

UNIVERSITATEA POLITEHNICĂ DIN BUCUREȘTI  
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE  
DEPARTAMENTUL DE CALCULATOARE



## PROIECT DE DIPLOMĂ

Arhitectură bazată pe microservicii pentru navigație vizuală și control coordonat al vehiculelor RC folosind markeri ArUco

Rădulescu Matei

**Coordonator științific:**

Șl. Dr. Ing. Jan-Alexandru Văduvă

**BUCUREȘTI**

2025

# CUPRINS

<b>1</b>	<b>Introducere</b>	<b>1</b>
1.1	Context . . . . .	1
1.2	Problema . . . . .	1
1.3	Obiective . . . . .	2
1.4	Soluția propusă . . . . .	3
1.5	Rezultatele obținute . . . . .	3
1.6	Structura lucrării . . . . .	4
<b>2</b>	<b>Motivație</b>	<b>6</b>
<b>3</b>	<b>Metode Existente</b>	<b>7</b>
3.1	Technologii alese . . . . .	7
3.1.1	Procesare centralizată vs procesare descentralizată în sisteme multi-robot	7
3.1.2	Tehnici de urmărire bazate pe markeri vizuali . . . . .	7
3.1.3	Arhitecturi bazate pe containere pentru aplicații cu roboți . . . . .	8
3.2	Alte metode de localizare . . . . .	9
3.2.1	Metode de localizare bazate pe LiDAR . . . . .	9
3.2.2	Metode de localizare bazate pe GPS . . . . .	9
3.2.3	Compararea metodelor . . . . .	9
<b>4</b>	<b>Soluția Propusă</b>	<b>11</b>
<b>5</b>	<b>Detalii de implementare</b>	<b>13</b>
5.1	Technologii . . . . .	13
5.1.1	Flutter . . . . .	13
5.1.2	Markeri ArUco . . . . .	13
5.1.3	Server: Python folosind Flask . . . . .	14

5.1.4	Redis . . . . .	15
5.1.5	Docker și Kubernetes . . . . .	15
5.2	Aplicație mobilă . . . . .	15
5.2.1	Parametrii cameră . . . . .	15
5.2.2	Interfață . . . . .	16
5.3	Server . . . . .	18
5.3.1	Functionare API-uri . . . . .	19
5.3.2	Containerizare și orchestrare . . . . .	23
<b>6</b>	<b>Evaluare</b>	<b>25</b>
6.1	Generare Imagini . . . . .	25
6.2	Testare eroare . . . . .	26
6.3	Testare viteză de procesare imagini . . . . .	28
6.4	Testare scalare cu numărul de vehicule . . . . .	29
6.5	Testare viteză transmitere imagini . . . . .	31
6.6	Testare terminare traseu . . . . .	32
6.6.1	Testare cu mai multe vehicule . . . . .	33
6.6.2	Testare terminare traseu aplicație . . . . .	34
<b>7</b>	<b>Concluzii</b>	<b>35</b>
	<b>Bibliografie</b>	<b>36</b>

## SINOPSIS

Controlul rapid și precis al mai multor vehicule teleghidate (RC) permite automatizarea mai multor unități mobile autonome prin asigurarea evitării coliziunilor, precum și a poziționării precise, permițându-le să ajungă la locațiile unde trebuie să își îndeplinească sarcinile.

Această lucrare prezintă un model care analizează imagini folosind o aplicație mobilă Flutter ce trimite cadrele camerei către un server de detecție implementat pe Kubernetes, care utilizează un Horizontal Pod Autoscaler pentru a asigura disponibilitate înaltă și scalabilitate. Serverul detectează markeri statici utilizați pentru definirea traseului pe care vehiculele trebuie să îl urmeze, precum și markeri atașați în partea lor frontală și posterioară pentru o poziționare precisă în lumea reală, bazată pe centrul fotografiilor și pe dimensiunea markerilor detectați. Vehiculele RC interoghează periodic serverul pentru a prelua propriile locații, locațiile celorlalte vehicule și pozițiile markerilor folosiți ca puncte de referință pentru traseu. Acestea sunt, de asemenea, notificate când finalizează traseul definit, primind coordonate false prestabilite pentru locațiile proprii, dacă este cazul.

Această logică bazată pe microservicii, cu o percepție centralizată care alimentează un proces decizional descentralizat, oferă feedback cu latență redusă, navigație metrică precisă și scalare ușoară pentru flote mai mari, demonstrând o fundație robustă pentru sistemele autonome coordonate.

## ABSTRACT

Fast and accurate control of multiple RC vehicles enables automation of multiple autonomous mobile units by ensuring collision avoidance as well as accurate positioning, allowing them to reach the locations where they need to perform their tasks.

This paper presents a model that analyzes images by using a Flutter mobile app that sends camera frames to a Kubernetes-deployed detection server, which uses a Horizontal Pod Autoscaler to ensure high availability and scalability. The server detects static markers used for defining the route that the vehicles have to follow, as well as markers attached to their front and rear for accurate real-world positioning based on the center of the photos and the size of the markers detected. RC vehicles periodically query the server to retrieve their own locations, the locations of the other vehicles, and the positions of the markers used as waypoints. They are also notified when they finish the defined route by receiving pre-known fake coordinates for their own locations, if applicable.

This microservice-based logic, with centralized perception feeding decentralized decision-making, delivers low-latency feedback, precise metric navigation, and easy scaling for larger fleets, demonstrating a robust foundation for coordinated autonomous systems.

# 1 INTRODUCERE

## 1.1 Context

Automatizarea vehiculelor autonome la scară mică, precum cele RC, este tot mai relevantă în domenii precum logistica interioară, transportul autonom experimental și testarea algoritmilor de navigație. În aplicațiile ce au acest scop, localizarea precisă a vehiculelor într-un timp cât mai scurt este esențială pentru evitarea coliziunilor și pentru coordonarea eficientă a traseului vehiculelor.

Soluțiile de localizare vizuală bazate pe markere, precum cele de tip ArUco, s-au dovedit a fi o alternativă eficientă și accesibilă la metodele tradiționale bazate pe GPS, LiDAR sau senzori montați pe vehicul. Aceste marcaje permit determinarea poziției și orientării obiectelor în spațiu folosind doar imagini capturate de o cameră standard, fără a necesita alte componente hardware. Utilizarea markerilor ArUco permite obținerea unei poziționări precise în sisteme cu constrângeri de buget și spațiu, fiind deja folosită în multiple proiecte de detecție și urmărire vizuală în timp real [4].

Deși tehnologiile LiDAR sunt frecvent utilizate în sisteme avansate de percepție 3D, acestea implică costuri ridicate și complexitate suplimentară. În comparație, utilizarea markerilor vizuali aduce o soluție mult mai simplă de implementat, cu acuratețe suficientă pentru aplicații de interior și medii controlate [6], [7].

Pe partea software, arhitecturile moderne tind tot mai mult spre modele distribuite, bazate pe microservicii, care permit scalarea dinamică și gestionarea eficientă a resurselor în sisteme autonome. Utilizarea unor platforme precum Kubernetes facilitează implementarea unor astfel de soluții, oferind viteze sporite, flexibilitate și posibilitatea de a adapta sistemul în funcție de volumul de date procesate. Soluțiile containerizate pot rula în paralel și pot gestiona eficient cereri simultane de la vehicule sau surse de imagini [5].

## 1.2 Problema

Coordonarea mai multor vehicule autonome într-un spațiu comun ridică provocări legate de localizarea precisă, evitarea coliziunilor și comunicarea eficientă într-un timp cât mai scurt. Soluțiile clasice implică montarea de senzori pe fiecare vehicul, ceea ce crește atât complexitatea sistemului, cât și costurile. În cazul vehiculelor RC, unde spațiul și resursele hardware sunt limitate, este necesară o alternativă mai simplă, dar la fel de precisă.

O problemă majoră constă în obținerea poziției fiecărui vehicul cu o eroare cât mai mică, doar pe baza imaginilor capturate de o cameră externă, fără a afecta precizia sau timpul de reacție. Astfel, procesarea imaginilor trebuie să fie suficient de rapidă pentru a ține pasul cu viteza maximă a vehiculelor, astfel încât deciziile luate pe baza pozițiilor să nu devină inexacte. În unele cazuri, o întârziere de doar câteva sute de milisecunde poate însemna o schimbare semnificativă a poziției reale, mai ales în scenarii dinamice.

În plus, sistemul trebuie să fie capabil să gestioneze simultan mai multe vehicule, să răspundă în timp real la cererile acestora și să fie ușor de extins fără modificări majore ale arhitecturii. Din punct de vedere software, acest lucru implică o arhitectură scalabilă și containerizabilă, capabilă să proceseze paralel imagini și să răspundă eficient cererilor REST. Prin utilizarea unei infrastructuri orchestrate, cum este Kubernetes, se poate asigura scalarea automată a componentelor de procesare în funcție de sarcina activă, menținând timpi de răspuns reduși și comportament determinist în scenarii cu trafic intens [5].

Aceste cerințe definesc o problemă complexă, care necesită o abordare eficientă din punct de vedere al percepției vizuale, procesării asincrone și comunicării între server și vehicule. Soluția propusă în această lucrare răspunde acestor nevoi printr-un sistem complet containerizat, alimentat de imagini trimise de pe un telefon mobil, care identifică pozițiile vehiculelor folosind markerii ArUco și coordonează deplasarea acestora fără a necesita hardware suplimentar montat pe fiecare mașină.

### 1.3 Obiective

Scopul principal al acestei lucrări este dezvoltarea unei soluții de localizare vizuală în timp real pentru vehicule RC, utilizând markerii ArUco, într-un sistem scalabil bazat pe microservicii.

În Tabela 1 sunt prezentate obiectivele principale ale acestei lucrări și modul în care acestea au fost realizate.

Tabela 1: Obiectivele sistemului și metodele de realizare

Obiectiv	Mod de realizare
Captarea și trimiterea imaginilor	Aplicație Flutter care capturează imagini și le transmite cât mai frecvent posibil către un server de procesare.
Procesarea imaginilor și detecția markerilor	Server containerizat care identifică markerii ArUco și calculează pozițiile vehiculelor în coordonate metrice bidimensionale. Înălțimea este ignorată pentru calculul distanței dintre markeri, eliminând astfel această sursă de eroare.
Scalabilitate și disponibilitate	Serverul rulează într-o arhitectură Kubernetes cu scalare automată folosind Horizontal Pod Autoscaler.

Obiectiv	Mod de realizare
Transmiterea datelor către vehicule	Serverul trimite fiecărui vehicul poziția proprie, pozițiile celorlalte vehicule și ale markerilor statici, interpretate într-un spațiu bidimensional, așa cum este văzut de cameră.
Detectarea finalizării traseului	Serverul verifică dacă vehiculul a trecut prin toți markerii de traseu și returnează o notificare corespunzătoare la următoarea solicitare.
Răspuns rapid la mișcare	Prin împărțirea sarcinilor de procesare între mai multe poduri Kubernetes, imaginile pot fi analizate în paralel, menținând o frecvență ridicată de răspuns chiar și în cazul unui număr crescut de vehicule.

## 1.4 Soluția propusă

Soluția propusă constă într-un sistem de localizare vizuală pentru vehicule RC, bazat pe detectarea markerilor ArUco din imagini capturate periodic de o cameră externă. Scopul sistemului este de a furniza vehiculelor poziția proprie și pozițiile celorlalți participanți într-un mod suficient de rapid pentru a permite deplasarea coordonată într-un spațiu comun.

Imaginile sunt preluate prin intermediul unei aplicații mobile dezvoltate în Flutter, rulată pe un telefon poziționat astfel încât să fie vizibil întreg traseul, precum și vehiculele. Acestea sunt transmise periodic către un server care detectează markerii și calculează pozițiile acestora în coordonate metrice.

Serverul este containerizat și rulează într-o arhitectură Kubernetes, folosind scalare și load balancing automate cu ajutorul unui Horizontal Pod Autoscaler. Această structură permite procesarea paralelă a cadrelor, contribuind la un timp de răspuns mic, chiar și în prezența mai multor vehicule.

Sistemul oferă un API HTTP prin care fiecare vehicul poate accesa poziția sa actuală, pozițiile celorlalte vehicule și ale markerilor statici. De asemenea, serverul urmărește progresul fiecărui vehicul pe traseu și îl anunță atunci când a parcurs toți markerii necesari.

## 1.5 Rezultatele obținute

Rezultatele obținute în urma testării implementării propuse confirmă funcționalitatea și eficiența arhitecturii dezvoltate, atât în medii controlate, cât și în condiții apropiate de utilizarea reală. Sistemul a fost evaluat din mai multe perspective: precizia poziționării, viteza de procesare a imaginilor, comportamentul la scalare, toleranța la trimitere rapidă de imagini și capacitatea de a detecta corect încheierea traseului de către vehicule.

- Precizia poziționării a fost testată folosind 125 de imagini în care două markere erau plasate la o distanță cunoscută de 40 cm. Sistemul a estimat distanța dintre ele cu o eroare medie de aproximativ 1.05 mm. În cele mai multe cazuri, eroarea a fost sub 1 mm, iar deviațiile mai mari (outliere) au fost rare și pot fi reduse prin creșterea frecvenței de captură.
- Viteza de procesare a imaginilor variază în funcție de metoda de rulare și de frecvența transmisiilor. În cazul rulării locale, timpul de răspuns mediu pentru o imagine a fost de aproximativ 2.2 secunde. La rularea în Kubernetes, cu autoscalare activă, timpii au scăzut semnificativ, până la aproximativ 0.1 secunde în scenarii cu sarcină moderată. În scenarii de stres, cu imagini trimise foarte rapid, timpul de răspuns a crescut spre 0.35 secunde, dar fără blocaje sau pierderi semnificative de performanță.
- Scalabilitatea sistemului a fost testată trimițând imagini conținând între 1 și 10 vehicule, fiecare vehicul trimițând în paralel cereri către server pentru obținerea poziției. Rularea locală a avut timpi de răspuns constanți și mari, în timp ce execuția în Kubernetes a permis menținerea unor timpi de răspuns stabili și mici, chiar și în scenarii cu sarcină ridicată. Graficul timpilor medii per vehicul a arătat că aceștia rămân foarte apropiați, ceea ce demonstrează că nu apare nicio favorizare accidentală între vehicule.
- Toleranța la frecvențe ridicate de trimitere a fost analizată prin trimiterea de imagini la intervale tot mai mici (până la 41 ms). Rularea fără containerizare a produs întârzieri semnificative, în timp ce execuția orchestrată cu autoscalare a permis procesarea paralelă. Chiar dacă s-a atins limita de procesare a unei imagini individuale, distribuirea automată a sarcinii pe mai multe poduri a menținut un răspuns stabil al sistemului.
- Detectarea finalizării traseului a fost validată în două moduri: cu imagini generate artificial și cu sesiuni reale în aplicația mobilă. În ambele cazuri, sistemul a identificat corect vehiculele care au parcurs toți markerii de traseu. Aplicația a afișat timpii corect ordonați, iar vehiculele care nu au terminat au fost marcate cu timpul -1 și plasate automat la final.

Rezultatele obținute confirmă atingerea obiectivelor formulate în introducerea lucrării. S-a realizat o soluție precisă, care funcționează în timp real, capabilă să scaleze dinamic în funcție de numărul de vehicule și de frecvența transmiterii imaginilor, oferind în același timp o interfață intuitivă prin aplicația mobilă. Integrarea cu o infrastructură containerizată și orchestrată asigură portabilitate, robustețe și adaptabilitate în scenarii diverse de utilizare.

## 1.6 Structura lucrării

Lucrarea este organizată în 7 capitole, fiecare având un rol bine definit în prezentarea și validarea soluției propuse:

- Capitolul 1 Introducere: prezintă contextul general al lucrării, motivația alegerii acestei teme, obiectivele propuse, rezultatele obținute și structura generală a lucrării.



- Capitolul 2 Context și formularea problemei: oferă fundalul tehnologic și teoretic necesar înțelegerii proiectului, discutând despre localizarea vizuală, sistemele autonome, avantajele markerilor ArUco și provocările specifice coordonării vehiculelor RC.
- Capitolul 3 Metode existente: analizează metodele deja existente în literatură pentru localizarea vehiculelor și coordonarea acestora, inclusiv soluții bazate pe centralizare, descentralizare, GPS sau LiDAR.
- Capitolul 4 Detalii de implementare: descrie în detaliu modul de realizare a aplicației mobile cu Flutter, a serverului de procesare cu Flask, comunicarea prin Redis, detectarea markerilor ArUco și containerizarea sistemului cu Docker și Kubernetes.
- Capitolul 5 Funcționarea sistemului: explică API-urile implementate, logica sesiunii de coordonare, modul de procesare a imaginilor și comportamentul serverului în raport cu vehiculele.
- Capitolul 6 Evaluare: prezintă metodologia de testare și rezultatele obținute, acoperind aspecte precum: generarea imaginilor, eroarea de poziționare, viteza de procesare, scalarea în funcție de numărul de vehicule și frecvența imaginilor, precum și corectitudinea detectării terminării traseului.
- Capitolul 7 Concluzii: sintetizează realizările lucrării, evidențiază contribuțiile majore, discută limitările actuale și direcțiile posibile de extindere.

## 2 MOTIVAȚIE

Odată cu creșterea interesului pentru dezvoltarea de sisteme autonome accesibile, a apărut tot mai clar nevoia unor metode simple, reproductibile și eficiente pentru testarea algoritmilor de localizare și coordonare. În special în domenii precum cercetarea academică, prototiparea rapidă sau educația practică în robotică, este important ca soluțiile propuse să nu necesite echipamente costisitoare sau configurații greu de reprodus.

Vehiculele RC reprezintă o opțiune foarte potrivită pentru explorarea acestor concepte. Ele oferă o platformă fizică realistă, dar în același timp suficient de accesibilă și flexibilă pentru a simula comportamentul unor sisteme autonome la scară redusă. Datorită dimensiunilor mici și costurilor reduse, aceste vehicule pot fi folosite în laboratoare, în proiecte educaționale sau în medii de testare controlate.

Totuși, chiar și într-un astfel de context simplificat, coordonarea mai multor vehicule într-un spațiu comun implică provocări concrete, în special în ceea ce privește localizarea acestora și evitarea coliziunilor. În multe cazuri, soluțiile existente presupun montarea unor senzori suplimentari pe fiecare vehicul, ceea ce poate complica și îngreuna procesul de implementare. În plus, în cazul vehiculelor de mici dimensiuni, spațiul fizic disponibil pentru montarea unor astfel de senzori este adesea limitat.

Lucrarea de față este motivată de nevoia de a elimina aceste constrângeri printr-o soluție mai flexibilă: utilizarea percepției vizuale externe, obținută cu o simplă cameră video, pentru a urmări și coordona poziția vehiculelor. Această abordare are avantajul că nu necesită modificări fizice ale vehiculelor și permite scalarea sistemului la mai multe unități fără o creștere direct proporțională a complexității tehnice.

În același timp, implementarea unei arhitecturi moderne, bazate pe microservicii, permite distribuirea sarcinilor de procesare și adaptarea sistemului la condiții variabile, aspecte importante atât din perspectivă tehnică, cât și din cea didactică. Astfel, această lucrare își propune să contribuie la dezvoltarea unei metode practice, accesibile și ușor de extins pentru testarea și experimentarea algoritmilor de coordonare în sisteme autonome.

## **3 METODE EXISTENTE**

### **3.1 Tehnologii alese**

#### **3.1.1 Procesare centralizată vs procesare descentralizată în sisteme multi-robot**

Procesarea centralizată reprezintă o abordare clasică în sistemele multi-robot, în care o entitate centrală este responsabilă cu colectarea datelor de la toți agenții și luarea deciziilor pentru întregul sistem. Această strategie permite o viziune globală asupra mediului și o coordonare coerentă între roboți, fiind utilă în special în sarcini de acoperire, planificare optimă sau sincronizare a acțiunilor.

Un avantaj major al procesării centralizate este faptul că permite optimizarea deciziilor la nivel de întreg sistem, bazându-se pe informații agregate din toate sursele disponibile. Acest lucru este evidențiat în articolul[1], unde sunt analizate performanțele sistemelor centralizate în sarcini de acoperire multi-robot. Autorii arată că, în contexte controlate, o astfel de abordare poate duce la strategii eficiente și la un nivel ridicat de acuratețe în coordonare. Totuși, aceeași lucrare subliniază și limitările acestei metode: o arhitectură centralizată devine vulnerabilă la pierderi de comunicație, nu scalează bine odată cu numărul de agenți și depinde critic de funcționarea continuă a nodului central.

O abordare alternativă este propusă în articolul[2], unde este descris un model hibrid între centralizare și descentralizare. În acest caz, fiecare robot ia decizii locale, dar există și o unitate centrală care monitorizează progresul întregului grup și poate distribui sarcinile în mod echilibrat. Această combinație aduce mai multă flexibilitate, păstrând în același timp avantajele unei percepții globale.

În cadrul lucrării de față, procesarea centralizată este utilizată pentru extragerea pozițiilor vehiculelor dintr-un flux de imagini și pentru transmiterea acestor informații către fiecare vehicul printr-un API. Această abordare permite o percepție comună asupra spațiului, fără a impune un control direct asupra comportamentului individual al vehiculelor, lăsând deciziile la nivelul acestora.

#### **3.1.2 Tehnici de urmărire bazate pe markeri vizuali**

Urmărirea obiectelor în spațiu cu ajutorul markerilor vizuali este o metodă bine documentată și des utilizată în diverse aplicații din domeniul roboticii, analizei mișcării și viziunii compute-

rizate. Această tehnică presupune atașarea unor marcaje vizibile (tipar codificat) pe obiectele de interes, care pot fi recunoscute automat de algoritmi de procesare a imaginilor.

Un avantaj important al acestei metode este precizia poziționării și simplitatea infrastructurii necesare. În articolul [3], este realizată o comparație între tehnologiile marker-based și markerless pentru captarea mișcării, concluzionându-se că sistemele cu markeri oferă o acuratețe mai mare, mai ales în medii controlate. Deși metodele markerless sunt în creștere, cele bazate pe markeri rămân o soluție stabilă și robustă, în special atunci când se dorește o integrare ușoară și reproductibilă.

O implementare modernă este descrisă în articolul [4], unde este prezentat un sistem de urmărire aproape în timp real folosind biblioteca ArUco. Lucrarea evidențiază avantajele markerilor fiducieri (precum codurile ArUco) în aplicații care necesită identificare rAPIdă și poziționare 2D sau 3D, cu un cost computațional redus. Sistemul utilizează imagini capturate de o cameră standard, iar poziția markerilor este determinată cu ajutorul transformărilor geometrice extrase din imagine.

În contextul lucrării de față, markerii ArUco sunt utilizați atât pentru identificarea vehiculelor, cât și pentru definirea traseului acestora. Aceștia permit o localizare precisă în coordonate metrice, fără a necesita senzori fizici atașați pe vehicul, făcând sistemul scalabil, accesibil și ușor de reprodus în alte medii.

### **3.1.3 Arhitecturi bazate pe containere pentru aplicații cu roboți**

Utilizarea containerelor în arhitecturi distribuite a devenit o metodă tot mai populară pentru dezvoltarea și rularea aplicațiilor robotice. Orchestrarea acestora prin platforme precum Kubernetes oferă avantaje importante precum scalabilitate dinamică, izolare între componente și o gestionare eficientă a resurselor disponibile.

În articolul [5], este propusă o metodologie de dezvoltare bazată pe containere pentru aplicații robotice distribuite, rulabile în medii hibride edge-cloud. Lucrarea evidențiază avantajele împărțirii funcționalităților robotice în microservicii containerizate, care pot fi distribuite inteligent în funcție de cerințele de latență sau resurse. Astfel, componentele care necesită timp de răspuns redus pot fi plasate în apropierea robotului (la marginea rețelei), iar cele mai puțin critice pot rula în cloud.

Această abordare permite o implementare flexibilă și modulară, reducând complexitatea dezvoltării și testării aplicațiilor robotice, în timp ce asigură scalabilitate și mentenanță ușoară într-un mediu de execuție modern.

## **3.2 Alte metode de localizare**

### **3.2.1 Metode de localizare bazate pe LiDAR**

Sistemele de localizare care utilizează senzori LiDAR (Light Detection and Ranging) oferă una dintre cele mai precise metode de poziționare pentru vehicule autonome. Ele funcționează prin emiterea de impulsuri laser și măsurarea timpului necesar pentru ca acestea să se reflecte din mediul înconjurător, generând astfel un nor de puncte 3D care descrie spațiul.

LiDAR este util mai ales în scenarii cu obstacole numeroase sau în condiții de lumină scăzută, deoarece nu depinde de vizibilitate optică. De exemplu, sistemele propuse în articolele [6] și [7] utilizează tehnici de segmentare și SLAM (Simultaneous Localization and Mapping) pentru a localiza vehiculele în timp real, inclusiv în medii subterane sau urbane dense.

Totuși, în cazul vehiculelor RC sau a aplicațiilor educaționale, aceste sisteme sunt dificil de integrat din cauza costului ridicat, greutateii senzorilor și necesității de procesare avansată.

### **3.2.2 Metode de localizare bazate pe GPS**

Sistemele GPS furnizează coordonate globale pe baza semnalelor transmise de sateliți. Ele sunt larg folosite în aplicații de navigație auto și robotică de exterior, datorită simplității de integrare și a acoperirii globale.

În aplicații autonome, vehiculele pot menține formații prin comunicarea pozițiilor proprii, ajustându-și traseele în funcție de deviațiile față de o configurație ideală, așa cum este demonstrat în articolul [8]. Alte lucrări precum [9] arată că un sistem GPS montat pe vehicule mici poate înregistra trasee și detecta obstacole simple cu o precizie acceptabilă pentru navigație de bază.

În scenarii urbane complexe sau în medii acoperite, acuratețea scade drastic. Pentru a compensa, unele soluții propun fuziunea GPS cu senzori inerțiali sau vizuali și comunicații V2V/V2I (vehicul-vehicul sau vehicul-infrastructură), pentru o localizare mai robustă și precisă [10].

### **3.2.3 Compararea metodelor**

Față de metodele prezentate anterior, soluția propusă în această lucrare utilizează procesare centralizată, markerii vizuali ArUco pentru localizare și o arhitectură containerizată orcheștrată prin Kubernetes. Această combinație oferă o vedere completă asupra mediului, procesare paralelă a imaginilor și transmiterea coordonatelor în format metric, fără a suprasolicita vehiculele.

Tabela 2: Comparație între metodele de localizare analizate și cea propusă de lucrare

Criteriu	GPS	LiDAR	Metoda propusă
Precizie	1–5m, afectată de mediu	Sub 10cm, foarte stabilă	Sub 2mm, în zona camerei
Funcționează în interior	Nu	Da, inclusiv în întuneric	Da, cu cameră fixă și lumină minimă
Hardware pe vehicul	Modul GPS cu antenă	Senzor LiDAR și unitate de procesare	Nu este necesar hardware pe vehicul
Complexitate hardware	Redusă	Ridicată, calibrare și montaj dificil	Redusă, doar cameră și markeri
Cost	Scăzut	Ridicat	Foarte scăzut
Infrastructură necesară	Sateliți GPS	LiDAR fix sau pe vehicul	Cameră fixă și markeri ArUco
Reproducere	Limitată la exterior	Dificilă, costuri mari	Ușor de reprodus, costuri minime
Scalabilitate	Limitată de erori și interferențe	Afectată de complexitate computațională	Kubernetes și procesare paralelă
Timp de răspuns	1s	0.1–0.5s	0.3s prin procesare paralelă
Integrare software	Protocoale standard	Necesită biblioteci specializate	OpenCV, Flask și API HTTP

După cum se observă în Tabela 2, metoda propusă oferă un echilibru superior între precizie, cost și scalabilitate. GPS-ul este limitat de precizia scăzută și dependența de semnal, așa cum se arată în [8, 9], iar LiDAR-ul, deși foarte precis, presupune costuri și complexitate ridicate [6, 7]. În schimb, soluția bazată pe markeri vizuali și procesare centralizată permite localizarea precisă în medii interioare cu infrastructură minimă [4, 13]. Totodată, utilizarea Kubernetes pentru orchestrare asigură scalabilitate și timpi de răspuns competitivi [17, 20].

Această soluție este astfel adecvată pentru aplicații educaționale, prototipuri și medii cu resurse reduse, fără a compromite semnificativ performanța de localizare.

## 4 SOLUȚIA PROPUȘĂ

În urma analizării altor soluții pentru localizare, această lucrare propune o implementare precisă pentru a urmări zone de dimensiuni reduse, cu o precizie în ordinul milimetric și o complexitate redusă, fără a avea o diferență semnificativă de timp de calcul și transmitere a informațiilor. Soluția se concentrează pe utilizarea unor componente software și hardware accesibile, evitând echipamente complexe sau algoritmi costisitori din punct de vedere computațional, păstrând în același timp acuratețea necesară pentru aplicații precum coordonarea vehiculelor în medii controlate.

Pentru a avea o poziționare clară, sunt folosite două markere ArUco pentru fiecare vehicul RC, unul pentru fața lui și unul pentru spate, ceea ce permite nu doar identificarea poziției, ci și determinarea orientării direcționale a fiecărui vehicul. În plus, este utilizat un număr ajustabil de markere ce reprezintă traseul pe care acestea îl au de parcurs, fiecare marker având o poziție fixă în cadrul traseului și o semnificație specifică în logica de navigație. Această structură flexibilă permite atât definirea unor trasee simple, cât și extinderea la medii mai complexe, doar prin amplasarea de markere suplimentare și actualizarea configurației din aplicație.

Soluția este alcătuită din două părți principale: o aplicație mobilă și un server pentru procesarea imaginilor și calcularea pozițiilor relevante în sistemul metric. Aplicația rulează direct pe telefonul utilizatorului, fără a necesita instalarea unor componente suplimentare, iar serverul este proiectat pentru a funcționa în medii containerizate, cu posibilitatea de scalare dinamică.

Aplicația mobilă pune la dispoziție ajustarea următorilor parametri: numărul de mașini, numărul de markere pentru traseu, dimensiunea markerelor. Acestea sunt pătrate, deci dimensiunea lor în cm reprezintă mărimea unei laturi. Setările pot fi modificate direct de către utilizator din interfața aplicației, oferind flexibilitate și adaptabilitate în funcție de scenariul testat. Tot aceasta permite folosirea camerei telefonului ca mediu de captare a imaginilor, captură ce se face în fundal la o frecvență constantă, și le trimite asincron o dată la 0.1 s către IP-ul unde serverul este rulat, IP introdus tot de utilizator. Transmiterea se face prin HTTP POST către un endpoint specific, optimizat pentru primirea rapidă a imaginilor. Prin API-ul pus la dispoziție de server, aplicația este capabilă să afișeze, după sfârșitul transmisiei, timpii în care vehiculele și-au terminat traseul, oferind astfel o formă de analiză post-cursă.

Serverul, realizat în limbajul Python folosind Flask, este inițializat cu toți parametrii reglabili trimiși din aplicație. Acesta detectează markerii din imagine folosind librăria OpenCV din Python, mai exact folosind modulul aruco ce oferă metode eficiente de detectare și identificare a markerelor cu ID unic. Pentru a ajunge la dimensiuni reale, serverul folosește metoda solvePnP() din aceeași librărie, pentru a translata coordonatele în planul văzut de cameră, folosindu-se de dimensiunea reală a markerului pentru a obține coordonate ce repre-

zintă distanța în metri în raport cu punctul focal al camerei cu o precizie de 15 zecimale. Astfel, se obține nu doar poziția bidimensională pe imagine, ci și distanța aproximativă față de cameră, care poate fi utilizată pentru estimarea mai precisă a poziției absolute în spațiu.

Pentru a calcula aceste coordonate, este nevoie de dimensiunea reală a markerelor, cât și de matricea de cameră și distorsiunea camerei cu care sunt făcute pozele. Pentru a realiza acest lucru, aplicația are la dispoziție un ecran ce trimite o imagine la apăsarea unui buton către un API. Prin această metodă se obțin acești parametri, prin analiza mai multor poze cu o tablă de șah și metoda `findChessboardCorners()` din OpenCV. Astfel se realizează o calibrare inițială a camerei telefonului, care permite corectarea erorilor de distorsiune și crește semnificativ acuratețea localizării.

De asemenea, pentru comunicare, vehiculelor le este pus la dispoziție un API ce le transmite poziția proprie, a celorlalte vehicule, cât și a markerelor de traseu, calculate în modul explicat anterior. Acest API poate fi apelat periodic de fiecare vehicul RC pentru a obține informații în timp real despre poziția relativă și pentru a putea lua decizii de navigație. Structura răspunsului este JSON, iar coordonatele sunt exprimate în metri în raport cu centrul imaginii captate.

În scopul de a menține viteze competitive cu alte metode, o imagine Docker a serverului este rulată într-un pod Kubernetes, ce folosește un mecanism de autoscalare orizontală pentru a crea alte poduri, în funcție de gradul de procesare ocupat de analiza acestor imagini și de cererile de poziții de la vehicule. Acest mecanism permite adaptarea dinamică la numărul de vehicule și frecvența imaginilor primite, evitând blocajele de procesare și oferind un timp de răspuns constant și scăzut.

Sincronizarea datelor este realizată cu ajutorul serviciului Redis, care stochează toate aceste date pentru a putea fi comunicate între poduri. Redis acționează ca un cache partajat între instanțele serverului și păstrează ultimele poziții calculate, precum și stările vehiculelor în traseu, asigurând coerență între răspunsurile oferite de diverse instanțe ale aplicației.

Prin îmbinarea acestor tehnologii, lucrarea își propune să obțină o soluție rapidă, precisă și ușor de implementat din punct de vedere hardware și software, pentru localizarea vehiculelor RC și coordonarea lor în vederea urmăririi unui traseu predefinit. Soluția oferă un echilibru între simplitatea arhitecturală, scalabilitate și precizie, fiind potrivită pentru aplicații educaționale, prototipuri de sistem de control distribuit sau testări în medii controlate.



## 5 DETALII DE IMPLEMENTARE

### 5.1 Tehnologii

Soluția propusă a fost implementată utilizând Flutter pentru aplicația mobilă și un server construit cu Flask în Python, rulat într-un pod Kubernetes scalat automat pentru procesare paralelă [5, 17]. Această arhitectură modulară permite adaptarea și extinderea ușoară a sistemului, asigurând totodată portabilitate și performanță în execuție.

#### 5.1.1 Flutter

Flutter este un framework dezvoltat de Google, care folosește limbajul Dart pentru crearea de aplicații native pe Android, iOS și Web [15].

Soluția prezentată aici este bazată pe o aplicație Android totuși, arhitectura Flutter permite extinderea către camere integrate în laptopuri sau dispozitive iOS. Frameworkul este bazat pe componente reactive și permite interacțiunea directă cu API-urile native ale platformei.

În cadrul soluției, Flutter se ocupă de controlul camerei telefonului, capturarea și transmiterea fluxului de imagini către server și configurarea parametrilor sistemului de localizare. Utilizarea pachetului camera pentru accesul la hardware-ul camerei, precum și implementarea unei interfețe simple și eficiente, contribuie la robustețea aplicației. Transmiterea datelor se face asincron, cu timpi constanți de captură, iar configurarea sesiunii se realizează direct din interfață, ceea ce permite o experiență de utilizare fluidă și predictibilă.

#### 5.1.2 Markeri ArUco

Markerii ArUco sunt repere bidimensionale fiduciare reprezentate de un contur pătrat negru și de un model interior alcătuit dintr-o matrice de biți alb-negru [4, 13]. Fiecare marker poartă un cod unic, ceea ce permite identificarea sa în imagine și asocierea cu traseul sau vehiculul corespunzător. Datorită contrastului puternic și a formei bine definite, acești markeri pot fi detectați rapid și cu precizie chiar și în condiții de iluminare variabilă sau unghiuri nefavorabile.

În proiectul de față, markerii ArUco servesc ca repere fixe pentru poziționarea traseului și pentru localizarea vehiculelor. Prin detectarea colțurilor și extragerea codului intern, se estimează atât poziția, cât și orientarea fiecărui marker în raport cu camera, făcând posibilă reconstrucția tridimensională a poziției sale [4]. Această informație este apoi folosită de server pentru a determina progresul vehiculelor și pentru a evalua performanța lor în timp real.

### 5.1.3 Server: Python folosind Flask

Serverul central al arhitecturii propuse este scris în Python, un limbaj de programare interpretat, cu sintaxă clară, larg adoptat în prototipare rapidă și aplicații de procesare a imaginilor. În cadrul acestui proiect, serverul se ocupă de primirea imaginilor capturate de aplicația mobilă, prelucrarea acestora pentru a detecta markerii ArUco, estimarea pozițiilor, stocarea rezultatelor și comunicarea acestora către vehiculele RC.

Pentru gestionarea traficului HTTP, a rutelor REST și a logicii de aplicație, s-a utilizat Flask, un microframework Python minimalist, dar puternic [14]. Flask permite definirea rapidă a endpoint-urilor prin utilizarea decoratoarelor precum `app.route`, oferind în același timp o arhitectură flexibilă pentru extindere și integrare cu alte componente.

Serverul utilizează biblioteca OpenCV pentru procesarea imaginilor primite și pentru interpretarea markerilor ArUco. Detectarea acestora și estimarea poziției lor spațiale se face în funcție de dimensiunea markerilor și parametrii camerei obținuți prin calibrare. OpenCV oferă suport complet pentru:

- calibrarea camerei utilizând imagini cu tablă de șah, prin funcțiile de extragere a colțurilor și estimare a matricei intrinseci și vectorului de distorsiune [11];
- detectarea markerilor ArUco prin metode robuste la zgomot și iluminare slabă, folosind coduri unice pentru fiecare marker [13];
- estimarea poziției 3D a markerilor prin rezolvarea problemei PnP, translată apoi în coordonate 2D metrice [12].

Pentru sincronizarea între multiple instanțe de poduri (create automat în Kubernetes), serverul folosește Redis ca stocare temporară în memorie. Aceasta asigură:

- păstrarea stării globale a sesiunii curente (markerii detectați, pozițiile vehiculelor, timpii de finalizare);
- acces rapid la aceste date între instanțele Flask, fără conflicte de scriere;
- persistență temporară cu posibilitatea resetării automate la încheierea sesiunii.

Prin containerizarea aplicației Flask cu Docker și rularea acesteia în Kubernetes, serverul devine portabil, scalabil și ușor de monitorizat. Resursele sunt alocate dinamic în funcție de încărcarea cu imagini, iar sincronizarea cu Redis permite consistența datelor indiferent de câte instanțe rulează în paralel.

Această arhitectură server-side susține performanța în timp real a aplicației și permite extinderea viitoare către scenarii cu un număr mai mare de vehicule sau cu algoritmi avansați de analiză a traseelor.

#### 5.1.4 Redis

Redis este un sistem open-source de stocare în memorie, pe bază de perechi cheie–valoare [16]. Pentru sincronizarea parametrilor globali (calibrare, poziții calculate și setări de sistem), Redis oferă acces rapid la date partajate între instanțele serverului. Acesta asigură persistență opțională pe disc și oferă mecanisme simple de publicare/abonare pentru coordonare internă. Astfel, serverele pot schimba rapid datele de stare între ele fără blocaje sau conflicte.

#### 5.1.5 Docker și Kubernetes

Docker Docker containerizează aplicația server, oferind un mediu izolat și reproductibil, ideal pentru testare și migrare pe infrastructuri diferite [18, 19]. Folosirea fișierului `requirements.txt` pentru specificarea dependențelor permite replicarea rapidă și sigură a mediului de execuție.

Kubernetes Kubernetes orchestrează serviciile și asigură load balancing intern pentru serviciile ClusterIP, autoscalare orizontală a podurilor în funcție de utilizarea resurselor [17, 20] și definirea serviciilor Flask și Redis ca obiecte Kubernetes. Acest sistem garantează reziliență și extindere automată în funcție de cerere, menținând performanța constantă în condiții variabile de trafic.

### 5.2 Aplicație mobilă

#### 5.2.1 Parametrii cameră

Aplicația dezvoltată pentru a demonstra eficiența soluției propuse are ca scop principal transmiterea de imagini suficient de rapid către server pentru a nu rata momentul când un vehicul trece peste un marker ce reprezintă o parte din traseu. Astfel, aplicația este responsabilă de colectarea informațiilor vizuale esențiale pentru algoritmul de localizare de pe server și funcționează ca interfață de interacțiune pentru utilizatorul care controlează camera telefonului.

Pentru a ușura procesul de procesare a imaginilor, are ca funcționalități secundare trimiterea de imagini la apăsarea unui buton, la aceeași rezoluție ca atunci când sunt trimise constant. Această funcție este utilă mai ales în cazul etapelor de calibrare și testare, permițând captarea unor imagini controlate manual. De asemenea, aplicația afișează, în momentul încetării transmiterii imaginilor, o listă cu timpii de terminare a traseului propus pentru fiecare vehicul detectat, ajutând la evaluarea performanței sistemului.

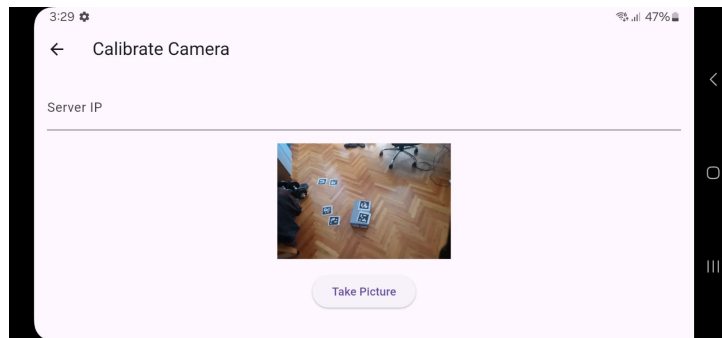


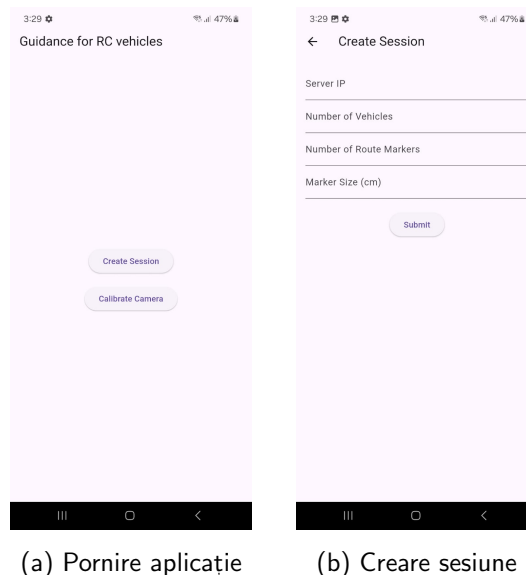
Figura 1: Calibrare cameră

Pentru a obține parametrii de calibrare ai camerei am folosit tutorialul din OpenCV pentru acest lucru, împreună cu imaginile trimise din aplicație. Rezultatul constă în matricea intrinsecă a camerei și vectorul de distorsiune, care sunt salvate într-un fișier auxiliar și transmise la inițializarea sesiunii.

## 5.2.2 Interfață

În continuare este descrisă funcționarea aplicației, exceptând modul în care se obțin parametrii pentru calibrarea camerei, explicat în secțiunea anterioară. Acești parametri ai camerei sunt puși într-un fișier al aplicației pentru a fi trimiși la server în momentul inițializării.

În continuare sunt prezentate capturi de ecran din diferite etape ale aplicației.



(a) Pornire aplicație

(b) Creare sesiune

Figura 2: Interfața aplicației mobile în două momente: pornire și inițializare sesiune

În Figura 2a este prezentat ecranul de pornire al aplicației, care conține următoarele două butoane:

- Create Session: redirecționează utilizatorul către ecranul explicat mai jos;
- Calibrate Camera: declanșează procesul de calibrare a camerei, necesar pentru determinarea parametrilor optici utilizați în calculul poziției markerilor ArUco.

În Figura 2b este prezentată interfața către care utilizatorul este redirecționat după apăsarea butonului de creare sesiune. În acest ecran este activat fluxul de captare și trimiterea continuă a imaginilor către server.

În etapa de creare a unei sesiuni, utilizatorul trebuie să completeze toți parametrii necesari pentru inițializarea corectă a sistemului:

- adresa IP a serverului către care vor fi trimise cererile HTTP;
- numărul total de markere care definesc traseul;
- numărul de vehicule care vor fi prezente în imagine;
- dimensiunea fizică a markerelor utilizate, exprimată în centimetri.

La apăsarea butonului Submit, aplicația trimite o cerere de inițializare către server, incluzând atât parametrii completați de utilizator, cât și parametrii camerei obținuți anterior prin calibrare. Acești parametri sunt folosiți de server pentru a calcula cu precizie pozițiile în coordonate metrice (în metri) ale markerilor ArUco detectați, fie că este vorba despre markere de traseu, fie despre cele asociate vehiculelor RC.

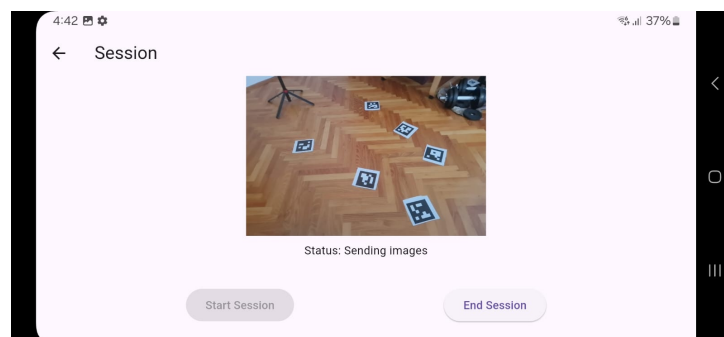
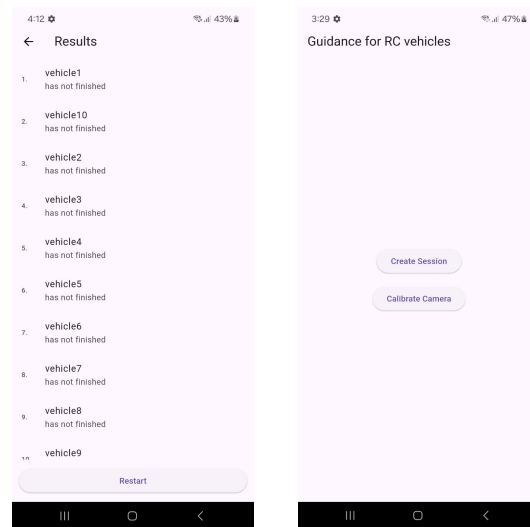


Figura 3: Transmitere imagini

În Figura 3 apare ecranul fixat în modul landscape, de data aceasta, unde este afișat ce vede camera, pentru a putea fi poziționată corect de către utilizator. În momentul în care butonul start session este apăsător, aplicația începe să trimită cereri de tipul POST la API-ul disponibil de imagine, cu frame-urile luate de cameră în mod asincron, ceea ce înseamnă că nu așteaptă răspuns la cerere, pentru a obține o viteză suficientă pentru a nu avea erori o dată la 10ms fiecare.

La apăsarea butonului End Session, aplicația întrerupe procesul de captură și transmitere a imaginilor și inițiază o cerere de tip GET către server, prin care solicită timpii de finalizare calculați pentru fiecare vehicul. Acești timpii sunt ulterior afișați în ecranul dedicat evaluării

sesiunii. Această abordare permite ca finalizarea urmăririi traseului să fie controlată explicit de operatorul camerei, nu de vehiculele autonome, care pot funcționa complet automatizat în cadrul sesiunii curente.



(a) Timpi finali

(b) Pornire aplicație

Figura 4: Interfața aplicației mobile: afișare rezultate și revenire

În Figura 4a este ilustrat ecranul aplicației ce afișează timpii returnați în urma ultimei cereri transmise către server. În exemplul prezentat, sunt analizate datele pentru 10 vehicule, oferind astfel un caz reprezentativ cu un număr ridicat de markere detectate în imagine. Deoarece niciun vehicul nu a finalizat traseul în această sesiune, toți timpii de finalizare sunt marcați ca inexistenți. Funcționalitatea acestui ecran este discutată detaliat în secțiunea dedicată evaluării.

La activarea butonului Restart, aplicația revine la ecranul inițial din Figura 4b, permițând utilizatorului să inițieze o nouă sesiune de testare. Acest comportament emulează o repornire completă a aplicației, resetând contextul curent.

Aplicația mobilă construită în Flutter are un rol esențial în desfășurarea întregului sistem, facilitând comunicarea dintre utilizator și serverul de procesare. Prin funcționalitățile sale, aceasta permite atât calibrarea camerei, cât și transmiterea rapidă și constantă a imaginilor, contribuind direct la precizia localizării vizuale. Etapele aplicației sunt bine delimitate: pornirea aplicației, configurarea sesiunii, rularea efectivă și afișarea rezultatelor. Acest flux oferă o utilizare clară și intuitivă, permițând testarea soluției propuse și adaptarea la diverse scenarii experimentale fără modificări majore.

### 5.3 Server

Componenta de backend urmărește desfășurarea unei întregi sesiuni de coordonare a vehiculelor RC, gestionând starea sistemului în timp real. Aceasta menține o vedere globală asupra

tuturor instanțelor de poduri create prin Kubernetes și expune un set de API-uri REST concepute pentru patru funcții principale: inițializarea sesiunii cu toți parametrii necesari, recepția imaginilor trimise de aplicația Android, calcularea pozițiilor markerilor detectați în imagini, furnizarea acestor poziții către vehiculele participante și extragerea timpilor de finalizare la încheierea cursei.

Fiecare pod care rulează instanța aplicației backend este capabil să preia cereri, să proceseze imagini și să răspundă vehiculelor în mod independent. Totuși, pentru a menține o stare comună și actualizată a sistemului, datele sunt partajate prin Redis, folosit ca serviciu dedicat de stocare în memorie. Acesta centralizează informațiile despre sesiune — inclusiv pozițiile vehiculelor, identificatorii markerilor, starea inițializării și timpii parcurgerii traseului — și permite accesul rapid și sincronizat din orice instanță containerizată.

Această abordare facilitează o scalare eficientă și flexibilă, permițând lansarea sau oprirea automată a podurilor în funcție de volumul de cereri și de resursele consumate. Totodată, asigură consistență în procesarea imaginilor și în răspunsurile furnizate vehiculelor, indiferent de instanța care gestionează fiecare cerere în parte.

Prin integrarea cu Kubernetes și utilizarea Redis ca mecanism de sincronizare, soluția îmbină flexibilitatea arhitecturii distribuite cu fiabilitatea unei logici centralizate de control. În paragrafele următoare sunt prezentate în detaliu toate API-urile disponibile, precum și modul în care această componentă este containerizată și orchestrată într-un sistem robust și scalabil.

### **5.3.1 Funcționare API-uri**

În continuare este explicată funcționarea API-urilor puse la dispoziție de server, fiecare având un rol esențial în gestionarea fluxului de date și în asigurarea unei comunicări stabile și eficiente între aplicația mobilă, server și vehiculele RC.

```

@app.route('/api/initialize', methods=['POST'])
def initialize():
    if 'number_of_route_markers' not in request.form:
        return jsonify({'error': 'Missing number of route markers'}), 400
    if 'number_of_cars' not in request.form:
        return jsonify({'error': 'Missing number of cars'}), 400
    if not r.exists("INITIALIZED"):
        r.set("INITIALIZED", "0")

    r.set("INITIALIZED", "1")

    number_of_route_markers = int(request.form['number_of_route_markers'])
    number_of_cars = int(request.form['number_of_cars'])

    size_cm = float(request.form['marker_size_cm'])
    camera_matrix = json.loads(request.form['camera_matrix'])
    dist_coeffs = json.loads(request.form['dist_coeffs'])

    if number_of_route_markers <= 0 or number_of_cars <= 0:
        return jsonify({'error': 'Incorrect parameters received'}), 400

    r.set("CAMERA_MATRIX", json.dumps(camera_matrix))
    r.set("DIST_COEFFS", json.dumps(dist_coeffs))
    r.set("MARKER_LENGTH", str(size_cm / 100))

    target_markers = list(range(0, number_of_route_markers))
    r.set("TARGET_MARKERS", json.dumps(target_markers))
    r.delete("VEHICLE_MARKERS")
    vehicle_markers = {}
    for i in range(number_of_cars):
        mid = number_of_route_markers + 2 * i
        name = f"vehicle{i+1}"
        vehicle_markers[name] = mid
        r.hset("VEHICLE_MARKERS", name, mid)
    r.delete("FINISH_TIME")
    for i in range(number_of_cars):
        mid = number_of_route_markers + 2 * i
        flags = [0] * number_of_route_markers
        r.hset("FINISH_TIME", mid, json.dumps(flags))
    r.delete("FINISH_TIME")
    for i in range(number_of_cars):
        name = f"vehicle{i+1}"
        r.hset("FINISH_TIME", name, "1")
    r.delete("LAST_POSITIONS")
    for i in range(0, number_of_cars):
        mid = number_of_route_markers + i
        pos = [0, 0]
        r.hset("LAST_POSITIONS", str(mid), json.dumps(pos))

    r.delete("START_TIME")
    r.delete("target_marker_containers")
    return jsonify({'successful': 'Successful initialization'}), 200

```

Figura 5: API Initializare

În Figura 5 este prezentat codul API-ului de inițializare a sesiunii. Acest endpoint este responsabil pentru configurarea inițială a sistemului înainte de începerea oricărei transmisii de imagini sau calcul de poziții. Aplicația trimite o cerere POST ce conține un obiect JSON cu toți parametrii esențiali pentru rularea corectă a sesiunii: numărul de vehicule RC care vor fi urmărite, numărul total de markere care definesc traseul, dimensiunea fizică (în cm) a markerilor ArUco utilizați și parametrii obținuți prin calibrarea camerei.

După recepționarea cererii, serverul validează toate câmpurile — spre exemplu, verifică faptul că numerele introduse nu sunt negative sau nule și că matricea camerei este completă. Odată ce validarea este trecută, serverul setează în Redis starea sistemului ca fiind „inițializat” și construiește structura de date de bază necesară pentru desfășurarea sesiunii. Aceasta include lista markerilor de traseu, cu ID-uri alocate secvențial de la 0 la N-1 (unde N este numărul de markere), pozițiile curente ale vehiculelor (inițial goale) și un registru de timp pentru fiecare vehicul, care va fi completat la finalul cursei. Vehiculele primesc ID-uri care urmează direct după markerii de traseu, astfel încât sistemul să poată distinge clar între markerii statici și cei mobili, fără ambiguități în identificare.



```

@app.route('/api/image', methods=['POST'])
def image():
    if not r.exists("INITIALIZED"):
        return jsonify({'error': 'Current session has not been initialized yet.'}), 400
    initialized = (r.get("INITIALIZED") == "1")
    if not initialized:
        return jsonify({'error': 'Current session has not been initialized yet.'}), 400
    if not r.exists("START_TIME"):
        r.set("START_TIME", str(time.time()))
    file = request.files.get('image')
    if not file:
        return jsonify({'error': 'No image sent.'}), 400
    file_name = secure_filename(file.filename)
    if not is_image(file_name):
        return jsonify({'error': 'Invalid file.'}), 400
    pil = image.open(io.BytesIO(file.read())).convert('RGB')
    buf = io.BytesIO()
    pil.save(buf, format='JPEG', quality=80, optimize=True)
    clean_img = buf.getvalue()
    image = cv2.imdecode(np.asarray(bytearray(clean_img), dtype='uint8'), cv2.IMREAD_COLOR)
    marker_corners, marker_ids, _ = detector.detectMarkers(image)
    if marker_ids is not None:
        markers = []
        for id, corner in zip(marker_ids.flatten(), marker_corners):
            pts = corner[0].tolist()
            markers.append({
                'id': id,
                'corner': pts
            })
        camera_matrix = np.array(json.loads(r.get("CAMERA_MATRIX")))
        distortion = np.array(json.loads(r.get("DIST_COEFFS")))
        marker_size = float(r.get("MARKER_SIZE"))
        marker_points = np.array([marker_size / 2, marker_size / 2, 0],
                                [marker_size / 2, marker_size / 2, 0],
                                [marker_size / 2, marker_size / 2, 0],
                                [marker_size / 2, marker_size / 2, 0]), dtype=np.float32)
        twvecs = []
        for marker in markers:
            img_pts = np.array(marker['corner'], dtype=np.float32)
            _ , twvec = cv2.solvePnP(
                marker_points, img_pts, camera_matrix, distortion=None, None, False, flags=cv2.SOLVEPNP_IPPE_SQUARE
            )
            twvecs[marker['id']] = twvec.flatten().tolist()
        last_positions = {}
        for k, v in r.hgetall("LAST_POSITIONS").items():

```

(a) API Image 1

```

        for marker in markers:
            if marker['id'] in last_positions.keys():
                last_positions[marker['id']] = [twvecs[marker['id']][0], twvecs[marker['id']][1]]
                r.hset("LAST_POSITIONS", str(marker['id']), json.dumps(last_positions[marker['id']]))

        target_markers = json.loads(r.get("TARGET_MARKERS"))
        t_centers = {}
        for i in target_markers:
            if not r.exists("target_marker_centers"):
                for marker in markers:
                    if marker['id'] in target_markers:
                        t_centers[marker['id']] = [twvecs[marker['id']][0], twvecs[marker['id']][1]]
                        r.set("target_marker_centers", json.dumps(t_centers))

        vehicle_map = {}
        for k, v in r.hgetall("VEHICLE_MAP").items():
            raw_t = r.get("target_marker_centers")
            t_pos = dist(json.loads(raw_t))

        for car in vehicle_map.values():
            raw_flags = r.hget("FINISHED", str(car))
            fcar = json.loads(raw_flags)

            it = 0
            for center in t_pos.values():
                x, y = center
                raw_ip = r.hget("LAST_POSITIONS", str(car))
                sx, sy = json.loads(raw_ip)
                distance = ((x - sx)**2 + (y - sy)**2) ** 0.5
                if distance <= marker_size and fcar[it] == 0:
                    fcar[it] = 1
                    r.hset("FINISHED", str(car), json.dumps(fcar))
                    if 0 not in fcar:
                        for name, mid in vehicle_map.items():
                            if mid == car:
                                identifier = name
                                if r.hget("FINISHED_TIMES", identifier) == "-1":
                                    elapsed = time.time() - float(r.get("START_TIME"))
                                    r.hset("FINISHED_TIMES", identifier, str(elapsed) + "s")
                                break
                    it += 1
            return jsonify({'successful': 'Successful image upload'})

```

(b) API Image 2

Figura 6: API image

În Figura 6 este prezentat codul API-ului care primește imaginile de la aplicație. Acesta este unul dintre cele mai frecvent apelate endpoint-uri din sistem, având rolul de a asigura fluxul continuu de date vizuale între telefonul mobil și server. Aplicația trimite o imagine JPEG la fiecare 0.1 secunde, iar serverul o primește și o decodează folosind biblioteca PIL pentru a reconstrui obiectul imagine într-un format compatibil cu OpenCV. Această etapă este necesară deoarece pot exista pierderi sau coruperi minore ale datelor în timpul transmisiei rapide și continue.

Odată ce imaginea este validă, se trece la procesarea efectivă: markerii ArUco sunt detectați în imagine, iar pentru fiecare marker identificat se extrag coordonatele acestuia în spațiul imaginii. Cu ajutorul funcției `cv2.solvePnP` și pe baza parametrilor de calibrare și a dimensiunii markerului, se calculează poziția markerului în spațiul tridimensional. Din acest set de coordonate, sunt păstrate doar componentele X și Y pentru a construi poziționarea bidimensională, adecvată pentru o reprezentare plană. Componenta Z este ignorată, deoarece markerii de pe traseu și cei de pe vehicule pot fi poziționați la înălțimi diferite, ceea ce poate introduce erori în evaluarea distanței față de cameră.

În urma acestei procesări, serverul compară poziția markerului de pe vehicul cu pozițiile markerilor de traseu. Dacă distanța față de următorul marker este mai mică decât jumătate din latura markerului (considerată rază de toleranță), se consideră că vehiculul a ajuns pe acel punct al traseului. Dacă toate markerii au fost parcurși, timpul curent (obținut din timpul global al sesiunii) este salvat pentru acel vehicul în Redis, marcând finalizarea traseului pentru acesta.

```

@app.route('/api/get_pos', methods=['GET'])
def get_pos():
    initialized = r.get("INITIALIZED") == "1"
    if not initialized:
        return jsonify({'error': 'Current session has not been initialized yet.'}), 400

    sender = request.args.get('sender')

    if not sender:
        return jsonify({'error': 'Missing sender field'}), 400

    row_t = r.get("target_marker_centers")
    if row_t is None:
        return jsonify({'error': 'No image has been received yet.'}), 400
    target_markers = json.loads(row_t)

    vehicle_map = {}
    for k, v in r.hgetall("VEHICLE_MARKERS").items():
        if sender not in vehicle_map:
            return jsonify({'error': 'Unknown sender'}), 400
        car_id = vehicle_map[sender]
        row_lp = r.hget("last_positions", str(car_id))
        tx, ty = json.loads(row_lp)

        row_finished = r.hget("FINISHED", str(car_id))
        finished_flags = json.loads(row_finished)

        last_positions = {
            int(k): json.loads(v)
            for k, v in r.hgetall("last_positions").items() if k != str(car_id) and k != (str(car_id) + 1)
        }

    if 0 not in finished_flags:
        tx, ty = -999, -999

    return jsonify({
        'sender': sender,
        'markers_locations': target_markers,
        'other_vehicles': last_positions,
        'your_location': (tx, ty),
    })

```

Figura 7: API get position

În Figura 7 este ilustrat API-ul utilizat de vehicule pentru a obține pozițiile actuale. Acest endpoint este apelat printr-un request GET de către fiecare vehicul RC, periodic sau în funcție de o logică locală de control, pentru a primi datele relevante. Răspunsul este formatat ca un obiect json care conține: id-ul vehiculului apelant, poziția markerului său frontal, pozițiile markerilor altor vehicule (inclusiv -uri și coordonate), precum și pozițiile markerilor de traseu rămase de parcurs.

În momentul în care un vehicul a ajuns la ultimul marker, acest API nu mai furnizează coordonatele active, ci returnează valori invalide (–999, –999). Acest semnal este interpretat de vehicul ca indicator că traseul a fost complet parcurs și că vehiculul poate opri mișcarea sau poate ieși de pe circuit. Această logică permite serverului să oprească procesarea inutilă pentru vehiculele finalizate și ajută la conservarea resurselor de calcul și comunicare.

```

@app.route('/api/get_times', methods=['GET'])
def get_times():
    finish_times_raw = r.hgetall("FINISH_TIMES")
    r.delete("INITIALIZED")

    return jsonify({'finish': finish_times_raw}), 200

```

Figura 8: API get times

Figura 8 arată API-ul pentru obținerea timpilor de finalizare pentru fiecare vehicul. Acesta este de regulă apelat de aplicația mobilă la finalul sesiunii, după ce toate vehiculele au parcurs traseul. Răspunsul este un obiect JSON care conține, pentru fiecare ID de vehicul, timpul în secunde în care acesta a ajuns la final. Aceste date pot fi folosite pentru afișarea de clasamente, evaluări de performanță sau analize comparative între mai multe sesiuni.

În plus, odată cu apelarea acestui endpoint, serverul revine automat la starea „neinițializată” în Redis, ștergând datele sesiunii curente și pregătind sistemul pentru o eventuală reluare de la zero. Acest comportament asigură o delimitare clară între sesiuni și previne conflictele între datele vechi și cele noi.

### 5.3.2 Containerizare și orchestrare

Această subsecțiune prezintă modul în care este creată imaginea Docker a serverului și cum este aceasta folosită de Kubernetes pentru a lansa instanțe containerizate și pentru a scala sistemul automat, în funcție de nivelul de încărcare cauzat de procesarea imaginilor și de cererile de la vehicule.

```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5000

CMD ["gunicorn", "--bind", "0.0.0.0:5000", "server:app"]
```

Figura 9: Dockerfile

În Figura 9 este prezentat Dockerfile-ul folosit pentru a crea containerul aplicației server. Acesta include un fișier requirements.txt pentru instalarea automată a tuturor dependențelor, ceea ce asigură portabilitatea și replicabilitatea imaginii. Fișierele serverului, împreună cu toate resursele necesare pentru rulare, sunt copiate în imagine, iar serverul Flask este lansat automat la pornirea containerului. Această abordare permite rularea identică a serverului pe orice infrastructură care suportă containere Docker, fără modificări suplimentare de configurare.

Imaginea astfel obținută este publicată pe Docker Hub, într-un registry accesibil public sau privat, și este preluată ulterior de Kubernetes pentru a lansa instanțe automatizate ale aplicației.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: aruco-api
spec:
  replicas: 1
  selector:
    matchLabels:
      app: aruco-api
  template:
    metadata:
      labels:
        app: aruco-api
    spec:
      containers:
        - name: aruco-api
          image: radulescumatei76/aruco-api:latest
          ports:
            - containerPort: 5000
          env:
            - name: REDIS_HOST
              value: "redis"
            - name: REDIS_PORT
              value: "6379"

```

(a) Pod

```

minReplicas: 1
maxReplicas: 10
metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 60

```

(b) Scalare

Figura 10: Creare pod Kubernetes și scalarea acestuia

În Figura 10a este reprezentată configurația podului principal folosit pentru lansarea aplicației într-un mediu controlat. Fiecare pod rulează o instanță izolată a aplicației Flask și se conectează la aceeași instanță a serviciului Redis, care acționează ca zonă de memorie partajată între toate replicile. Astfel, datele critice precum pozițiile vehiculelor, markerii de traseu și timpii de finalizare sunt disponibile oricărui pod din sistem, indiferent de cine procesează imaginea sau răspunde unei cereri API.

În Figura 10b este ilustrat mecanismul de scalare automată implementat cu ajutorul unui Horizontal Pod Autoscaler (HPA). Acesta monitorizează în timp real consumul de CPU al containerelor și menține o încărcare maximă de 60%. Astfel, în momentul în care mai multe imagini sunt trimise simultan de aplicația mobilă, iar cererile vehiculelor se acumulează, Kubernetes detectează creșterea utilizării resurselor și creează automat noi instanțe ale serverului. Noile poduri sunt încărcate cu imaginea publicată anterior, inițializează serverul Flask și se sincronizează imediat cu celelalte prin Redis.

Această orchestrare permite o scalare elastică și complet automatizată, fără intervenție manuală. Practic, în funcție de volumul de date procesat și de gradul de paralelism cerut de sistem (număr de vehicule, frecvența transmiterii imaginilor), aplicația poate funcționa constant în regim de performanță ridicată, adaptându-se la condițiile reale de funcționare. Astfel se evită blocajele cauzate de procesare secvențială, iar serverul rămâne receptiv și în cazul unor sarcini ridicate.

Această abordare modulară și containerizată oferă multiple avantaje: portabilitate completă a aplicației, ușurință în implementare pe orice sistem compatibil Docker, toleranță la erori prin replicare și posibilitatea de extindere ulterioară a arhitecturii prin adăugarea de componente auxiliare, fără modificări majore în sistemul de bază.

## 6 EVALUARE

Pentru a evalua soluția propusă de această lucrare au fost testate mai multe aspecte: eroarea medie de pozitionare pentru calcularea distanței dintre două marker, viteza cu care este procesată o imagine, scalarea implementării în funcție de numărul de mașini, scalarea în funcție de viteza cu care sunt transmise imaginile și corectitudinea terminării traseului din două puncte de vedere distincte la creșterea numărului de vehicule cât și în aplicație pentru cazuri reale.

Testele pentru validarea funcționalității aplicației și evaluarea performanței sistemului au fost realizate pe o mașină fizică cu următoarea configurație hardware:

- Procesor: Intel Core i5-12400, generația a 12-a, 6 nuclee fizice și 12 fire de execuție;
- Frecvență de bază: 3.90 GHz, frecvență maximă: 4.40 GHz;
- Cache total: 288 KiB (L1d), 192 KiB (L1i), 7.5 MiB (L2), 18 MiB (L3);
- RAM disponibil: 15 GiB memorie fizică, din care 11 GiB utilizați în timpul testelor;
- Sistem de operare: Debian GNU/Linux 12 (Bookworm), kernel 6.8.12, arhitectură x86\_64;
- Virtualizare activă: VT-x, suport complet pentru containere Docker și orchestrare cu Kubernetes.

Pentru a testa îmbunătățirea adusă de rularea în Kubernetes, aplicația a fost testată în două scenarii distincte:

1. Test local: aplicația Flask care procesează imaginile a fost rulată direct, fără orchestrare, împreună cu instanța Redis. Aceste teste au permis validarea inițială a funcționalității și măsurarea comportamentului aplicației fără izolare sau replicare.
2. Test în cluster Kubernetes: instanțele serverului și Redis au fost containerizate și lansate ca poduri într-un cluster orchestrat, configurat cu autoscalare activă.

Această dublă abordare a evidențiat avantajele semnificative ale rulării în Kubernetes: scalabilitate dinamică, izolare completă între componente, precum și stabilitate crescută în condiții de trafic ridicat, fără a compromite timpii de răspuns ai API-urilor.

### 6.1 Generare Imagini

Pentru a evalua comportamentul sistemului în scenarii complexe, ce implică un număr mare de vehicule RC, a fost necesară generarea unui set extins de imagini de test. Întrucât nu au

fost disponibile fizic suficiente vehicule pentru testare în paralel, a fost adoptată o metodă de simulare vizuală realistă, folosind biblioteci specializate.

Astfel, au fost generate imagini folosind funcționalitatea `generateImageMarker` din biblioteca `OpenCV`, împreună cu crearea unui padding alb în jurul fiecărui marker pentru a evidenția conturul acestuia. Imaginea de fundal utilizată a fost extrasă dintr-o fotografie reală trimisă de aplicația mobilă, pentru a păstra aceeași rezoluție și condiții de captură ca în cazurile reale. Acest lucru asigură compatibilitatea completă cu procesul de detecție din server, dar și cu performanțele de rețea testate în cadrul benchmark-ului.

Folosind această metodă, au fost generate serii de imagini care conțin între 1 și 10 vehicule, fiecare reprezentat printr-un marker frontal unic. Vehiculele sunt adăugate secvențial, permițând testarea scalării sistemului în funcție de numărul entităților active.

De asemenea, au fost generate alte serii de imagini cu până la șase vehicule, în care markerii acestora au fost poziționați direct pe markerii de traseu, pentru a simula scenariul în care vehiculele ating secvențial punctele de control. Aceste cazuri sunt utile pentru a testa logica de detectare a parcurgerii complete a traseului de către fiecare vehicul.

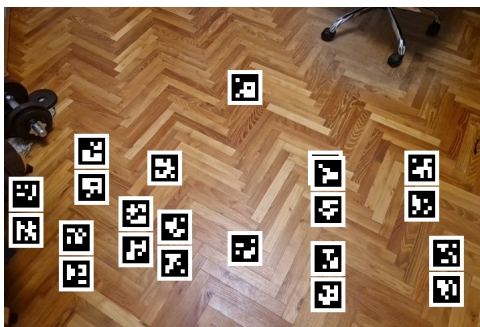


Figura 11: Imagine generată cu 9 vehicule

În Figura 11 este ilustrată una dintre imaginile generate care conține 9 vehicule. Fiecare marker este poziționat într-o zonă clar delimitată și este procesabil de către server fără diferențe semnificative față de capturile obținute din aplicația mobilă.

Prin utilizarea acestei metode de generare artificială de imagini s-a putut simula o varietate de situații reale, fără a fi necesară prezența fizică a vehiculelor. Această abordare s-a dovedit eficientă pentru testarea algoritmilor de procesare, validarea corectitudinii sesiunilor și evaluarea performanțelor sistemului în condiții variate de complexitate.

## 6.2 Testare eroare

Pentru a evalua precizia sistemului de localizare, a fost realizat un test controlat în care două markere `ArUco` au fost plasate la o distanță fixă și cunoscută de 40 de centimetri. În acest context, au fost capturate și trimise către server 125 de imagini în care pozițiile

celor două markere au fost detectate și procesate folosind funcționalitatea implementată cu `cv2.solvePnP` din biblioteca OpenCV.

Scopul acestui test a fost acela de a cuantifica eroarea absolută a distanței calculate între cele două markere și de a observa stabilitatea estimării pozițiilor în timp.

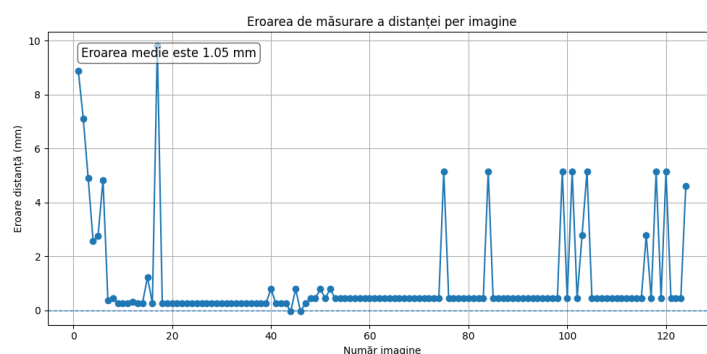


Figura 12: Eroare distanță calculată între markere

În Figura 12 este prezentat graficul diferenței dintre distanța calculată și valoarea de referință de 40 cm. Se poate observa că, în majoritatea cazurilor, eroarea absolută este mai mică de 1 mm, ceea ce denotă o precizie excelentă a sistemului de localizare. Eroarea medie calculată pe întregul set de date a fost de 1.05 mm, o valoare remarcabil de mică pentru o metodă de localizare vizuală bazată pe imagini RGB și calibrare standard a camerei.

Există câteva valori anormale (outliere) cu erori ce ajung până la 10 mm, însă acestea sunt rare și pot fi explicate prin factori perturbatori precum unghiuri de filmare nefavorabile, blur de mișcare sau iluminare neuniformă în zona markerelor.

Este important de menționat că aceste erori mai mari pot fi reduse semnificativ în practică dacă imaginile sunt transmise cu o frecvență suficient de ridicată. Un interval scurt între capturi asigură faptul că markerii sunt vizibili din unghiuri mai favorabile în cadrul secvenței de imagini, iar estimările de poziție pot fi filtrate temporal (ex. mediere sau interpolare). Acest comportament este vizibil și în graficul din Figura 12, unde se observă că nu există secvențe lungi de imagini cu eroare mare – între două valori anormale există întotdeauna estimări corecte cu eroare sub pragul mediu.

Acest test confirmă faptul că soluția propusă poate furniza estimări de poziție cu o precizie foarte bună pentru aplicații reale de coordonare a vehiculelor, fără a necesita senzori suplimentari sau echipamente hardware costisitoare. Rezultatele validează astfel alegerea markerelor ArUco ca metodă principală de localizare în cadrul arhitecturii propuse.

## 6.3 Testare viteză de procesare imagini

Pentru a evalua timpul necesar procesării unei imagini trimise de aplicație, au fost realizate teste comparative între două scenarii de rulare: rularea locală directă (fără containere) și rularea containerizată în Kubernetes cu autoscalare activă. În ambele cazuri, au fost transmise imagini cu 10 vehicule în același cadru, la un interval constant de 0.5 secunde.

Scopul testului a fost să se observe dacă sistemul poate menține o viteză de procesare suficient de rapidă pentru a face posibilă coordonarea în timp real, chiar și în scenarii aglomerate vizual.

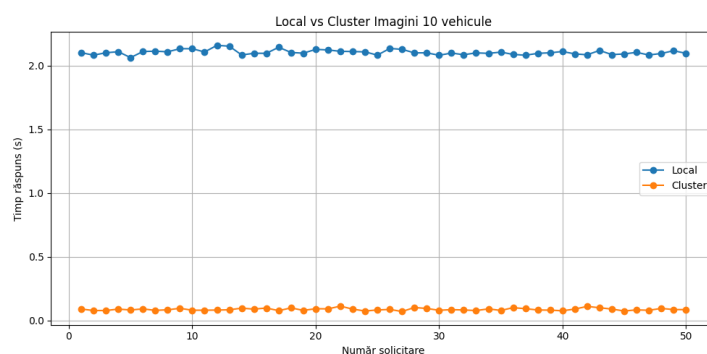


Figura 13: Comparație timp procesare imagine – local vs Kubernetes

În Figura 13 este ilustrată diferența dintre timpul de răspuns la trimiterea unei imagini către server în cele două configurații.

În varianta locală, fără orchestrare sau containere, se observă un timp mediu de aproximativ 2.2 secunde pentru primirea răspunsului, cu fluctuații semnificative cauzate de lipsa izolației și a alocării dinamice de resurse. Aceste fluctuații sunt vizibile sub forma unor oscilații neregulate în grafic și sugerează o instabilitate în comportamentul serverului sub sarcină.

Prin contrast, în cazul rularii în Kubernetes, timpul de răspuns mediu este redus considerabil, situându-se în jurul valorii de 0.1–0.35 secunde, în funcție de rata de trimitere și de numărul de poduri active în acel moment. Fluctuațiile sunt semnificativ mai mici, iar comportamentul sistemului este mai aproape de unul liniar, ceea ce sugerează o predictibilitate mult mai bună.

Un alt aspect important de menționat este faptul că viteza de procesare nu depinde exclusiv de puterea de calcul disponibilă, ci și de cât de repede sunt transmise imaginile. Dacă acestea sunt trimise cu o frecvență ridicată, sistemul este forțat să răspundă în mod continuu, ceea ce pune presiune pe orchestrator pentru a crea poduri noi. În aceste condiții, valorile observate de 0.35 secunde sunt reprezentative pentru sarcini reale.

Performanța crescută observată în Kubernetes se datorează în principal:

- distribuirii automate a sarcinii de procesare pe mai multe poduri;
- separării logice a fiecărei instanțe de server, ceea ce elimină blocajele mutuale între cereri;



- capacității sistemului de a scala automat în funcție de încărcare, prin adăugarea dinamică de resurse.

Rezultatele testului demonstrează că arhitectura propusă poate procesa cadre cu densitate mare (10 vehicule simultan), la o frecvență de 0.5 secunde sau mai mică, fără a introduce întârzieri care să compromită precizia localizării. Această performanță confirmă faptul că implementarea este adecvată pentru aplicații în timp real, chiar și în condiții dificile.

## 6.4 Testare scalare cu numărul de vehicule

Pentru a testa comportamentul sistemului în funcție de numărul de vehicule conectate simultan, au fost trimise imagini cu între 1 și 10 vehicule, fiecare imagine fiind transmisă la un interval constant de 0.5 secunde. În paralel, toate vehiculele simulau comportamentul real, trimițând cereri periodice către server pentru a primi poziția curentă.

Acest test a vizat două direcții principale:

- influența numărului de vehicule asupra timpului de procesare al cererilor de poziție
- identificarea eventualelor blocaje sau creșteri semnificative de latență atunci când sarcina sistemului crește.

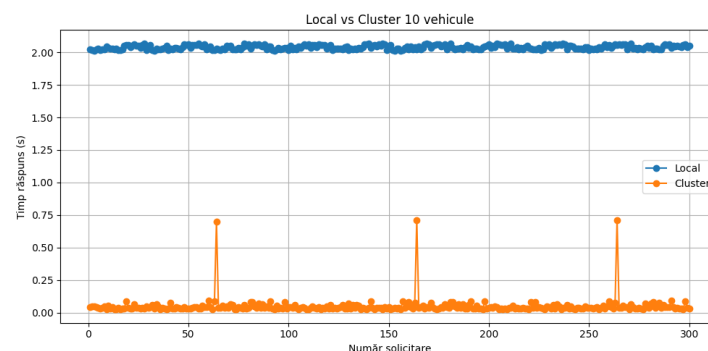


Figura 14: Comparatie timp de raspuns – cereri GET pozitie

În Figura 14 este prezentat timpul de răspuns pentru cererile GET făcute de vehicule pentru a-și obține poziția. Comparăm aici două scenarii: rularea locală, fără containerizare, și rularea în Kubernetes cu autoscalare activă.

Se observă că în cazul rulării locale, timpii de răspuns cresc odată cu numărul de vehicule, în timp ce în Kubernetes, răspunsurile sunt semnificativ mai rapide și mai constante. În mod interesant, în Kubernetes, timpii pentru aceste cereri sunt mai mici decât cei pentru procesarea imaginilor (prezentată anterior), ceea ce sugerează că procesarea imaginilor reprezenta principalul factor limitativ.

Prezența autoscalării și a distribuției cererilor GET către mai multe poduri permite o balansare eficientă a încărcării. Acest lucru a dus la eliminarea aproape completă a blocajelor, cu excepția a 3 cazuri izolate dintr-un total de 300 de cereri – ceea ce confirmă fiabilitatea ridicată a soluției implementate.

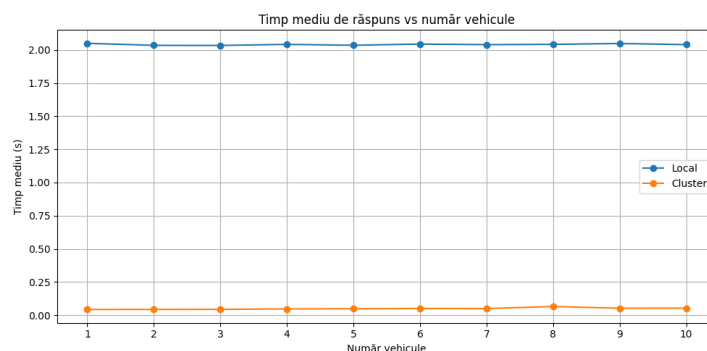


Figura 15: Timp mediu de răspuns – în funcție de numărul de vehicule

Figura 15 evidențiază timpul mediu de răspuns pentru cererile de poziție în funcție de numărul de vehicule. Chiar dacă toate vehiculele trimit cereri simultan, timpul de răspuns rămâne aproape constant în ambele scenarii, cu o performanță semnificativ mai bună în varianta rulată în Kubernetes.

Acest rezultat confirmă că sistemul este scalabil în raport cu numărul de vehicule și că infrastructura containerizată răspunde adecvat la creșterea cererii, fără a introduce întârzieri majore.

Mai mult, se observă că limita superioară de vehicule testate nu este impusă de capacitatea serverului, ci de constrângerile fizice – mai exact, de câte vehicule pot fi captate simultan în cadrul camerei video folosite pentru detecție.

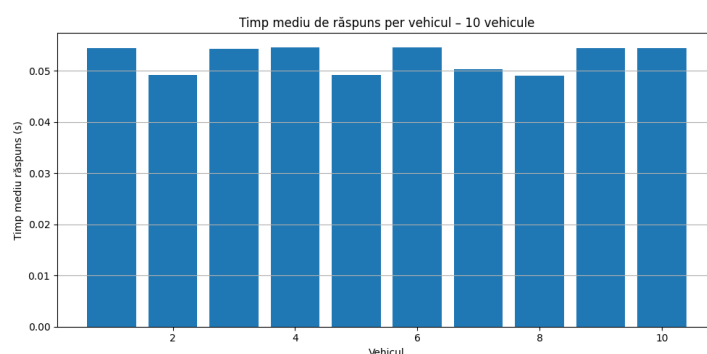


Figura 16: Timp de răspuns mediu pe vehicul – 10 vehicule

În Figura 16 este reprezentată distribuția timpilor de răspuns pentru fiecare vehicul în parte, în cazul testului cu 10 vehicule. Se remarcă faptul că diferențele între vehicule sunt extrem de mici – sub 0.01 secunde – ceea ce arată că sistemul oferă un tratament echitabil și uniform tuturor cererilor, fără favorizarea anumitor vehicule sau apariția unor decalaje semnificative.

Această observație este importantă în scenarii unde sincronizarea și coordonarea sunt critice, deoarece asigură că toate vehiculele primesc informații actualizate într-un timp aproape identic, evitând astfel comportamente imprevizibile cauzate de latențe neuniforme.

În concluzie, testele arată că sistemul este robust, scalabil și optimizat pentru coordonarea simultană a unui număr ridicat de vehicule RC. Rularea în Kubernetes oferă beneficii clare în ceea ce privește performanța și fiabilitatea, făcând posibilă extinderea rapidă a sistemului fără modificări arhitecturale majore.

## 6.5 Testare viteză transmitere imagini

Pentru a testa scalarea în funcție de viteza cu care sunt transmise imaginile către server, au fost trimise în paralel cereri folosind mai multe thread-uri. Acest test a urmărit să determine la ce punct viteza de trimitere a imaginilor devine mai mare decât capacitatea de procesare a serverului, generând un backlog și influențând negativ timpul de răspuns.

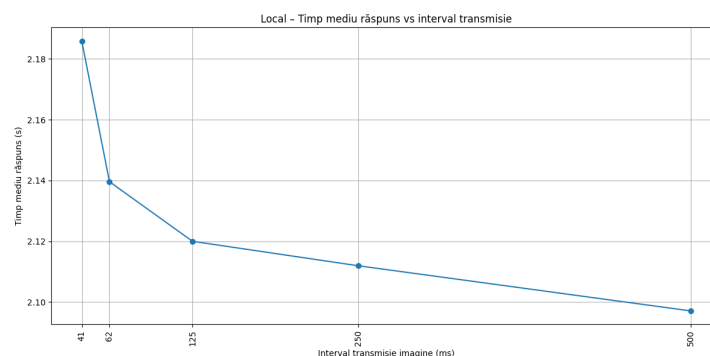


Figura 17: Timp vs Viteză de transmisie – rulare locală

În Figura 17 se observă timpul de răspuns în cazul rulării fără containerizare. Timpii sunt mari, similari cu celelalte teste efectuate anterior în mediul local. Se remarcă faptul că, începând cu intervale de trimitere sub 200 ms, serverul începe să nu mai țină pasul, crescând timpul de răspuns progresiv. Acest lucru indică o lipsă de paralelism în procesarea cererilor și o limitare clară a resurselor disponibile în procesul unic.

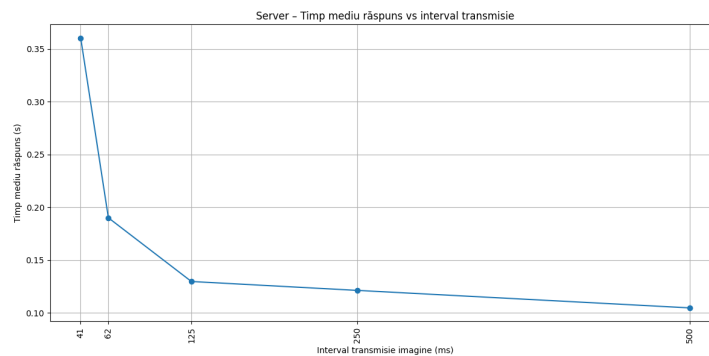


Figura 18: Timp vs Viteză de transmisie – rulare în Kubernetes

În Figura 18 se observă comportamentul serverului în varianta rulată cu scalare automată activă. Spre deosebire de cazul local, aici sistemul reușește să mențină timpi de răspuns stabili pentru intervale de trimitere mai scurte. Cu toate acestea, se remarcă faptul că, odată ce intervalul scade sub 50 ms, timpul de procesare per imagine începe să crească semnificativ.

Acest comportament este cauzat de bottleneck-ul observat anterior, legat de procesarea intensivă a imaginilor în paralel. Spre deosebire de cazul local, unde serverul nu procesa deloc în anumite perioade scurte, aici toate imaginile sunt procesate, dar creșterea frecvenței de trimitere depășește temporar capacitatea de alocare a podurilor suplimentare, ceea ce duce la o întârziere cumulativă.

Se observă că la transmisia la 41 ms a unei imagini, serverul se apropie de limita maximă de procesare, fiind deja foarte aproape de a avea un timp de răspuns mai mare decât viteza de trimitere. Acest punct marchează începutul saturației sistemului, moment în care nu mai este posibilă scalarea în timp real fără un număr mai mare de resurse disponibile (ex: CPU cores).

Timpul exact de transmisie asincron nu a putut fi determinat cu precizie din cauza numărului limitat de core-uri disponibile la testarea benchmark-ului. Totuși, deoarece aplicația realizată trimite imagini de aproximativ 4 ori mai rapid decât cazul descris anterior, se poate concluziona că implementarea atinge deja pragul maxim de procesare paralelă. Trimiterea la o viteză mai mare decât cea de procesare are un efect benefic: asigură generarea automată de poduri noi prin mecanismul de autoscalare, prevenind acumularea de întârzieri în perioadele de vârf.

Această testare confirmă că arhitectura containerizată este capabilă să răspundă eficient în scenarii cu transmisie rapidă și concurență ridicată, atâta timp cât sistemul are la dispoziție suficiente resurse pentru a menține scalarea în timp real.

## 6.6 Testare terminare traseu

Pentru a testa faptul că serverul detectează corect terminarea unui traseu de către un vehicul, au fost realizate teste folosind atât imagini generate automat, cât și poze reale trimise direct din aplicația mobilă. Scopul acestor teste a fost de a valida comportamentul logic al serverului

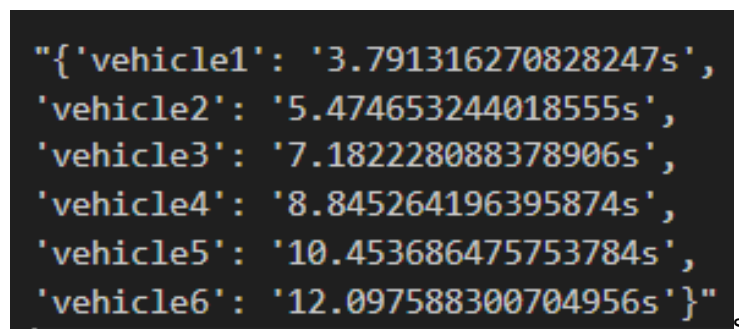
în momentul în care un vehicul parcurge toate marker-ele de traseu atribuite și de a verifica dacă informațiile transmise prin API sunt conforme cu starea reală a vehiculelor.

### 6.6.1 Testare cu mai multe vehicule

Au fost trimise la server teste cu un număr variabil de vehicule, de la 1 la 6, generate conform metodei descrise în secțiunea de generare imagini. Pentru fiecare caz, markerii frontali ai vehiculelor au fost poziționați astfel încât să coincidă cu markerii de traseu, simulând faptul că vehiculul a ajuns la poziția respectivă. Pozele astfel generate imită cu precizie cazuri reale, fără a fi nevoie de folosirea simultană a mai multor vehicule fizice.

După procesarea imaginilor, serverul a determinat corect, pe baza distanței dintre markerii vehiculelor și cei ai traseului, că toate vehiculele au ajuns pe ultimul marker din traseu. Conform logicii definite în sistem, un vehicul este considerat că a terminat traseul dacă markerul său frontal se află la o distanță mai mică decât jumătate din dimensiunea markerului față de markerul final de traseu. Acest lucru a fost respectat în toate cazurile testate.

După identificarea completării traseului pentru fiecare vehicul, serverul a actualizat timpii de finalizare și i-a pus la dispoziție prin API-ul dedicat, ca răspuns la cererea aplicației mobile.



```
"{'vehicle1': '3.791316270828247s',  
'vehicle2': '5.474653244018555s',  
'vehicle3': '7.182228088378906s',  
'vehicle4': '8.845264196395874s',  
'vehicle5': '10.453686475753784s',  
'vehicle6': '12.097588300704956s'}"
```

Figura 19: Timpii de terminare traseu pentru 6 vehicule

În Figura 19 se observă pachetul JSON primit de la server în cazul unei sesiuni cu 6 vehicule, fiecare identificat printr-un ID unic. Timpul de terminare este exprimat în secunde față de momentul de start al sesiunii și este diferit pentru fiecare vehicul, reflectând momentul în care fiecare dintre ele a fost detectat pentru ultima oară pe ultimul marker de traseu.

Acest rezultat confirmă că sistemul poate urmări starea individuală a fiecărui vehicul, poate detecta corect momentul terminării traseului și poate furniza aplicației informații coerente despre finalizarea sesiunii, inclusiv în scenarii cu mai multe vehicule active în paralel. Capacitatea sistemului de a scala și de a menține această acuratețe chiar și în condiții de încărcare mare (vehicule multiple, imagini trimise simultan) este un indicator important al robusteții soluției propuse.

## 6.6.2 Testare terminare traseu aplicație

Pentru a testa acest aspect direct cu aplicația, dar și pentru a valida logica de sortare finală a timpilor de finalizare afișați în interfața mobilă, au fost realizate două sesiuni complete cu vehicule RC reale într-un scenariu controlat. Aceste sesiuni au fost menite să verifice dacă aplicația primește, interpretează și afișează corect timpii de finalizare într-un clasament ordonat logic, în funcție de momentul ajungerii la ultimul marker.

Prima sesiune a constat în rularea simultană a două vehicule pe un traseu format din 4 markere, ambele vehicule pornind în același timp. Al doilea vehicul a avut o traiectorie mai directă și a reușit să finalizeze traseul înaintea primului. Sistemul a detectat corect ordinea de sosire, iar aplicația a ordonat timpii în consecință, afișând ID-ul vehiculului care a terminat primul în capul clasamentului, urmat de celălalt.

A doua sesiune a fost construită special pentru a testa comportamentul sistemului atunci când un vehicul nu finalizează traseul. Primul vehicul a fost lăsat să parcurgă toți markerii, iar al doilea a fost oprit intenționat la jumătatea traseului. Sistemul a semnalizat corect, prin timpul -1 returnat pentru vehiculul inactiv, faptul că acesta nu a terminat. Aplicația a interpretat valoarea și a plasat vehiculul respectiv la finalul clasamentului, considerându-l ultimul, indiferent de ordinea numerică a ID-urilor.

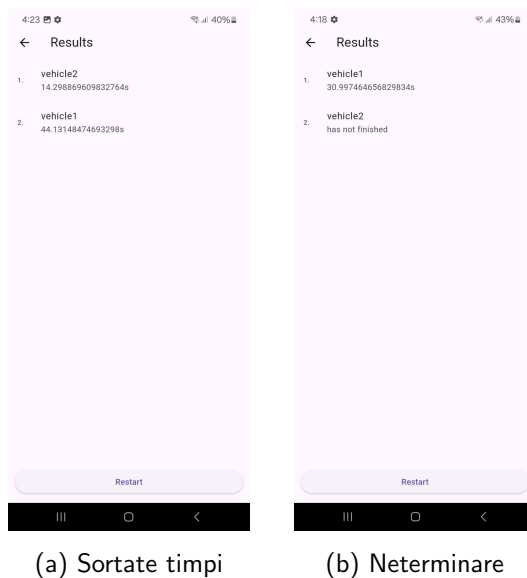


Figura 20: Timp Application

În Figura 20a se observă timpii primiți de aplicație, unde ambii roboți au finalizat traseul parcurs, iar timpii sunt ordonați în mod corect în funcție de valoarea minimă. Figura 20b prezintă cazul în care vehiculul 2 nu a finalizat traseul, iar aplicația îl identifică și afișează ca având timpul cel mai mare (-1).

Această testare demonstrează funcționarea corectă și completă a mecanismului de comunicare între server și aplicație, precum și coerența logicii de ordonare a rezultatelor.

## 7 CONCLUZII

Lucrarea de față propune o soluție modulară și scalabilă pentru localizarea și coordonarea vehiculelor RC utilizând markeri ArUco, procesare de imagine cu OpenCV, infrastructură containerizată și o aplicație mobilă dezvoltată în Flutter. Obiectivul principal a fost crearea unui sistem capabil să funcționeze în timp real, cu o precizie ridicată, într-un spațiu controlat, fără a necesita senzori dedicați pe vehicule sau soluții hardware costisitoare.

Implementarea a integrat mai multe componente interconectate:

- o aplicație mobilă responsabilă de captarea și trimiterea periodică a imaginilor către server, configurarea parametrilor sesiunii și afișarea rezultatelor;
- un server Flask containerizat, capabil să proceseze imaginile primite, să extragă poziții metrice din markerii detectați și să răspundă vehiculelor cu pozițiile actuale;
- o infrastructură orchestrală bazată pe Kubernetes, care permite autoscalarea podurilor de procesare și sincronizarea datelor între ele prin Redis;
- un protocol de evaluare și testare riguroasă a sistemului, care a inclus atât imagini generate, cât și imagini reale.

Testele efectuate au arătat că sistemul îndeplinește obiectivele inițiale. Precizia medie de poziționare a markerilor a fost de ordinul milimetrilor, timpul de procesare a fost redus considerabil prin scalare dinamică, iar comportamentul sistemului s-a menținut stabil chiar și în condiții de încărcare ridicată sau transmisie rapidă a cadrelor.

De asemenea, implementarea detectării finalizării traseului și gestionarea corectă a ordonării vehiculelor în funcție de timpii obținuți demonstrează funcționalitatea completă a arhitecturii propuse. Faptul că toate aceste componente au fost containerizate și orchestrate cu succes confirmă caracterul portabil și extensibil al soluției.

În concluzie, lucrarea prezintă o arhitectură viabilă pentru sisteme distribuite de localizare vizuală, cu aplicabilitate directă în domenii educaționale, de cercetare sau testare de prototipuri autonome. Sistemul poate fi extins cu ușurință pentru a include alte tehnologii de detecție, metode de planificare a traseului sau integrare cu senzori suplimentari, păstrând în același timp simplitatea și modularitatea esențială a designului.

## BIBLIOGRAFIE

- [1] Aryo Jamshidpey, Mostafa Wahby, Michael Allwright, Weixu Zhu, Marco Dorigo, Mary Katherine Heinrich. "Centralization vs. decentralization in multi-robot sweep coverage with ground robots and UAVs". <https://arxiv.org/abs/2408.06553>.
- [2] Khalil Mohamed, Ayman Elshenawy Elsefy, Hany M. Harb. "A Hybrid Decentralized Coordinated Approach for Multi-Robot Exploration Task". <https://doi.org/10.1093/comjnl/bxy107>.
- [3] Elena Ceseracciu, Zimi Sawacha, Claudio Cobelli. "Comparison of Markerless and Marker-Based Motion Capture Technologies through Simultaneous Data Collection during Gait: Proof of Concept". <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0087640>.
- [4] J.L. Pulloquina, D. Corrata, V. Mata, Á. Valera, M. Vallés. "Experimental Analysis of Pose Estimation Based on ArUco Markers". [https://doi.org/10.1007/978-3-031-61582-5\\_12](https://doi.org/10.1007/978-3-031-61582-5_12).
- [5] F. Lumpp, M. Panato, F. Fummi, N. Bombieri. "A Container-based Design Methodology for Robotic Applications on Kubernetes Edge-Cloud architectures". <https://doi.org/10.1109/FDL53530.2021.9568376>.
- [6] Santiago Royo, Maria Ballesta-Garcia. "An Overview of LiDAR Imaging Systems for Autonomous Vehicles". <https://www.mdpi.com/2076-3417/9/19/4093>.
- [7] Kai Dai, Bohua Sun, Guanpu Wu, Shuai Zhao, Fangwu Ma, Yufei Zhang, Jian Wu. "LiDAR-Based Sensor Fusion SLAM and Localization for Autonomous Driving Vehicles in Complex Scenarios". <https://www.mdpi.com/2313-433X/9/2/52>.
- [8] K. Rahnamai, K. Gorman, A. Gray, P. Arabshahi. "Formations of autonomous vehicles using Global Positioning Systems (GPS)". <https://doi.org/10.1109/AERO.2005.1559754>.
- [9] Yassine Zein, Mohamad Darwiche, Ossama Mokhiamar. "GPS tracking system for autonomous vehicles". <https://doi.org/10.1016/j.aej.2017.12.002>.
- [10] A. Mahmoud, A. Noureldin, H.S. Hassanein. "Integrated Positioning for Connected Vehicles". <https://doi.org/10.1109/TITS.2019.2894522>.
- [11] OpenCV Documentation. Camera Calibration with OpenCV. [https://docs.opencv.org/4.x/dc/dbb/tutorial\\_py\\_calibration.html](https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html)



- [12] OpenCV Documentation. Pose Estimation. [https://docs.opencv.org/4.x/d7/d53/tutorial\\_py\\_pose.html](https://docs.opencv.org/4.x/d7/d53/tutorial_py_pose.html)
- [13] OpenCV Documentation. Detection of ArUco Markers. [https://docs.opencv.org/4.x/d5/dae/tutorial\\_aruco\\_detection.html](https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html)
- [14] Flask documentation. *Flask Web Development*. Disponibil la: <https://flask.palletsprojects.com/en/stable/>
- [15] A. E. Fentaw. *Cross platform mobile application development: A comparison study of React Native Vs Flutter*. 2020. Disponibil la: <https://urn.fi/URN:NBN:fi:jyu-202006295155>
- [16] *Redis vs. Memcached in Microservices Architectures: Caching Strategies, International Journal of Multidisciplinary Research and Growth Evaluation*, vol. 4, no. 3, pp. 1084–1091, Jan. 2023. doi: 10.54660/IJMRGE.2023.4.3.1084-1091
- [17] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, *Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration*, *Sensors*, vol. 20, no. 16, p. 4621, 2020. doi: 10.3390/s20164621
- [18] Docker Documentation. "Build and push your first image". <https://docs.docker.com/get-started/introduction/build-and-push-first-image/>
- [19] Docker Documentation. „Building Docker images: Core concepts". <https://docs.docker.com/get-started/docker-concepts/building-images/>
- [20] A. Pereira Ferreira, R. Sinnott. "A Performance Evaluation of Containers Running on Managed Kubernetes Services," 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Sydney, NSW, Australia, 2019, pp. 199–208. <https://doi.org/10.1109/CloudCom.2019.00038>.