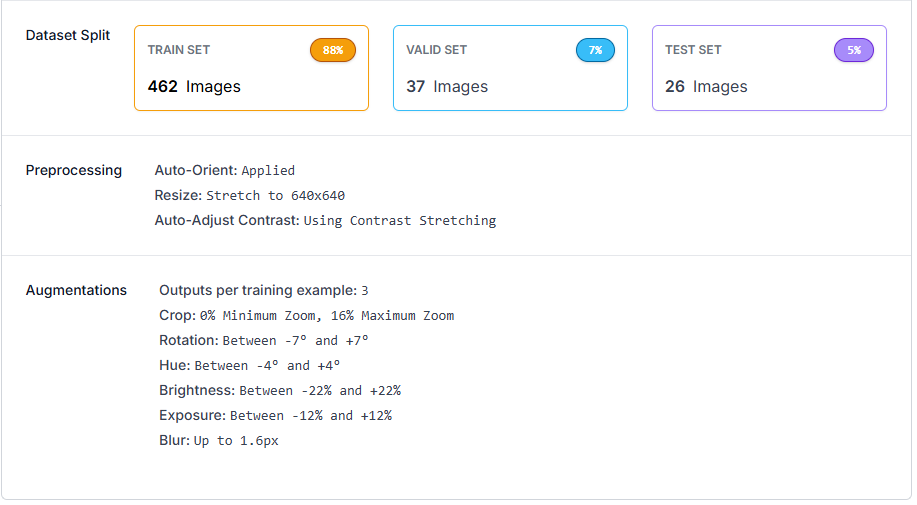


***Figura 3.1: Etichetarea manuală a imaginilor***

După adăugarea imaginilor, setul de date a fost procesat cu următoarele operații de preprocesare automată:

* Auto-Orient: activat – a aliniat toate imaginile în mod corect după metadatele EXIF.
* Redimensionare: toate imaginile au fost scalate la rezoluția de 640x640 pixeli, valoare acceptată implicit de majoritatea modelelor YOLO.
* Ajustarea automată a contrastului: folosind tehnica Contrast Stretching, pentru a maximiza distribuția de intensități din imagini.

În plus, au fost aplicate o serie de augmentări automate(vizibile în figura 3.2), pentru a crește diversitatea datelor de antrenament și a îmbunătăți capacitatea de generalizare a modelului. Aceste augmentări sunt esențiale în special pentru seturi de date mici,precum cel folosit pentru sistemul nostru.



***Figura 3.2: Informații despre setul de date***

Astfel, această etapă a asigurat generarea unui set de date diversificat, disponibil public la [39], echilibrat și bine etichetat, pregătit pentru a susține antrenarea eficientă a modelului YOLOv8, cu rezultate promițătoare în scenarii reale de navigație autonomă.

### **Antrenarea modelului YOLOv8**

După finalizarea procesului de colectare, etichetare și augmentare a datelor, următorul pas a fost antrenarea efectivă a unui model de detecție a obiectelor.

Pentru această sarcină s-a optat pentru arhitectura YOLOv8 (You Only Look Once) versiunea 8), pusă la dispoziție de librăria Ultralytics, datorită eficienței sale ridicate în sarcini de recunoaștere vizuală în timp real. YOLOv8 integrează numeroase optimizări față de versiunile anterioare, oferind o viteză ridicată de inferență, acuratețe crescută și o interfață de antrenare prietenoasă în Python.

Antrenarea a fost realizată utilizând interfața de tip API oferită de pachetul ultralytics, pe baza unui fișier de configurare data.yaml care descrie structura setului de date.

Setul de antrenament folosit este cel pregătit în subcapitolul anterior,folosind următorul cod Python ce folosesțe librăria Ultralytics:

model = YOLO("YOLOv8.pt")  
results = model.train(  
 data=os.path.join(ROOT\_DIR, "data.yaml"), epochs=1, batch=16,  
 amp=True, lr0=0.01, augment=True,  
 imgsz=640, save\_period=10, device=0  
)  
model.export(format="onnx")

În acest exemplu, modelul a fost antrenat timp de 100 de epoci pe un GPU dedicat, cu o dimensiune a imaginilor de 640×640 pixeli.

A fost activată și opțiunea de augmentare (augment=True) pentru a asigura diversitate în timpul antrenamentului, împreună cu AMP (automatic mixed precision), pentru o utilizare eficientă a resurselor de calcul.

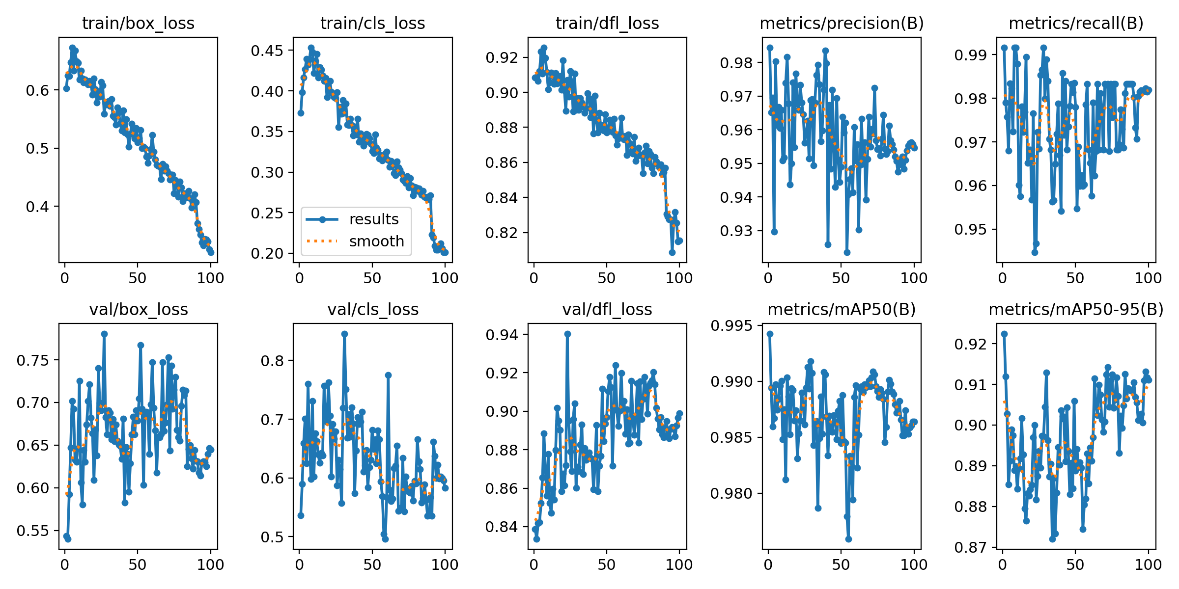
La final, modelul a fost exportat în format .onnx, un format intermediar standard compatibil cu compilatorul Hailo, pentru implementarea ulterioară pe hardware-ul specializat.

Figura de mai jos (Fig. 3.2) prezintă un eșantion vizual cu predicțiile modelului pe datele de validare, evidențiind atât corectitudinea poziționării bounding-box-urilor, cât și recunoașterea etichetelor corespunzătoare:



***Figura 3.3: Exemplu de predicții ale modelului YOLOv8 pe setul de validare***

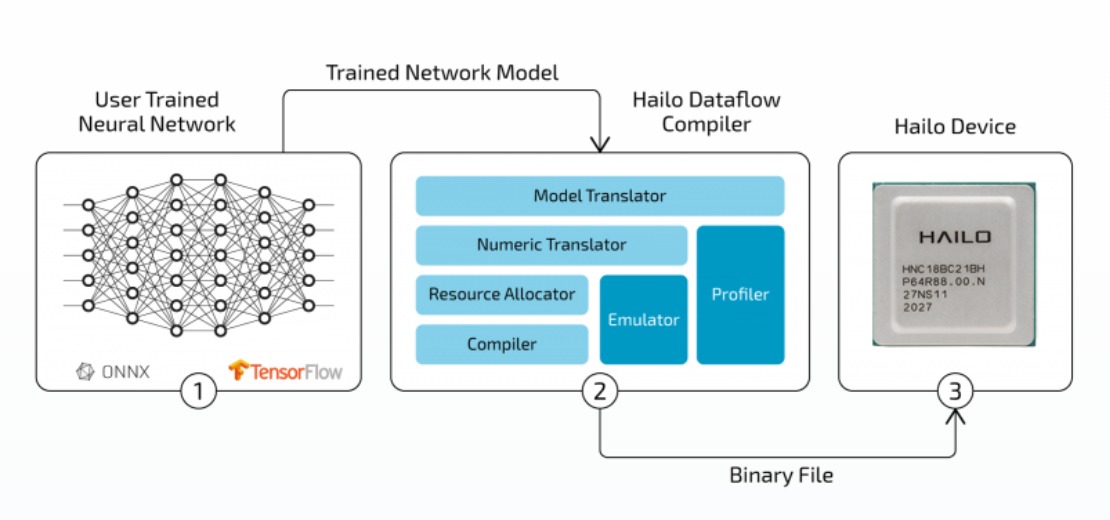
Performanța modelului pe parcursul celor 100 de epoci este prezentată sintetic în Figura 3.3, care include evoluția funcțiilor de cost (box loss, class loss, DFL loss) atât pentru setul de antrenare cât și pentru cel de validare, precum și valorile metrice de performanță: precizia, recall-ul și mAP-ul (mean Average Precision).



***Figura 3.4: Evoluția metricilor YOLOv8 în timpul antrenării***

### **Conversia modelului pentru implementare pe hardware Hailo**

După antrenarea si convertirea modelului de detecție în format ONNX, următorul pas esențial a fost conversia acestuia într-un format compatibil cu acceleratorul Hailo 8L, astfel încât să poată rula eficient pe hardware-ul dedicat. Acest proces de conversie este realizat cu ajutorul Hailo Dataflow Compiler, o unealtă oferită oficial de producător, care transformă modelul antrenat într-un fișier binar specializat, utilizabil pe dispozitivele Hailo. Arhitectura generală a acestui proces este ilustrată în figura 3.4.



***Figura 3.5: Arhitectura compilării unui model neuronal pentru Hailo***

În prima etapă (1), modelul neuronal antrenat de utilizator – fie în format ONNX, fie TensorFlow – este oferit ca intrare către compilatorul Hailo. În cazul acestui proiect, s-a utilizat modelul YOLOv8 exportat în format ONNX după antrenarea cu librăria Ultralytics.

Etapa a doua (2) constă în procesarea modelului prin diferitele module ale Hailo Dataflow Compiler:

* Model Translator interpretează structura rețelei neuronale și transformă straturile rețelei într-o reprezentare compatibilă cu arhitectura hardware specifică Hailo.
* Numeric Translator adaptează precizia numerelor și formatele datelor la specificațiile de inferență ale acceleratorului.
* Resource Allocator determină modul optim de împărțire a resurselor interne ale cipului Hailo (unități de calcul, memorie, etc.), pentru maximizarea performanței.
* Compiler generează efectiv codul binar pentru rulare.
* Emulatorul permite testarea modelului pe un simulator software, iar Profiler-ul oferă statistici legate de performanță și utilizarea resurselor.

Rezultatul procesului de compilare este un fișier binar cu extensia .hef (Hailo Executable File), care poate fi ulterior încărcat pe acceleratorul fizic (3) pentru rularea inferenței în timp real.

Pentru această etapă a fost utilizat WSL2 (Windows Subsystem for Linux), oferind un mediu Linux în interiorul sistemului Windows de dezvoltare, conform recomandărilor Hailo. În acest mediu, au fost instalate toate componentele Hailo SDK, incluzând hailo-compiler, utilitarul CLI cu ajutorul căruia s-a realizat efectiv conversia modelului:

hailo-compile yolov8\_model.onnx --hardware-target hailo8 --output yolov8\_model.hef

După obținerea fișierului .hef, a fost necesară și definirea unei configurații asociate rețelei, în format JSON. în care au fost specificate etichetele claselor detectate și diverse proprietăți utile pentru postprocesare. Acest fișier JSON este utilizat de API-ul Hailo în timpul execuției, pentru a traduce rezultatul brut al inferenței în etichete semantice interpretabile de aplicație.

{

    "iou\_threshold":0.6,

    "detection\_threshold": 0.85,

    "output\_activation":"none",

    "label\_offset":1,

    "max\_boxes": 200,

    "labels": [

        "unlabeled",

        "accesul\_interzis",

        "inainte\_sau\_la\_dreapta",

        "inainte\_sau\_la\_stanga",

        "la\_dreapta",

        "la\_stanga",

        "stop"

    ]

}

Întregul flux de lucru — de la exportul modelului în format ONNX, până la generarea fișierului .hef — a fost realizat pe baza tutorialului video disponibil la [38] care oferă o demonstrație clară a pașilor necesari.

Prin acest proces de compilare, modelul YOLOv8 este transformat dintr-o rețea generică într-o rețea optimizată pentru rulare pe arhitectura paralelă a cipului Hailo, cu latență redusă și consum energetic eficient.

Acest aspect este esențial pentru funcționarea în timp real a vehiculului autonom, mai ales în condițiile unui sistem embedded cu resurse limitate.

## Asamblarea componentelor hardware

După pregătirea modelului de detecție și conversia sa în formatul compatibil cu acceleratorul Hailo, etapa următoare a constat în asamblarea platformei hardware care să permită testarea practică a funcționalităților sistemului. Figura 3.6 ilustrează platforma fizică a vehiculului autonom, alcătuită dintr-un șasiu cu două roți motrice, o placă Raspberry Pi 5 cu modulul Hailo-8L HAT montat, un microcontroler Arduino Uno R3, un modul de control motoare L298N și două surse de alimentare independente.



***Figura 3.6: Componentele fizice ce vor fi asamblate***

Platforma de bază este construită pe un șasiu din plexiglas, prevăzut cu două motoare DC, un suport pentru 6 baterii AA (9V total) și un spațiu dedicat montării componentelor electronice. Asamblarea mecanică a șasiului, a roților și a suporturilor pentru baterii a fost realizată urmând pașii descriși în tutorialul video[40], unde sunt explicate în detaliu etapele necesare pentru obținerea unei structuri stabile și funcționale pentru prototip.

Placa Raspberry Pi 5 este montată într-o carcasă imprimată 3D, prevăzută cu ventilație pentru răcire pasivă. Pe aceasta este conectat modulul Hailo-8L prin interfața PCIe, care permite accelerarea inferenței YOLOv8. Raspberry Pi este alimentat independent de restul sistemului printr-o baterie externă portabilă (powerbank), vizibilă în imagine în partea stângă-jos.

Această decizie de alimentare separată asigură stabilitate electrică și elimină riscul de interferență sau suprasarcină între sistemul de procesare și partea de acționare.

De cealaltă parte, sistemul de acționare – alcătuit din Arduino Uno R3 și driverul L298N – este alimentat de un pachet de 6 baterii AA, oferind o tensiune adecvată pentru a comanda motoarele DC și a alimenta logica driverului. Conectarea Arduino-ului la driverul L298N a fost realizată conform recomandărilor din videoclipul[41], care explică detaliat cum trebuie alimentat circuitul fără a produce conflicte între sursele de tensiune ale Arduino-ului și Raspberry Pi-ului.

Pentru a permite comunicarea între componente, Raspberry Pi și Arduino sunt conectate prin cablu USB, care asigură transferul comenzilor seriale de control.

Liniile de semnal PWM și de direcție (IN1–IN4, ENA/ENB) între Arduino și L298N sunt trasate pe o placă de montaj prototip și conectate prin fire DuPont. Totodată, camera oficială Raspberry Pi v2.1 este fixată deasupra carcasei, conectată printr-un cablu plat la interfața CSI a Raspberry-ului, oferind astfel o poziționare stabilă pentru captarea imaginilor în timpul deplasării vehiculului.

Această structură modulară și alimentare separată pentru fiecare bloc funcțional oferă un echilibru între performanță, stabilitate și siguranță electrică, esențială pentru funcționarea sistemului în scenarii reale.

Designul permite testarea individuală a fiecărei componente, precum și întreținerea sau extinderea facilă a platformei în etapele următoare ale dezvoltării.

## Implementarea tehnicilor de prelucrare a imaginiilor

Implementarea componentelor de prelucrare a imaginilor în sistemul de navigație autonomă s-a realizat utilizând biblioteca OpenCV (cv2), o unealtă extrem de versatilă pentru procesarea imaginilor în timp real. Această bibliotecă oferă funcții optimizate pentru conversii de culoare, transformări geometrice, filtrare, segmentare, morfologie digitală și extragerea contururilor, toate fiind utilizate în pipeline-ul de viziune artificială implementat în cadrul proiectului.

Prima etapă a prelucrării o constituie conversia spațiului de culoare. Deoarece fluxul de date primit de la cameră este în RGB, iar OpenCV utilizează implicit formatul BGR, a fost necesară conversia fiecărui cadru folosind următoarea instrucțiune:

current\_frame = cv2.cvtColor(current\_frame, cv2.COLOR\_RGB2BGR)

Această conversie asigură afișarea corectă a culorilor în cadrul procesului de detecție și este aplicată imediat după capturarea fiecărui cadru.

Pentru a corecta orientarea camerei montate invers pe vehicul, fiecare imagine este rotită la 180 de grade printr-o funcție definită ce utilizează funcțiile cv2.getRotationMatrix2D() și cv2.warpAffine().

Codul funcției folosite pentru rotație este:

def rotate(image, angle):

    h, w = image.shape[:2]

    center = (w // 2, h // 2)

    rotation\_matrix = cv2.getRotationMatrix2D(center, angle, 1.0)

    rotated\_image = cv2.warpAffine(image, rotation\_matrix, (w, h))

    return rotated\_image

După rotație, sistemul aplică o filtrare Gaussiană pentru a reduce zgomotul provenit din vibrațiile mișcării sau iluminare variabilă. Implementarea utilizată este:

current\_frame = cv2.GaussianBlur(current\_frame, (5, 5), 0)

Alegerea unui kernel 5x5 s-a dovedit eficientă în eliminarea zgomotului fără a pierde detalii relevante pentru detecția traseului.

Pentru a compensa pierderea detaliilor cauzată de filtrare, se aplică un kernel de ascuțire:

sharpen\_kernel = np.array([[-1, -1, -1], [-1, 9, -1], [-1, -1, -1]])

current\_frame = cv2.filter2D(current\_frame, -1, sharpen\_kernel)

Această etapă evidențiază contururile liniei negre și alte margini relevante, contribuind la segmentarea eficientă a traseului.

Pentru identificarea traseului, sistemul folosește segmentarea bazată pe culoare, izolând linia neagră folosind:

thresh=cv2.inRange (current\_frame, lower\_black, upper\_black)

Aceasta creează o imagine binară unde linia este reprezentată cu alb (255) și restul cu negru (0).

Pentru a corecta zgomotul și discontinuitățile, sistemul aplică operații morfologice:

thresh = cv2.erode(

thresh, np.ones((5, 5), np.uint8), iterations=pipeline\_server.erode\_count)

thresh = cv2.dilate(

thresh, np.ones((5, 5), np.uint8), iterations=pipeline\_server.dilate\_count)

Erodarea elimină punctele izolate, iar dilatarea reface grosimea liniei detectate, restabilind continuitatea acesteia.

Ulterior, imaginea binară este analizată pentru a identifica forma și complexitatea traseului. Se extrag contururile folosind cv2.findContours() și se aplică algoritmul Ramer–Douglas–Peucker pentru a aproxima contururile cu poligoane simplificate. Această etapă este esențială pentru identificarea intersecțiilor sau curbelor majore din traseu, în funcție de numărul de vârfuri identificat.

Toate aceste etape sunt implementate secvențial în pipeline-ul de viziune artificială și sunt executate în timp real pe fiecare cadru capturat, asigurând sistemului o percepție clară și actualizată a traseului.

Prelucrarea imaginilor joacă astfel un rol critic în furnizarea de informații utile modulului de control, facilitând o navigație autonomă precisă și stabilă.

## Implementarea controlului vehicului

Sistemul propus utilizează un pipeline complet de procesare a imaginilor și control embedded, în care deciziile sunt luate pe baza analizei vizuale efectuate de Raspberry Pi și transmise către Arduino pentru acționare. În acest proces, Raspberry Pi interpretează imaginea captată de cameră, identifică linia de ghidaj sau alte repere vizuale, și calculează deviația (eroarea) față de traiectoria ideală. Ulterior, transmite această informație către Arduino printr-o conexiune serială UART.

Mesajele trimise prin serial respectă o structură clară: prima literă este un header care indică tipul comenzii, urmată de unul sau mai mulți parametri, separați prin virgulă. Arduino interpretează header-ul și extrage parametrii pentru a executa acțiunea corespunzătoare. Această metodă de comunicare oferă flexibilitate și claritate, facilitând controlul autonom, manual, sau de siguranță.

Un mesaj de bază utilizat pentru controlul autonom este cel cu header-ul E, urmat de o valoare numerică ce reprezintă eroarea de poziționare calculată de Raspberry Pi. Această eroare va fi trimisă mai departe spre controller-ul PID:

if (command.startsWith("E")) {

      int first\_comma = command.indexOf(',');

      int second\_comma = command.indexOf(',', first\_comma + 1);

      if (first\_comma != -1) {

        double error = command.substring(first\_comma + 1,

                      second\_comma != -1 ? second\_comma : command.length()).toFloat();

        if (second\_comma != -1) {

          String mode\_str = command.substring(second\_comma + 1);

          if (mode\_str == "F") current\_mode = MODE\_FORWARD;

          else if (mode\_str == "L") current\_mode = MODE\_LEFT;

          else if (mode\_str == "R") current\_mode = MODE\_RIGHT;

          else if (mode\_str == "S") current\_mode = MODE\_STOP;

        }

        apply\_pid\_with\_mode(error);

      }

    }

Când sistemul se află în modul manual, poate primi un mesaj cu header-ul H, urmat de două valori: vitezele motoarelor stâng și drept. Acestea sunt aplicate direct, fără a activa regulatorul PID:

else if (command.startsWith("H")) {

      int firstComma = command.indexOf(',');

      int secondComma = command.indexOf(',', firstComma + 1);

      if (firstComma != -1 && secondComma != -1) {

        int left = command.substring(firstComma + 1, secondComma).toInt();

        int right = command.substring(secondComma + 1).toInt();

        set\_motors(left, right);

        reset\_pid();

      }

}

Pentru reglajul comportamentului sistemului, se poate modifica dinamic PID-ul printr-un mesaj cu header-ul P, urmat de valorile Kp, Ki și Kd:

else if (command.startsWith("P")) {

      int first = command.indexOf(',');

      int second = command.indexOf(',', first + 1);

      int third = command.indexOf(',', second + 1);

      if (first != -1 && second != -1 && third != -1) {

        Kp = command.substring(first + 1, second).toFloat();

        Ki = command.substring(second + 1, third).toFloat();

        Kd = command.substring(third + 1).toFloat();

        update\_pid\_limits();

        reset\_pid();

      }

}

Pentru a seta viteza de deplasare în regim normal, Raspberry Pi trimite un mesaj cu header-ul S:

else if (command.startsWith("S")) {

      int comma = command.indexOf(',');

      if (comma != -1) {

        base\_speed = command.substring(comma + 1).toInt();

        base\_speed = constrain(base\_speed, 0, MAX\_SPEED);

        update\_pid\_limits();

      }

}

Mesajul cu header R este folosit pentru resetarea valorilor PID:

else if (command == "R") {

      reset\_pid();

}

Oprirea de urgență este activată prin comanda X, care dezactivează complet motoarele și PID-ul:

else if (command == "X") {

      current\_mode = MODE\_STOP;

      set\_motors(0, 0);

      reset\_pid();

}

Această abordare orientată pe mesaje și identificatori expliciți face ca fluxul de comenzi dintre Raspberry Pi și Arduino să fie eficient și extensibil. Protocolul poate fi completat cu noi comenzi, fără a afecta funcționarea sistemului existent.

## Implementarea pipeline-ului pentru navigare autonomă

### **Implementarea pipeline-ului GStreamer de inferență**

Pipeline-ul GStreamer utilizat în sistem este definit în fișierul source\_pipeline.py și reprezintă lanțul de procesare prin care trec datele video de la captură până la inferență și vizualizare. Acest pipeline este construit dintr-o secvență de elemente GStreamer interconectate, fiecare având un rol specific în fluxul de procesare1.

Definirea pipeline-ului de sursă se realizează printr-o funcție simplă dar esențială care configurează elementul de intrare pentru fluxul video:

def SOURCE\_PIPELINE(video\_format="RGB", name="app\_source"):

    return (

        f"appsrc name={name} is-live=true leaky-type=downstream max-buffers=1 ! "

        f"video/x-raw,format={video\_format} ! "

    )

Elementul fundamental appsrc este configurat cu proprietăți critice pentru funcționarea în timp real: is-live=true indică faptul că sursa produce date în timp real, leaky-type=downstream permite eliminarea bufferelor vechi când sistemul nu poate procesa suficient de rapid, iar max-buffers=1 limitează memoria utilizată și reduce latența prin menținerea unui singur buffer activ.

Pipeline-ul de afișare integrează mai multe componente pentru procesarea și vizualizarea datelor:

def DISPLAY\_PIPELINE(

    video\_sink="autovideosink", sync="false", show\_fps="true", name="hailo\_display"

):

    display\_pipeline = (

        f'{OVERLAY\_PIPELINE(name=f"{name}\_overlay")} ! '

        f'{QUEUE(name=f"{name}\_videoconvert\_q")} ! '

        f'videoconvert name={name}\_videoconvert n-threads=2 qos=false ! '

        f'{QUEUE(name=f"{name}\_q")} ! '

        f'fpsdisplaysink name={name} video-sink={video\_sink} sync=false text- overlay={show\_fps} signal-fps-measurements=true '

    )

    return display\_pipeline

Această configurație include OVERLAY\_PIPELINE pentru suprapunerea detecțiilor pe imagine, elemente QUEUE pentru gestionarea bufferelor între stagii, videoconvert cu două thread-uri pentru conversiile de format și fpsdisplaysink pentru afișarea finală cu măsurarea performanțelor în timp real.

### **Implementarea aplicației GStreamer**

Clasa personalizată GStreamer moștenește funcționalitatea de bază din GStreamerApp și o extinde cu logica specifică aplicației de navigație autonomă.

Această clasă funcționează ca un orchestrator complex care coordonează toate aspectele procesării video:

class CustomGStreamerDetectionApp(GStreamerApp):

    def \_\_init\_\_(self, app\_callback, user\_data):

        parser = get\_default\_parser()

        parser.add\_argument(

            "--labels-json",

            default=None,

            help="Path to custom labels JSON file")

        parser.add\_argument(

            "--headless",

            action="store\_true",

            help="Run the application in headless mode (no display).")

        parser.add\_argument(

            "--stream-address",

            action="store\_true",

            help="Streams address of the video source.")

        super().\_\_init\_\_(parser, user\_data)

        args = parser.parse\_args()

Inițializarea clasei include configurarea argumentelor specifice aplicației, cum ar fi fișierul de etichete JSON pentru clasificarea obiectelor și modul headless pentru funcționarea fără interfață grafică. Parametrii critici pentru procesarea video sunt definiți explicit:

self.batch\_size = 2

self.video\_width=800

self.video\_height=600

nms\_score\_threshold = 0.5

nms\_iou\_threshold = 0.5

Acești parametri controlează dimensiunea batch-ului pentru inferență, rezoluția de procesare și pragurile pentru algoritmul Non-Maximum Suppression folosit în post-procesarea detecțiilor.

Un aspect crucial al implementării este crearea unui pipeline separat pentru streaming video către dashboard:

self.stream\_pipeline = Gst.parse\_launch(

f"appsrc name=stream\_src format=time is-live=true leaky-type=downstream max- buffers=1 "

       f"caps=video/x- raw,format=RGB,width={self.video\_width},height={self.video\_height},framerate=30/1 ! videorate ! "

       f"videoconvert ! {OVERLAY\_PIPELINE(name='hailo\_overlay')} ! "

       f"jpegenc ! tcpserversink port=4956 sync=false async=false"

)

Acest pipeline dedicat permite transmiterea în timp real a imaginilor procesate către dashboard prin TCP pe portul 4956, utilizând compresie JPEG pentru optimizarea benzii de date.

Funcția de callback pentru procesarea cadrelor implementează logica complexă de analiză și streaming:

def on\_new\_sample(self, sink) -> Gst.FlowReturn:

try:

            sample = sink.emit("pull-sample")

            if not sample:

                return Gst.FlowReturn.ERROR

            buf, \_, width, height = get\_frame\_info(sink.get\_static\_pad("sink"), sample)

            success, mapinfo = buf.map(Gst.MapFlags.READ)

            if not success:

                return Gst.FlowReturn.ERROR

            frame = np.frombuffer(mapinfo.data, dtype=np.uint8)

            frame = frame.reshape((height, width, 3))

            frame\_copy = cv2.resize(frame, (self.video\_width, self.video\_height))

Această implementare extrage datele din buffer-ul GStreamer, le convertește într-un array NumPy și le redimensionează la rezoluția standard pentru procesare.

Ajustările de luminozitate și contrast sunt aplicate dinamic pe baza setărilor din server:

adjusted\_frame = cv2.addWeighted(frame\_copy,pipeline\_server.contrast,

                np.zeros(frame\_copy.shape, frame\_copy.dtype), 0,

                pipeline\_server.brightness)

### **Implementarea logicii de navigare autonomă**

Logica de navigare autonomă este realizată în cadrul aplicației GStreamer, folosind cadre video preluate în timp real.

Pe baza acestora, sistemul decide în mod autonom mișcările vehiculului, prelucrarea inițială a imaginilor fiind deja detaliată în subcapitolul 3.3.

#### **Inițializare și captură video**

La pornirea aplicației, este lansat pipeline-ul GStreamer și serverul Flask pentru modificarea parametrilor în timp real:

pipeline\_server.start()

app\_instance = CustomGStreamerDetectionApp(app\_callback, user\_data)

app\_instance.run()

#### **Detectarea liniei și calculul erorii**

După preprocesare, se aplică o segmentare pentru extragerea liniei negre, urmată de erodare și dilatare. Se identifică conturul cel mai mare și se calculează eroarea față de centrul imaginii:

contours, \_ = cv2.findContours(blackline, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)

c = max(contours, key=cv2.contourArea)

M = cv2.moments(c)

if M["m00"] != 0:

        cx = int(M["m10"] / M["m00"])

error = cx - 320

#### **Comunicare cu Arduino și control PID**

Eroarea calculată este transmisă microcontrolerului Arduino, care aplică un algoritm PID pentru controlul motoarelor:

command = f"E,{error}\n"

arduino.write(command.encode("utf-8"))

#### **Detectarea intersecțiilor**

Se aproximează contururile și se identifică cele care au un număr mare de colțuri – interpretate drept intersecții:

approx = cv2.approxPolyDP(c, 0.02 \* cv2.arcLength(c, True), True)

if approx is not None and len(approx) >= 6:

user\_data.can\_process\_intersection = True

#### **Recunoașterea semnelor de circulație**

Se analizează doar semnele detectate în zona inferioară a cadrului pentru a exclude zgomotul:

lower\_threshold = frame\_height \* pipeline\_server.height\_modifier

valid\_detections = []

for det in detections:

bbox = det.get\_bbox()

       y\_center = (bbox.ymin() + bbox.ymax()) / 2 \* frame\_height

      if y\_center < lower\_threshold:

        valid\_detections.append(det.get\_label())

action, \_ = determine\_action(valid\_detections)

#### **Gestionarea manevrelor în intersecții**

Dacă este detectat un semn valid (ex. „stop”, „acces interzis”), se inițiază o secvență de oprire urmată de direcționare:

if action:

user\_data.sign\_detected = True

       sign\_direction = COMMAND\_MAP[action]

      if sign\_direction == DIRECTION\_STOP:

        direction = DIRECTION\_STOP

              user\_data.pending\_direction = None

       else:

              user\_data.found\_direction = sign\_direction

În cadrul buclei principale de procesare a cadrelor :

if user\_data.pending\_direction:

if user\_data.forward\_frames\_ > 0:

user\_data.forward\_frames\_ -= 1

        return DIRECTION\_FORWARD, current\_frame, blackline, dynamic\_modifier

if user\_data.direction\_frames > 0:

              if user\_data.stop\_frames > 0:

                    user\_data.stop\_frames -= 1

                    return DIRECTION\_STOP, current\_frame, blackline, dynamic\_modifier

              user\_data.direction\_frames -= 1

              return user\_data.pending\_direction,current\_frame, blackline,dynamic\_modifier

#### **Tranziția între stări**

După finalizarea unei manevre, sistemul revine la comportamentul standard de urmărire a liniei:

if user\_data.direction\_frames > 0:

user\_data.pending\_direction = None

# Reluare trimitere comenzi de tip "E,eroare"

Această implementare modulară asigură un comportament reactiv și predictibil în scenarii variate de navigație, incluzând trasee sinuoase, intersecții și semnalizare rutieră.

## Implementarea server-ului Flask

Server-ul Flask reprezintă principala modalitate de comunicare între sistem și alte componente externe.

Acesta pornește odată cu aplicația ,însă este gestionată de un fir de execuție separat:

def start(self):

        self.server\_thread = thread.Thread(

            target=lambda: socketio.run(

                app, host="0.0.0.0", port=5000, allow\_unsafe\_werkzeug=True

            ),

            daemon=True,

        )

        self.server\_thread.start()

Server-ul primește mesaje de la surse externe (dashboard-ul în cazul nostru), și acualizează coresunzător componentele sistemului.

Un exemplu ar fi intrarea sistemului în modul de navigație manual, care este transmis prin mesajul "set\_mode", mesaj care odată receptionat actualizează valoarea variabilei responsabile de urmărirea modului de control pe modul manual.

@socketio.on('set\_mode')

def handle\_set\_mode(data):

    mode = data.get('mode', 'auto')

    pipeline\_server.set\_control\_mode(mode)

    emit('mode\_ack', {'mode': mode}, broadcast=False)

În mod analog sunt setate atât valorile parametrilor de control al intersecției cât și cele de control al imaginii.

Această implementare permite un control granular și în timp real al parametrilor critici ai sistemului de navigație, fără a fi necesară repornirea aplicației sau recompilarea codului.

## Implementarea dashboard-ului

Dashboard-ul reprezintă interfața grafică principală prin care utilizatorul interacționează cu sistemul embedded de navigație autonomă. Acesta este implementat în Python cu biblioteca Tkinter și oferă o interfață intuitivă și completă pentru controlul vehiculului, monitorizarea în timp real a fluxului video și ajustarea parametrilor critici ai sistemului. Comunicarea cu serverul Flask, găzduit pe Raspberry Pi, este realizată prin Socket.IO, asigurând un canal bidirecțional de comunicare în timp real, esențial pentru control și feedback rapid.

La inițializarea aplicației, fereastra principală Tk() este populată cu mai multe secțiuni logice: panoul de stream video, panoul de control pentru comutarea modului (automat/manual), zona de reglare a PID-ului, secțiunea de configurare a procesării imaginilor și un tab dedicat pentru vizualizarea statisticilor de detecție. Interfața este modulară și permite extinderea ușoară a funcționalităților.

Componenta de afișare a fluxului video folosește un Label în care este actualizată în mod regulat o imagine JPEG decodificată, obținută prin requests.get(...) de la endpointul /video al serverului Flask. Imaginea este convertită folosind PIL.Image și afișată în format compatibil cu Tkinter (folosind ImageTk.PhotoImage), actualizată periodic cu after(15, update\_video), asigurând astfel o rată de refresh de aproximativ 60 FPS.

frame = Image.open(BytesIO(response.content))  
photo = ImageTk.PhotoImage(image=frame)  
video\_label.config(image=photo)  
video\_label.image = photo  
root.after(15, update\_video)

De asemenea, dashboard-ul include butoane și controale tip Radiobutton, Checkbutton, Scale, Spinbox, Entry și Button, toate conectate la variabile de tip StringVar, IntVar sau DoubleVar, care permit actualizarea în timp real a setărilor. La modificarea acestor controale, sunt declanșate funcții de tip callback, care folosesc sio.emit() pentru a trimite datele către server.

Un exemplu relevant este controlul coeficienților PID. Utilizatorul poate ajusta valorile lui Kp, Ki și Kd prin Entry-uri dedicate, iar la apăsarea unui buton „Send PID”, dashboard-ul emite un eveniment de tip set\_pid, astfel:

pid\_data = {'kp': float(kp.get()), 'ki': float(ki.get()), 'kd': float(kd.get())}  
sio.emit('set\_pid', pid\_data)

Pe server, acest mesaj este interceptat în @socketio.on('set\_pid') și valorile sunt transmise către controlerul Arduino, actualizând parametrizarea buclei PID.

Fluxul video este afișat în panoul central, utilizând un Label care primește periodic cadre video codate JPEG din serverul Flask. Acestea sunt decodificate și afișate cu ajutorul PIL.Image și ImageTk.PhotoImage, folosind un after(10, update\_label) pentru a menține un refresh fluid (~60 FPS).

Pentru depanare, dashboard-ul oferă opțiunea de a activa un „debug overlay”, care suprapune pe imagine diverse informații utile pentru dezvoltator: zona de detecție, centrul traseului, vectorii de eroare etc.

Aceasta este controlată printr-un buton de tip Checkbutton:

debug\_var = tk.BooleanVar(value=False)  
debug\_check = ttk.Checkbutton(  
 debug\_controls\_frame,  
 text="Show Debug Overlays",  
 variable=debug\_var,  
 command=lambda: sio.emit('toggle\_debug',  
 {'enabled': debug\_var.get()}) if sio.connected else None  
)

Odată activată, această opțiune transmite serverului comanda toggle\_debug, care determină pipeline-ul să adauge aceste suprapuneri peste imaginea transmisă, utilă mai ales în testarea algoritmilor de segmentare sau control.

Un aspect important al dashboard-ului este posibilitatea de a comuta între modul automat și cel manual, în care utilizatorul controlează direct vehiculul. Pentru acest scop, aplicația include un joystick virtual, implementat în fișierul joystick.py.

Acesta este o componentă personalizată derivată din Canvas, care desenează o zonă circulară interactivă, un buton mobil (knob) și marcaje pentru direcții.

Pe baza mișcării utilizatorului cu mouse-ul, se calculează vectorul de direcție și o viteză proporțională.

def \_send\_command(self, dx, dy, dist):  
 norm\_x = dx / self.radius  
 norm\_y = -dy / self.radius  
 normalized\_dist = min(1.0, (dist - self.deadzone) / (self.radius - self.deadzone))  
 self.motor\_left = normalized\_dist \* (norm\_y - norm\_x) \* self.max\_speed  
 self.motor\_right = normalized\_dist \* (norm\_y + norm\_x) \* self.max\_speed  
 self.motor\_left = max(-self.max\_speed, min(self.max\_speed, self.motor\_left))  
 self.motor\_right = max(-self.max\_speed, min(self.max\_speed, self.motor\_right))  
 if self.send\_callback:  
 self.send\_callback(self.motor\_left, self.motor\_right)  
 else:  
 try:  
 print(f"Motors: L={self.motor\_left:.0f}, R={self.motor\_right:.0f}")  
 requests.post(  
 'http://localhost:5000/manual',  
 json={'left': self.motor\_left, 'right': self.motor\_right},  
 timeout=0.5  
 )  
 except Exception as e:  
 print(f"Command send error: {e}")

Callback-ul send\_callback este o funcție definită în Dashboard.py, care emite un eveniment manual\_control prin Socket.IO:

joystick = Joystick(  
 master=control\_frame,  
 size=150,  
 send\_callback=lambda d, s: sio.emit('manual\_cmd', {'direction': d, 'speed': s})  
)

Serverul Flask procesează acest eveniment prin handlerul:

@socketio.on('manual\_cmd')

def handle\_manual\_cmd(data):

    pipeline\_server.set\_manual\_state(data['direction'], data['speed'])

Dashboard-ul este conectat permanent la server printr-un client Socket.IO, instanțiat cu:

sio = socketio.Client()

sio.connect('http://localhost:5000')

Mesajele trimise de server sunt interceptate prin decoratori de tip @sio.on(...). Un exemplu este mesajul detection, care conține informații despre obiectele detectate. La recepție, dashboard-ul actualizează o imagine cu semnul de circulație detectat, afișată într-o fereastră separată:

@sio.on('detection')  
def on\_detection(data):  
 with open("temp.png", "wb") as f:  
 f.write(base64.b64decode(data['img']))  
 *# display in Tkinter window*

Această funcționalitate oferă utilizatorului un feedback vizual clar despre ceea ce „vede” sistemul în timp real.

Aplicația este construită într-un mod modular, fiecare panou (stream video, comenzi, PID, joystick) fiind definit ca o secțiune separată, organizată în Frame-uri Tkinter. Astfel, se poate extinde ușor cu noi funcționalități, fără a afecta logica existentă.

Un avantaj suplimentar este și suportul pentru comutarea modurilor de operare.

De exemplu, prin selectarea unui Radiobutton, dashboard-ul poate trimite un eveniment set\_mode, prin care utilizatorul decide dacă sistemul funcționează în mod complet autonom sau manual:

sio.emit('set\_mode', {'mode': mode\_var.get()})

# Capitolul 4: Prezentarea sistemului

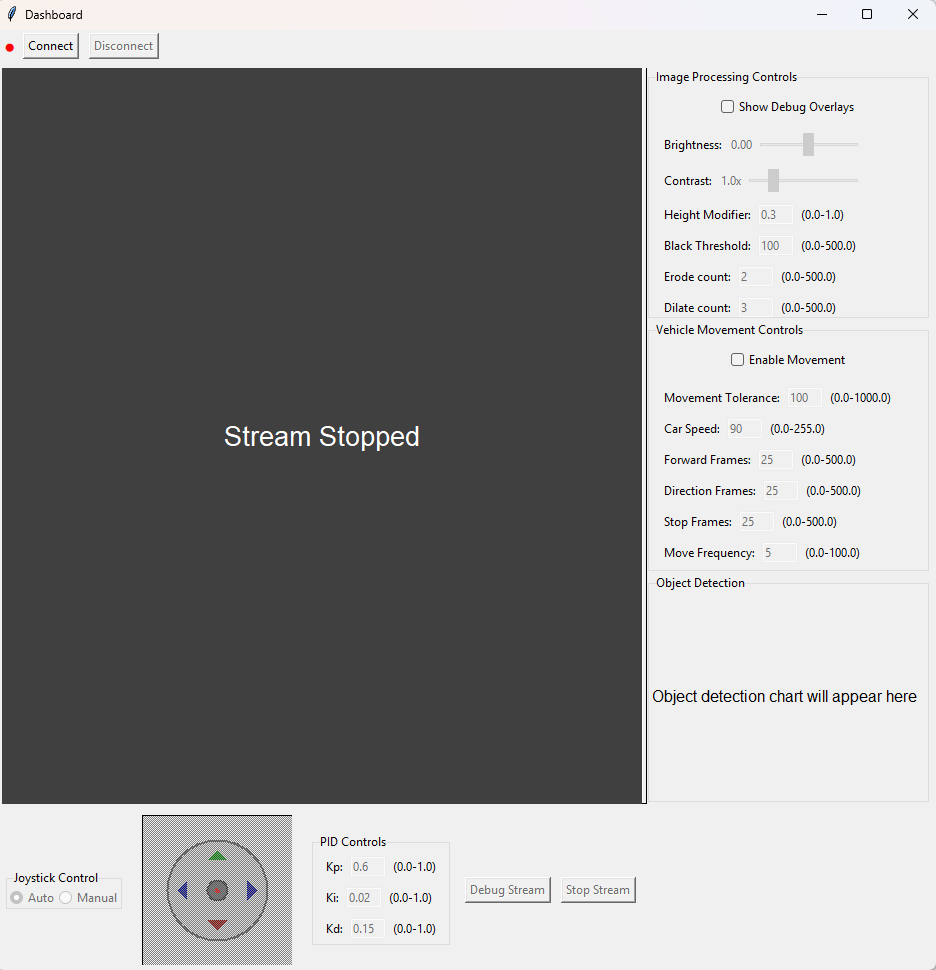
Acest capitol este dedicat prezentării detaliate a interfeței grafice de control (dashboard) și a scenariilor de operare ale vehiculului autonom. Interfața servește drept centru de comandă și monitorizare, permițând utilizatorului să interacționeze cu sistemul, să ajusteze parametrii și să vizualizeze în timp real comportamentul mașinii.

Interfața grafică este proiectată pentru a fi intuitivă și funcțională, centralizând toate elementele de control și monitorizare necesare pentru operarea și depanarea sistemului.

## Prezentarea ecranului inițial

La pornirea aplicației, utilizatorul este întâmpinat de ecranul inițial. În acest stadiu, fluxul video este oprit ("Stream Stopped"), iar vehiculul este inactiv. Panoul de control din dreapta afișează setările implicite pentru diverși parametri, cum ar fi luminozitatea (0.0), contrastul (1.0x) și pragul pentru detecția liniei negre (100)

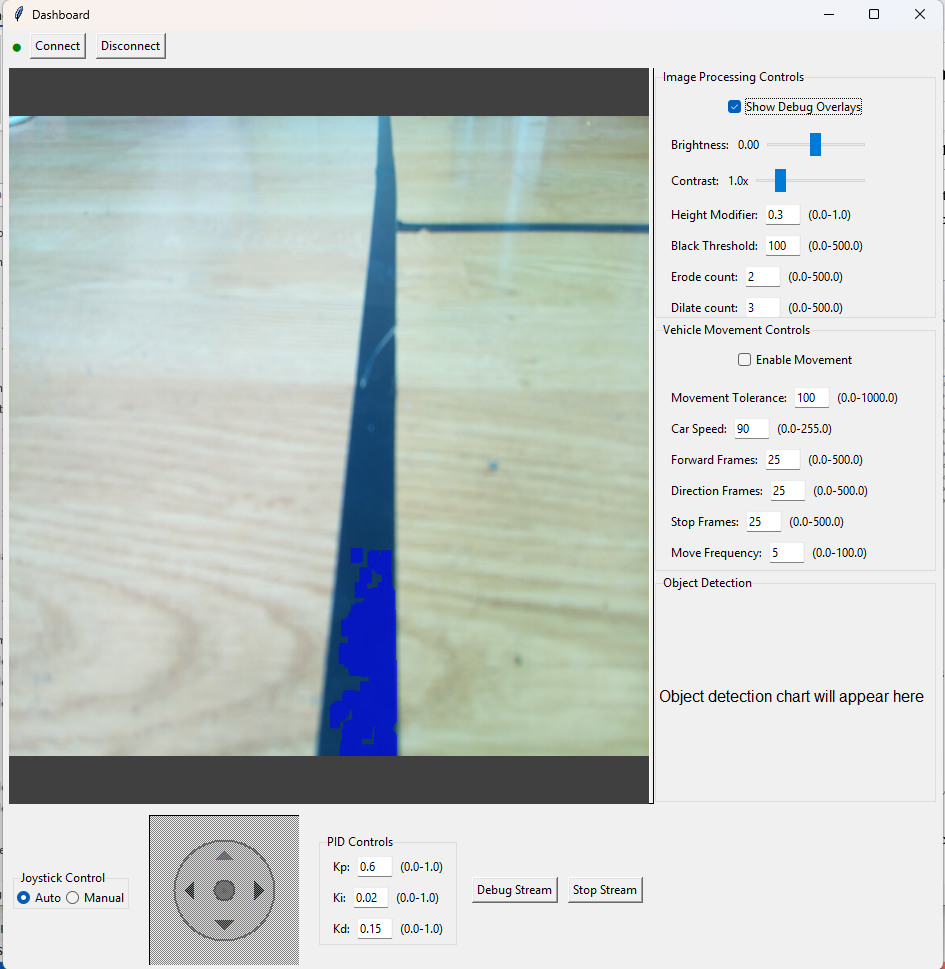
Utilizatorul are la dispoziție butoane pentru a porni sau opri fluxul video și pentru a comuta între modurile de control



***Figura 4.1: Ecran initial***

## Prezentarea fluxului video

La apăsarea butonului "Start Stream", sistemul inițializează camera și începe transmiterea fluxului video către dashboard. În acest moment, se poate observa imaginea captată de camera vehiculului, afișând mediul înconjurător. Inițial, valorile parametrilor de imagine și de procesare sunt cele standard, iar vehiculul rămâne în așteptare, fără a iniția vreo mișcare autonomă. Pe ecran poate fi observat un obiect albastru pe o suprafață de lemn, reprezentând traseul pe care mașina îl va urma



***Figura 4.2: Flux video deschis cu valori inițiale***

## Prezentarea modurilor de funcționare

Sistemul poate funcționa în două moduri principale: manual și automat, fiecare având sub-scenarii specifice.

## Modul manual de funcționare

În modul manual, utilizatorul preia controlul complet asupra mișcărilor vehiculului prin intermediul unui joystick virtual integrat în dashboard

Acest mod este ideal pentru testarea motoarelor, calibrarea senzorilor sau navigarea în zone necartografiate.

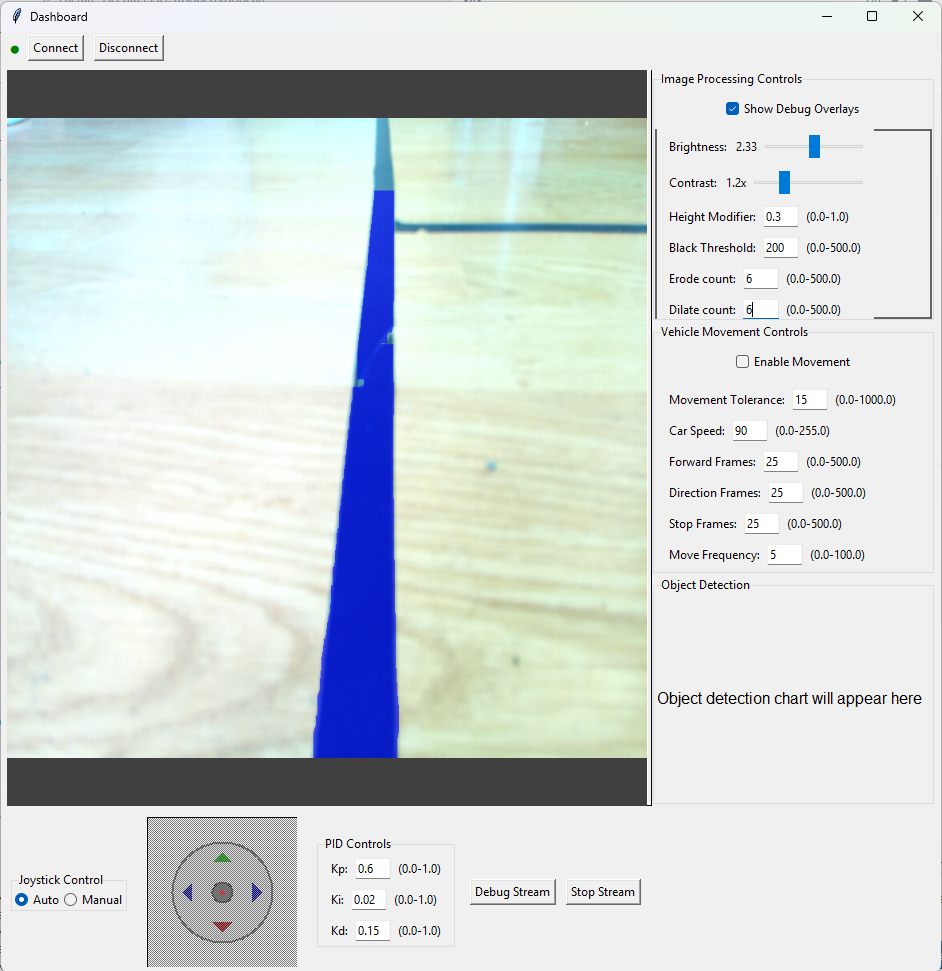
* **Activare:** Utilizatorul selectează opțiunea "Manual" din panoul "Joystick Control"
* **Control:** Prin mișcarea cursorului în interiorul joystick-ului virtual, utilizatorul poate comanda direcția (înainte, înapoi, stânga, dreapta) și viteza vehiculului. Aplicația traduce poziția cursorului în comenzi de viteză și direcție, care sunt trimise în timp real către Arduino pentru a controla motoarele

## Modul automat de funcționare

Modul automat permite vehicului să navigheze autonom.

Odată cu acest mod activat, vehiculul începe să navigheze autonom, urmărind linia neagră de pe traseu și reacționând la indicatoarele rutiere detectate.

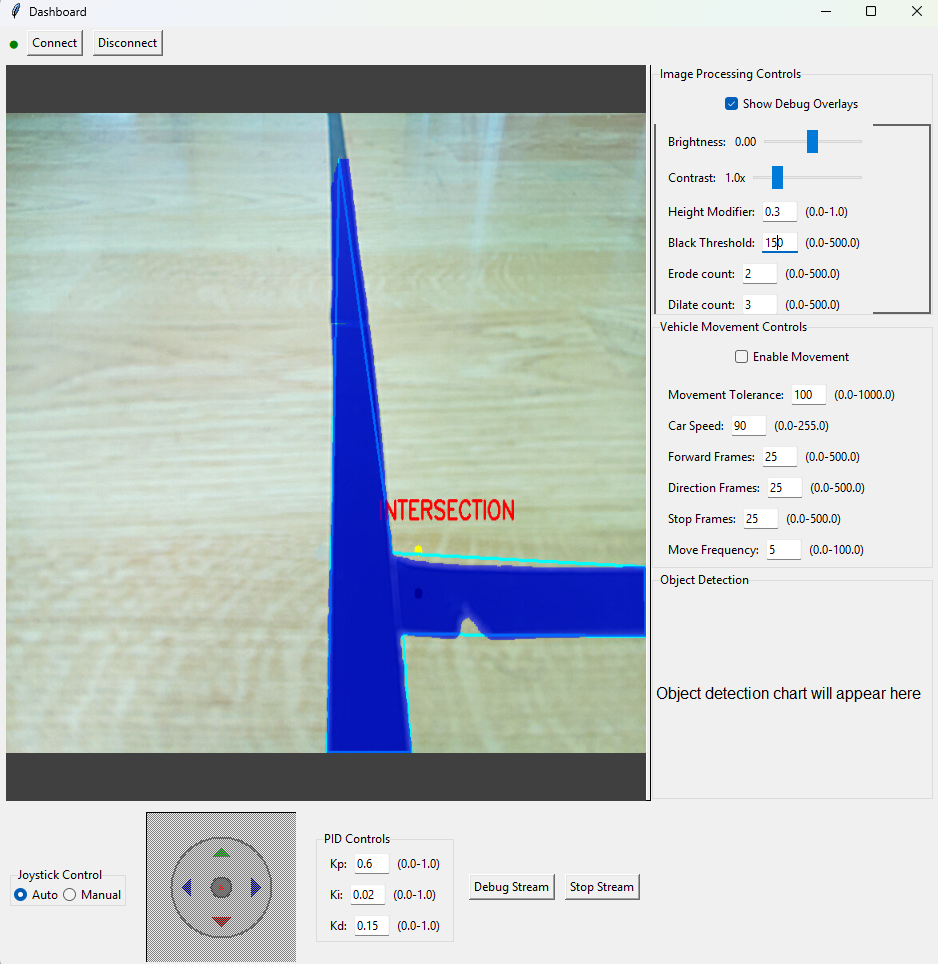
* Urmărirea liniei: Sistemul procesează imaginile pentru a identifica linia neagră, aplicând tehnici de filtrare, segmentare și operații morfologice pentru a obține un contur clar al traseului. Pe baza poziției liniei în cadru, un algoritm de control PID calculează și ajustează continuu viteza motoarelor pentru a menține vehiculul centrat pe traseu. Imaginea arată ajustarea unor parametri precum luminozitatea (6.0), contrastul (1.2x) și numărul de operații de erodare (5) și dilatare (7) pentru o detecție optimă.



***Figura 4.3: Parametri imaginii modificați***

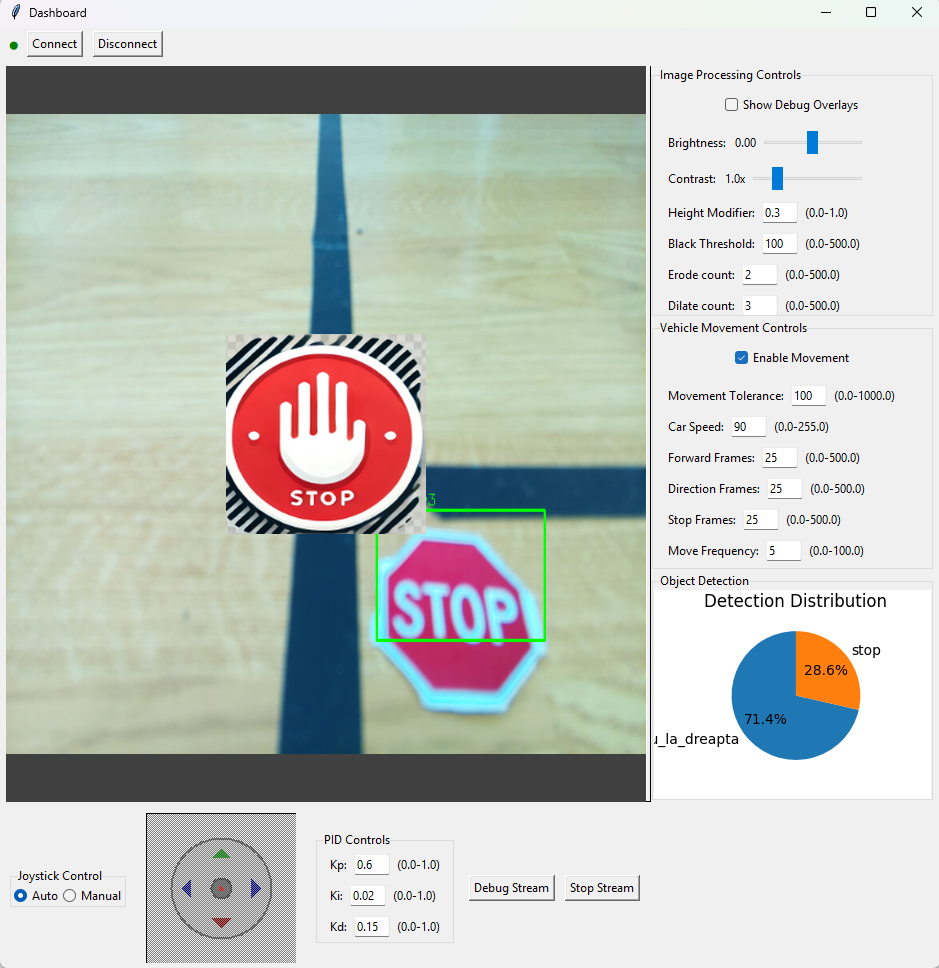
* Detecția intersecțiilor: Atunci când algoritmul de aproximare a contururilor identifică o formă geometrică complexă (cu mai mult de 6 vârfuri), sistemul recunoaște o intersecție și afișează textul "INTERSECTION" pe fluxul video.

În acest moment, vehiculul încetinește și așteaptă detecția unui indicator rutier pentru a decide următoarea acțiune.

****

***Figura 4.4: Intersecție detectată***

* Recunoașterea indicatoarelor rutiere: Când vehiculul se apropie de o intersecție, modelul de inteligență artificială analizează fluxul video pentru a detecta semne de circulație. De exemplu, dacă este detectat un indicator "la dreapta" cu o încredere de 95%, informația este afișată pe ecran ("la\_dreapta 95%"). Un grafic de tip pie-chart poate arăta distribuția obiectelor detectate, confirmând că 100% din detecții corespund indicatorului "La Dreapta".



***Figura 4.5: Comportament în intersecție***

Luarea deciziei în intersecție: Odată ce un indicator este recunoscut, sistemul inițiază o secvență de acțiuni predefinite:

* Vehiculul avansează un număr specific de cadre pentru a se poziționa corect în intersecție.
* Execută virajul corespunzător (stânga sau dreapta) pentru un anumit număr de cadre.
* După finalizarea virajului, sistemul intră într-o buclă de realiniere, căutând din nou linia neagră pentru a continua deplasarea pe noul segment de traseu.

Aceste scenarii demonstrează capacitatea sistemului de a integra percepția vizuală avansată cu algoritmi de control preciși pentru a naviga autonom într-un mediu structurat, luând decizii bazate pe interpretarea în timp real a indicatoarelor rutiere.

# Capitolul 5: Direcții viitoare

Această lucrare a prezentat proiectarea, implementarea și testarea unui sistem embedded pentru navigație autonomă, bazat pe recunoașterea obiectelor în timp real. Proiectul a demonstrat cu succes integrarea unei platforme hardware accesibile, precum Raspberry Pi 5, cu un accelerator de inteligență artificială (Hailo-8L), pentru a rula modele complexe precum YOLOv8 în aplicații care necesită latență redusă.

Sistemul a dovedit capacitatea de a urmări o linie, de a detecta intersecții și de a lua decizii de navigație pe baza indicatoarelor recunoscute, validând astfel fezabilitatea arhitecturii propuse pentru sarcini de navigație în medii controlate.

## Îmbunătățirea percepției și a interacțiunii

Dezvoltarea viitoare a sistemului va include extinderea capabilităților de percepție și interacțiune, transformând vehiculul într-o platformă mai robustă și mai inteligentă.

Prima prioritate va fi integrarea unui senzor LiDAR, care este esențial pentru implementarea algoritmilor de cartografiere și localizare simultană (SLAM). Această dezvoltare majoră va permite vehiculului să construiască hărți digitale ale mediului și să navigheze eficient pe trasee deja parcurse, reducând timpul de procesare.

De asemenea, o direcție importantă este îmbunătățirea interacțiunii om-mașină prin adăugarea unui modul de sunet. Sistemul va putea oferi feedback vocal utilizatorului despre acțiunile pe care le execută (de exemplu, "Virez la dreapta"), crescând transparența deciziilor. Această funcționalitate, combinată cu analiza statisticilor generate de graficele din dashboard, va permite sistemului să înțeleagă mai profund mediul înconjurător și să comunice mai eficient cu utilizatorul.

## Migrarea către o platformă mobilă și modernizarea hardware

Tranziția către o arhitectură de control mobilă și modernizarea componentelor fizice reprezintă următorul pas major în evoluția proiectului. Dezvoltarea unei aplicații mobile dedicate pentru Android sau iOS va înlocui actualul dashboard, oferind o interfață de control mult mai flexibilă, accesibilă și intuitivă, permițând monitorizarea și gestionarea sistemului direct de pe un smartphone.

Această migrare va deschide calea către un management al sistemului independent de locație, eliminând necesitatea unui server local și a unei conexiuni directe la rețeaua locală.

În paralel, se va urmări îmbunătățirea platformei fizice. Proiectarea și imprimarea 3D a unui șasiu mai rezistent, din materiale precum PETG sau ABS, va crește durabilitatea vehiculului. Un șasiu modernizat va putea susține motoare mai puternice și roți mai mari, extinzând capabilitățile de navigație ale vehiculului pe suprafețe mai variate și permițând integrarea de senzori suplimentari fără a compromite performanța.

# Concluzii

Lucrarea de față a prezentat proiectarea, implementarea și testarea unui sistem embedded complet pentru navigație autonomă, axat pe recunoașterea obiectelor în timp real. Obiectivul principal, acela de a dezvolta o soluție autonomă sigură și eficientă la un preț cât mai accesibil, a fost atins cu succes prin integrarea strategică a unor tehnologii hardware și software modern.

Contribuția originală a proiectului constă în demonstrarea practică a fezabilității rulării unor algoritmi de deep learning complecși, precum YOLOv8, pe o platformă cu resurse limitate. Acest lucru a fost posibil prin combinarea unui single-board computer (Raspberry Pi 5) cu un accelerator hardware dedicat (Hailo-8L), obținându-se astfel performanțe de procesare în timp real care altfel ar fi fost imposibile pe un astfel de sistem. Prin această arhitectură hibridă, s-a reușit dezvoltarea unui vehicul capabil să navigheze autonom urmărind o linie, să identifice intersecții și să ia decizii pe baza indicatoarelor rutiere recunoscute, totul într-un buget redus.

S-a urmat o metodologie riguroasă de proiectare, implementare și testare, care a cuprins toate etapele, de la asamblarea componentelor hardware și antrenarea modelului de inteligență artificială, până la dezvoltarea logicii de control PID pe un microcontroler Arduino și crearea unui dashboard interactiv pentru monitorizare.

Alegerea tehnologiilor, precum Python pentru dezvoltarea software, GStreamer pentru managementul fluxului video și Flask pentru comunicare, a reflectat o abordare echilibrată între performanță, modularitate și eficiență.

În final, proiectul nu doar că validează o soluție tehnică funcțională pentru navigația autonomă la scară redusă, dar servește și ca o contribuție educațională, integrând cunoștințe din domenii diverse precum procesarea imaginilor, inteligența artificială și sistemele embedded într-un demers practic și coerent.

Rezultatele obținute deschid calea către viitoare dezvoltări și optimizări, confirmând potențialul sistemelor embedded accelerate hardware în democratizarea tehnologiilor autonome.

# Bibliografie

1. An overview of autonomous navigation, <https://www.vaia.com/en-us/explanations/engineering/artificial-intelligence-engineering/autonomous-navigation/> (data accesării: 17:05:2025)
2. Waymo explores using Google’s Gemini to train its robotaxis,Andrew J. Hawkins , <https://www.theverge.com/2024/10/30/24283516/waymo-google-gemini-llm-ai-robotaxi> (data accesării : 18:05:2025)
3. Tesla robotaxi Elon Musk claims safety driverless level 5, Andrew J.Hawkins, <https://www.theverge.com/2024/10/9/24265781/tesla-robotaxi-elon-musk-claims-safety-driverless-level-5> (data accesării:18:05:2025)
4. Dasaradharami Reddy K. , Anusha S.and Ashalatha N., Recent Advances in Autonomous Navigation for Robots- A Comprehensive Review, Nobel Science Publisher, <https://nspublisher.com/wp-content/uploads/2024/01/NSPEEE-23-RW-101.pdf> (data accesării: 18:05:2025)
5. Thomas Ryan, Autonomous Navigation Report:Analysis on the Market, Trends, and Technologies, trendfedr.com, <https://trendfeedr.com/reports/autonomous-navigation-report/> (data accesării:18:05:2025)
6. Saeid Nahavandi, Roohallah Alizadehsani, Darius Nahavandi, Shady Mohamed, Navid Mohajer, Mohammad Rokonuzzaman, Ibrahim Hossain, A Comprehensive Review on Autonomous Navigation, arxiv.org, <https://arxiv.org/pdf/2212.12808> (data accesării: 18:05:2025)
7. Zumo Robot for Android, Adafruit.com, <https://www.adafruit.com/product/1639> (data accesării:18:05:2025)
8. Donkey Car S1, robocarstore.com, <https://www.robocarstore.com/products/donkey-car-starter-kit> (data accesării:18:05:2025)
9. JetRacer, waveshare.com, <https://www.waveshare.com/wiki/JetRacer_AI_Kit> (data accesării:18:05:2025)
10. Opencv, <https://coseries.com/python-opencv/> (data accesării: 18:05:2025)
11. Image Rotation and Transformation,OpenCv.org, <http://opencv.org/blog/image-rotation-and-translation-using-opencv/> (data accesării 18:05:2025)
12. Python OpenCV - getRotationMatrix2D() Function, GeeksForGeeks.org, <https://www.geeksforgeeks.org/python-opencv-getrotationmatrix2d-function/> (data accesării: 18:05:2025)
13. OpenCV-Rotate Image,Adrian Rosebrock, pyimagesearch,com, <https://pyimagesearch.com/2021/01/20/opencv-rotate-image/> (data accesării: 18:05:2025)
14. Adrian Rosebrock ,OpenCV Smoothing and Blurring, <https://pyimagesearch.com/2021/04/28/opencv-smoothing-and-blurring/> (data accesării: 18:05:2025)
15. Smoothing Images, docs.opencv.org, <https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html> (data accesării: 18:05:2025)
16. Kernel(Image Processing), wikipedia.org, <https://en.wikipedia.org/wiki/Kernel_(image_processing)> (data accesării: 18:05:2025)
17. Sharpen Image, medium.com, <https://medium.com/@colombia202324/opencv-sharpen-image-pycharm-b09558e9ade1> (data accesării: 18:05:2025)
18. Thresholding-Based Image Segmentation, geeksforgeeks.org, <https://www.geeksforgeeks.org/thresholding-based-image-segmentation/> (data accesării: 18:05:2025)
19. Detecting lines in opencv, stackoverflow.com, <https://stackoverflow.com/questions/45322630/how-to-detect-lines-in-opencv> (data accesării: 18:05:2025)
20. Image Erosion, wikipedia.org, <https://en.wikipedia.org/wiki/Erosion_(morphology)> (data accesării: 18:05:2025)
21. Morphological operations, roboflow.com, <https://blog.roboflow.com/morphological-operations/> (data accesării: 18:05:2025)
22. Image Dilation, wikipedia.org, <https://en.wikipedia.org/wiki/Dilation_(morphology)> (data accesării: 18:05:2025)
23. TAYMANS, Wim, et al. Gstreamer application development manual (1.2. 3). *Publicado en la Web*, 2013, 72., <https://www.academia.edu/download/44962524/Gstreamer_Application_Development_Manual_1.2.3.pdf> (data accesării: 19:05:2025)
24. Secure Shell, wikipedia.org, <https://en.wikipedia.org/wiki/Secure_Shell> (data accesării: 19:05:2025)
25. Flask Server Official Documentation, <https://flask.palletsprojects.com/en/stable/> (data accesării: 19:05:2025)
26. John Long,What is WSGI(Web Server Gateway Interface), liquidweb.com, <https://www.liquidweb.com/blog/what-is-wsgi/> (data accesării:19:05:2025)
27. Inteligența artificială, wikipedia.org, <https://ro.wikipedia.org/wiki/Inteligen%C8%9B%C4%83_artificial%C4%83> (data accesării: 19:05:2025)
28. Convolutional Neural Network, developersbreach.com, <https://developersbreach.com/convolution-neural-network-deep-learning> (data accesării: 19:05:2025)
29. Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi, You Only Look Once: Unified, Real-Time Object Detection, <https://arxiv.org/pdf/1506.02640> (data accesării:19:05:2025)
30. Juan R. Terven & Diana M. Cordova-Esparza, A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS, <https://arxiv.org/html/2304.00501v6> (data accesării: 19:05:2025)
31. Raspberry Pi – Product overview, <https://www.raspberrypi.com/products/raspberry-pi-5/> (data accesării: 20:05:2025)
32. Raspberry Pi Camera Module 2.1 – Product overview, <https://www.raspberrypi.com/products/camera-module-v2/> (data accesării: 20:05:2025)
33. Hailo-8L – Product overview, <https://hailo.ai/products/ai-accelerators/hailo-8l-m-2-ai-acceleration-module-for-ai-light-applications/#hailo8lm2-overview> (data accesării:20:05:2025)
34. Arduino Uno R3 – Product overview, <https://store.arduino.cc/products/arduino-uno-rev3> (data accesării: 20:05:2025)
35. UART Communication Protocol and How It Works, seeedstudio.com, <https://www.seeedstudio.com/blog/2022/09/08/uart-communication-protocol-and-how-it-works/> (data accesării: 20:05:2025)
36. Transmisie de date serială, wikipedia.org, <https://ro.wikipedia.org/wiki/Transmisie_de_date_serial%C4%83> (data accesării: 20:05:2025)
37. G. Lucas, Using a PID-based Technique For Competitive Odometry and Dead-Reckoning, <http://www.seattlerobotics.org/encoder/200108/using_a_pid.html> (data accesării: 20:05:2025)
38. Raspberry Pi AI Kit – Custom YOLOV8 Object Detection, <https://www.youtube.com/watch?v=7pgSFgqo8gY&t=1s> (data accesării: 20:05:2025)
39. Set de date semne de circulatie, <https://app.roboflow.com/daniel-bsc7p/detectie-semne-de-circulatie/11> (data accesării: 20:05:2025)
40. Arduino Robot Car Chasis Assembly, <https://www.youtube.com/watch?v=WSMFLkL-niY&list=PLO4bQCKgdOzvSxTjD0aHRIdx1ePybwxyn&index=2>, data accesării: 20:05:2025)
41. How to control DC motor with L298N driver and Arduino, <https://www.youtube.com/watch?v=dyZolgNOomk&list=PLO4bQCKgdOzvSxTjD0aHRIdx1ePybwxyn&index=4> , (data accesării: 20:05:2025)
42. L298N Dual H-Bridge Motor Driver,handsontec.com, <https://www.handsontec.com/dataspecs/L298N%20Motor%20Driver.pdf> (data accesării:20:05:2025)