БАКАЛАВРСЬКА РОБОТА

БР.ІП – 56.00.00.000 ПЗ

Група ІП-22-2

Чудик Вероніка

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Чудик Вероніка Ігорівна

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Розробка програмного забезпечення системи автономної навігації з розпізнаванням візуальних міток та веб-інтерфейсом керування

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____Яцишин Микола Миколайович, к.т.н., доцент
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Чудик Вероніці Ігорівній

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Розробка програмного забезпечення системи автономної навігації з розпізнаванням візуальних міток та веб-інтерфейсом керування"

керівник проекту (роботи) Яцишин Микола Миколайович, к.т.н., доцент _____

затверджені наказом вищого навчального закладу від “ 21 ” квітня 2026 р. № 179/7

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Use Case діаграма системи автономного робота між користувачем і акторами (рис. 3.9, ст. 40)
2. Sequence діаграма сценарію для встановлення з'єднання (рис. 3.10, ст. 41)
3. Sequence діаграма сценарію для гри (рис. 3.11, ст. 43)
4. Activity діаграма для алгоритму логіки автономної навігації (рис. 3.12, ст. 45)
5. Component діаграма всіх елементів системи (рис. 3.13, ст. 46)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	17.01.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.02.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.03.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.05.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	09.06.2026	виконано

Студент _____.Чудик В.І.
(підпис)

Керівник роботи _____Яцишин М. М.
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 80 сторінок, 25 рисунків, 5 таблиць список використаних джерел із 24 найменувань, 2 додатки.

Мета роботи: дослідження можливості побудови повноцінної розподіленої IoT-архітектури на базі мікроконтролера ESP32-S3 шляхом проєктування та реалізації програмного забезпечення для керування автономним мобільним роботом у форматі інтерактивної гри, що поєднує робототехніку та Інтернет речей.

Об'єкт дослідження: взаємодія Інтернет речей (IoT), роботехніки, Full-Stack застосунку у реальному часі.

Предмет дослідження: компоненти та збірка апаратного забезпечення, протоколи передачі та обміну даних, побудова архітектури застосунку, дизайн інтерфейсу, тестування, забезпечення доступності та масштабованості системи.

Результати дослідження: спроєктовано, реалізовано та протестовано інтегровану IoT-платформу керування автономним мобільним роботом на базі ESP32-S3 із підтримкою WebSocket,

У першому розділі проаналізовано сучасний стан предметної області та розглянуто основні поняття.

У другому розділі сформульовано вимоги до системи та поставлено задачу проєктування.

У третьому розділі описано архітектурні підходи та обґрунтовано вибір технологій і інструментів.

У четвертому розділі подано опис процесу реалізації проєкту.

У п'ятому розділі наведено стратегії тестування, результати перевірки функціональності системи та варіанти її впровадження.

Висновок: розроблено інтегровану IoT-платформу керування автономним мобільним роботом на базі ESP32-S3,

КЛЮЧОВІ СЛОВА: ІНТЕРНЕТ РЕЧЕЙ, МОБІЛЬНИЙ РОБОТ, ESP32-S3, АВТОНОМНА НАВІГАЦІЯ, WEBSOCKET, РОЗПОДІЛЕНА АРХІТЕКТУРА, REACT NATIVE, ГЕЙМІФІКАЦІЯ, ВІДЕОПОТІК, FFMPEG,

ANNOTATION

The bachelor's thesis contains 80 pages, 25 figures, 5 tables, and a reference list containing 24 sources.

The aim of the work is to study software security testing methods for vulnerability detection, system security assessment, and improvement of modern software reliability.

Research object: processes of software security testing and assurance.

Research subject: methods, models, and tools of software security testing across different stages of the software development life cycle, including static, dynamic, and model-based testing approaches.

Research results: Research results: an integrated IoT platform for controlling an autonomous mobile robot based on ESP32-S3 with WebSocket support was designed, implemented and tested.

The first section analyzes the current state of the subject area and considers the main concepts

The second section formulates the requirements for the system and sets the design task.

The third section describes the architectural approaches and justifies the choice of technologies and tools.

The fourth section describes the project implementation process.

The fifth section presents testing strategies, results of system functionality testing and options for its implementation.

Conclusion: an integrated IoT platform for controlling an autonomous mobile robot based on ESP32-S3 has been developed.

KEYWORDS: INTERNET OF THINGS, MOBILE ROBOT, ESP32-S3, AUTONOMOUS NAVIGATION, WEBSOCKET, DISTRIBUTED ARCHITECTURE, REACT NATIVE, GAMIFICATION, VIDEO STREAMING, FFMPEG, MICROCONTROLLER,

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

IoT — Internet of Things (Інтернет речей)

ESP32-S3 — назва мікроконтролера (Espressif Systems 32-bit, серія S3)

IMU — Inertial Measurement Unit (інерційний вимірювальний модуль)

PSRAM — Pseudo Static Random Access Memory (псевдостатична оперативна пам'ять)

MJPEG — Motion JPEG (формат відеопотоку)

WebSocket — протокол двоспрямованого зв'язку

HTTP — HyperText Transfer Protocol

HTTPS — HyperText Transfer Protocol Secure

RTSP — Real Time Streaming Protocol

SRT — Secure Reliable Transport

WebRTC — Web Real-Time Communication

FFmpeg — Fast Forward MPEG (інструмент обробки медіапотоків)

GPIO — General Purpose Input/Output (порти введення-виведення загального призначення)

I2C — Inter-Integrated Circuit (послідовний інтерфейс зв'язку)

SCCB — Serial Camera Control Bus

ШІМ — широтно-імпульсна модуляція (PWM — Pulse Width Modulation)

EDA — Event-Driven Architecture (архітектура, керована подіями)

API — Application Programming Interface

URI — Uniform Resource Identifier

BMS — Battery Management System (система керування акумулятором)

ВСТУП

Актуальність теми: у сучасну епоху технологічної еволюції **Інтернет речей (IoT)** та **розумна роботехніка** стають критичними каталізаторами цифрової трансформації та дуже взаємопов'язанність тільки зростає. **Системи Інтернет-речей** – це фізично реальні системи і комплекси, функціонування яких базується на використанні величезної кількості датчиків, комп'ютерних і телекомунікаційних технологій, штучного інтелекту, хмарних обчислень, мережі Інтернет, застосування яких надає можливості за участю або без участі людей приймати і реалізовувати рішення, що створює умови для високоефективного, економного, раціонального здійснення будь-якої діяльності з мінімальним використанням ресурсів [1].

За визначенням, **розумна роботехніка** – це галузь автономних систем на базі штучного інтелекту, оснащені розширеними можливостями сенсорного сприйняття, прийняття рішень навчання та самонавчання. На відміну від своїх традиційних аналогів, розумні роботи можуть навчатися на власному досвіді, з часом розширювати свої навички та постійно покращувати свою продуктивність на основі набутих знань [2].

Мета роботи: дослідження можливості побудови повноцінної розподіленої IoT-архітектури на базі мікроконтролера ESP32-S3 шляхом проєктування та реалізації програмного забезпечення для керування автономним мобільним роботом у форматі інтерактивної гри, що поєднує робототехніку та Інтернет речей. Завдання дослідження: провести аналіз сучасних технологій та архітектурних підходів у сфері IoT і автономної робототехніки, визначити функціональні та нефункціональні вимоги до системи керування мобільним роботом, спроєктувати багаторівневу клієнт-серверну архітектуру системи, розробити програмне забезпечення для мікроконтролера ESP32-S3, серверної частини та мобільного застосунку, реалізувати передачу телеметричних і відеоданих у режимі реального часу, створити мобільний інтерфейс, провести тестування та оцінити ефективність розробленого рішення.

Об'єкт дослідження: взаємодія Інтернет речей (IoT), роботехніки, Full-Stack застосунку у реальному часі.

Предмет дослідження: компоненти та збірка апаратного забезпечення, протоколи передачі та обміну даних, побудова архітектури застосунку, дизайн інтерфейсу, тестування, забезпечення доступності та масштабованості системи.

Результати дослідження: спроєктовано, реалізовано та протестовано інтегровану IoT-платформу керування автономним мобільним роботом на базі ESP32-S3 із підтримкою WebSocket.

Методи дослідження: аналіз та узагальнення наукових джерел, методи системного аналізу проєктування, моделювання, експериментальне тестування,.

Наукова новизна роботи полягає у розробці та практичній реалізації інтегрованої IoT-платформи керування автономним мобільним роботом, яка поєднує edge-computing підхід на базі ESP32-S3, розподілену клієнт-серверну архітектуру, двосторонній обмін даними в режимі реального часу через WebSocket та гейміфікований мобільний інтерфейс, що забезпечує ефективну взаємодію користувача з роботизованою системою за обмежених апаратних ресурсів мікроконтролера.

Бакалаврська робота містить 80 сторінок, 25 рисунків, 5 таблиць, 5 розділів, список використаних джерел із 24 найменувань.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ТА ТЕОРЕТИЧНИХ ОСНОВ ПРОЄКТУВАННЯ ІНСТРУМЕНТІВ АНАЛІЗУ БЕЗПЕКИ ПРОГРАМНОГО КОДУ

1.1. Аналіз сучасного стану проблематики

Все більше IoT-пристроїв та систем з’являється з кожним днем, а разом з цим підвищується рівень їх складності, що, в свою чергу, обумовлює збільшення зусиль та часу на їхню підтримку й покращення зручності роботи з додатками. [7]. У зв’язку з експотенційним та лінійним розвитком технологій Інтернет речей з’являються певні технічні **виклики**.

Однією з найважливіших проблем є забезпечення **кібербезпеки**, оскільки кожен інтегрований вузол може слугувати потенційною точкою входу для кібератак. Зловмисники експлуатують вразливості в системі захисту для несанкціонованого доступу до персональних даних або перехоплення контролю над пристроєм. У певних сценаріях тисячі підключених об'єктів можуть бути використані для реалізації масштабних кібератак, зокрема розподілених атак типу "відмова в обслуговуванні" (DDoS).

Ще однією технічною перешкодою є управління та обробка даних. Підключені пристрої генерують колосальні обсяги інформації, що класифікуються як **"великі дані" (Big Data)**. Це зумовлює потребу в інфраструктурі та багато рівневій архітектурі, здатних здійснювати збереження та аналіз цих даних у режимі реального часу, що вимагає залучення потужних обчислювальних рішень.

Щодо **обчислювальних парадигм** у сфері Інтернету речей (IoT), то сюди належать **периферійні (edge computing)** та **хмарні (cloud computing)** обчислення [8].

Згідно з реальними Інтернет даними хмарні обчислення часто спричиняють затримки у 30–60 мілісекунд, периферійна обробка (edge processing) дозволяє отримати відповідь з локального сервера всього за 5–10

мілісекунд,що у 6 разів менше! Це пояснюється тим, що забезпечення обчислювальних потужностей безпосередньо на межі мережі дозволяє нівелювати ряд обмежень хмарних моделей, зокрема щодо пропускної здатності каналів зв’язку, латентності (затримки) та стабільності з’єднання [9].
Саме тому однією із найважливіших переваг мого рішення є впровадження методів edge computing та розподілена архітектура.

Іншим вагомим викликом в архітектурі IoT є **інтероперабельність**. IoT-середовища складаються з пристроїв різних виробників, що використовують різноманітні комунікаційні протоколи та формати даних. Забезпечення безперешкодної взаємодії між цими гетерогенними компонентами залишається складним завданням. Для вирішення проблем інтероперабельності часто застосовуються рішення на базі проміжного ПЗ та стандартизовані API, проте вони вносять додаткові витрати ресурсів та складність.Крім того, обробка даних у реальному часі створює виклики для продуктивності через високу швидкість та обсяг даних. Зі зростанням кількості підключених пристроїв бекенд-системи повинні ефективно обробляти потокові дані з мінімальною латентністю. Це зумовлює необхідність використання асинхронних, подієво-орієнтованих моделей обробки та масштабованих баз даних. Енергоефективність та обмеженість ресурсів пристроїв також відіграють критичну роль, особливо для IoT-пристроїв із живленням від батарей, що робить використання полегшених протоколів зв'язку та оптимізованої передачі даних надзвичайно важливим.

Автономні системи та робототехніка дедалі більше стають невід'ємною частиною різних секторів, зокрема охорони здоров'я, сільського господарства, виробництва та транспорту. Ці системи призначені для виконання завдань без участі людини, використовуючи досягнення штучного інтелекту , машинного навчання та сенсорних технологій. Однак розробка цих систем пов'язана з численними викликами, які необхідно вирішити для забезпечення їх ефективного та безпечного впровадження.

Одним із основних технічних викликів при розробці автономних систем є **злиття даних із сенсорів(sensor fusion)** та локалізація. Автономні мобільні

роботи покладаються на різні сенсори для навігації та виконання завдань. Інтеграція даних із кількох сенсорів для формування цілісного розуміння навколишнього середовища є складним процесом, схильним до помилок. Ефективні методи злиття сенсорних даних є необхідними для точної локалізації, оцінки стану та навігації. Роботи повинні навчатися та адаптуватися до свого середовища, щоб ефективно виконувати завдання. Реальний світ є надзвичайно мінливим, і неможливо заздалегідь запрограмувати робота так, щоб передбачити кожен можливий сценарій.

Методи машинного навчання є ключовими для того, щоб роботи могли маніпулювати об'єктами та взаємодіяти з навколишнім середовищем. Проте розробка надійних алгоритмів навчання, здатних справлятися з великою мінливістю реального світу, залишається значним викликом. Також крім складності алгоритмізації, навчання автономних роботів потребує колосального обсягу даних. Для того щоб робот міг ефективно функціонувати в реальному світі, необхідно зібрати, обробити та структурувати мільйони прикладів різноманітних сценаріїв. Це створює величезне навантаження як на обчислювальні ресурси, так і на інфраструктуру зберігання даних.

Енергоспоживання залишається одним із найбільших викликів у розробці IoT-продуктів. Багато пристроїв працюють у віддалених умовах, де часта заміна батарей є непрактичною, тому інженери приділяють значну увагу

проєктуванню систем із мінімальним енергоспоживанням. Основні стратегії включають: подієво-орієнтовану архітектуру прошивки, мікроконтролери та сенсори з низьким енергоспоживанням, оптимізовані інтервали бездротової передачі, а також розширені режими сну та управління живленням.

1.2. Основні поняття та визначення інженерії ПЗ в даному домені

Щоб зрозуміти, як розумні пристрої взаємодіють та функціонують разом, важливо вивчити архітектуру Інтернету речей (IoT). Вона визначає, як датчики,

мерми. Архітектура Інтернету речей складається з чотирьох різних рівнів, а саме: рівня чутливості, мережевого рівня, рівня обробки даних та рівня додатків [10]. Системи Інтернету речей (IoT) зазвичай мають трирівневу структуру: рівень пристроїв, мережевий рівень та прикладний рівень. На рівні пристроїв сенсори та актуатори використовуються для захоплення та передачі даних. Мережевий шар забезпечує комунікацію пристроїв із бекендом, а прикладний шар обробляє дані для створення візуалізацій для кінцевих користувачів.

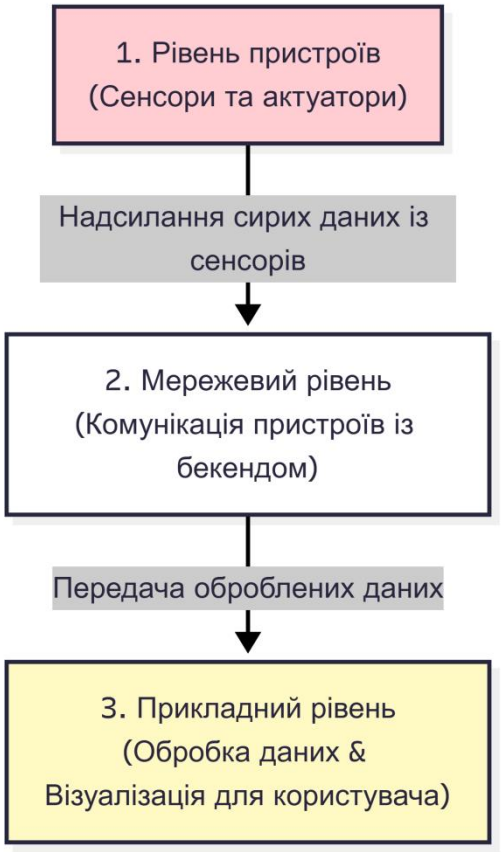

Рисунок 1.1 - Трьох рівнева архітектура системи IoT

На додаток до базової трирівневої архітектури (рисунок 1.1), сучасні IoT-системи часто впроваджують розширені моделі, такі як чотири- або п'ятирівневі структури, для підвищення модульності та керованості системи. Ці додаткові рівні зазвичай включають обробку даних, проміжне ПЗ (middleware) та рівень бізнес-логіки, що сприяє фільтрації сирих даних із сенсорів, виконанню аналітики та управлінню комунікацією між гетерогенними пристроями. Така

багаторівнева архітектура забезпечує кращу масштабованість та відмовостійкість, особливо у великомасштабних розгортаннях IoT.

Архітектура Інтернет речей (IoT) фундаментально починається з програмування мікроконтролеру зчитує та відноситься до парадигми Embedded Systems і Embeded Development**. Embedded Development -** це напрям, де програмування виходить за межі інтерфейсів і починає керувати фізичним світом: мікроконтролерами, сенсорами, транспортом, промисловим обладнанням і критичною інфраструктурою. Невід'ємною частиною любої embedded системи є **мікроконтролер**. Мікроконтролер в інтернеті речей (IoT) та в роботехніці найчастіше прирівнюють до «крихітного мозку» або центрального процесора пристрою. Він забезпечує збір даних із сенсорів, їх обробку та передачу через мережу, виступаючи інтелектуальним центром, який керує поведінкою фізичного об'єкта. Я обрала **ESP32-S3-CAM WROOM (РИСУНОК 1.2)**, що є платою з підтримкою WIFI/Blutetooth, побудована на базі мікроконтролера ESP32. Завдяки своїй універсальності та доступності, **ESP32-S3-CAM WROOM** здобув значну популярність серед розробників та професіоналів для реалізації широкого спектра рішень, включаючи проєкти в галузі Інтернету речей (IoT), систем домашньої безпеки та цифрової обробки зображень. **ESP32-S3-CAM WROOM** має модуль камери (зазвичай моделі OV2640). Завдяки вбудованим інтерфейсам Wi-Fi та Bluetooth, ESP32-CAM забезпечує можливість бездротової передачі зображень та потокового відео. Компактні габарити, низька вартість та висока обчислювальна потужність роблять цей модуль оптимальним вибором для проєктів, що потребують захоплення та трансляції візуальних даних, таких як відеоспостереження, розпізнавання облич та дистанційний моніторинг.

Ключові характеристики та технічні специфікації:

Мікроконтролер:ESP32-S3(XtensaLX7).

Процесор: двоядерний 32-бітний процесор із тактовою частотою до 240 МГц з апаратним прискоренням векторних операцій, що робить його придатним для задач машинного навчання та обробки зображень безпосередньо на пристрої.

Модуль камери (OV2640): забезпечує захоплення зображень із роздільною здатністю до 1600×1200 пікселів (UXGA). Підтримує стиснення у форматі JPEG, що дозволяє суттєво економити пропускну здатність каналу при бездротовій передачі.

Оперативна пам'ять: 512 КБ SRAM + 8 МБ зовнішньої PSRAM, що забезпечує достатній обсяг для буферизації відеопотоку та виконання алгоритмів обробки даних у реальному часі.

Флеш-пам'ять: 8 МБ використовується для зберігання вбудованого програмного забезпечення (firmware), файлових систем та ресурсів, необхідних для функціонування камери.

Wi-Fi: IEEE 802.11 b/g/n (2.4 ГГц).

Bluetooth: BLE 5.0 + Bluetooth v4.2.

Формати виводу зображень: JPEG, BMP, GRAYSCALE, RGB565.

Інтерфейси: слот для SD-карти (до 4 ГБ) для додаткового зберігання даних.

Робоча напруга: 3.3 В.

Габаритні розміри: 25.5 × 18 мм, один із найкомпактніших модулів із вбудованою камерою та Wi-Fi.

Слот для SD-карти: дозволяє зберігати медіадані безпосередньо на зовнішньому носії, що робить пристрій автономним у питаннях накопичення великих обсягів інформації.

GPIO-контакти: до 45 ліній введення-виведення загального призначення, що суттєво розширює можливості підключення периферійних датчиків та керування виконавчими пристроями порівняно з попередніми версіями модуля. **Апаратне прискорення:** вбудований блок AI/ML прискорення (Vector Instructions) для виконання нейромережевих операцій на рівні пристрою. **Варіанти антени:** модифікації з роз'ємом IPEX для підключення зовнішньої антени, що значно покращує дальність та стабільність сигналу.

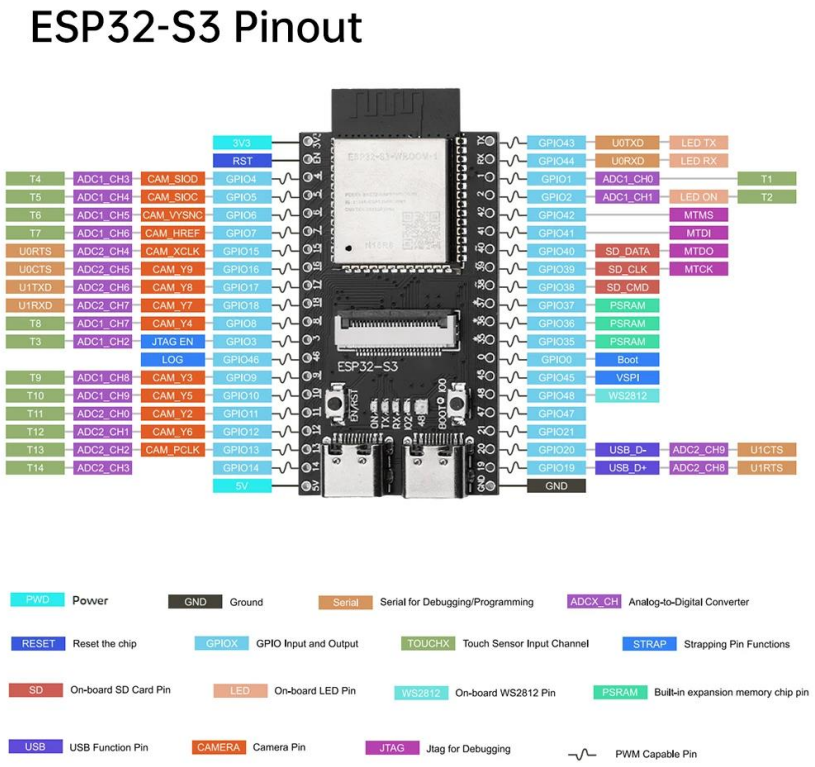

Рисунок 1.2 - Розіновка пінів ESP32S3CAM WROOM

Мікроконтолер залежить від **сервера** або хмарного середовища, оскільки більшість розумних пристроїв розроблені так, щоб бути компактними, недорогими та енергоефективними. Їм бракує апаратних ресурсів, ємності акумулятора та обчислювальної потужності, щоб самостійно зберігати великі обсяги даних, розміщувати веб-сторінки або безпечно обробляти необроблені дані. Одним з найкращих архітектурних рішень и виборі технології, на якій буде сервер є **NodeJs**. Node.js побудований на JavaScript-рушії V8 від Google, який є відкритим і широко відомий своєю ефективністю та масштабованістю. Завдяки цьому він добре підходить для навантажених застосунків реального часу. Оскільки IoT-застосунки також інтенсивно працюють з даними в режимі реального часу, вони природно поєднуються з цією технологією. Крім того, більшість IoT-застосунків використовує швидкі протоколи MQTT та WebSocket, які добре підтримуються Node.js, а менеджер пакетів NPM містить корисні модулі для IoT, зокрема понад 80 пакетів для Intel IoT Edison, Raspberry Pi та

Arduino, а також понад 30 пакетів для різних пристроїв, сенсорів і Bluetooth, що суттєво прискорює розробку. Оскільки IoT-пристрої генерують великі обсяги даних і запитів, Node.js чудово справляється з їх обробкою завдяки підтримці потоків, які забезпечують керування запитами та тимчасове зберігання даних.

Система IoT потребує додатку як посередника між апаратними сенсорами та користувачами. Без прикладного рівня дані, зібрані пристроями, є нечитабельними, що унеможливлює дистанційне керування обладнанням, моніторинг аналітики або отримання автоматичних сповіщень. Зв'язка **React Native та Node.Js** (рисунок 1.3) давно стало стандартним підходом розробки комплексних мобільних додатків, де фронтенд та бекенд розділені за ролями. Node.js використовується як серверна платформа, що надає API, а React Native - популярний фреймворк для написання мобільних програм на JavaScript з можливістю реального нативного рендерингу інтерфейсу. Зазвичай архітектура інтеграції між Node.js та React Native виглядає так: Node.js працює на сервері, надає публічний HTTP API (REST або GraphQL) та підтримує постійне з'єднання через WebSocket або інші методи обміну. React Native запущений на пристрої, наприклад, телефоні, і виконує запити до серверного API через HTTP.

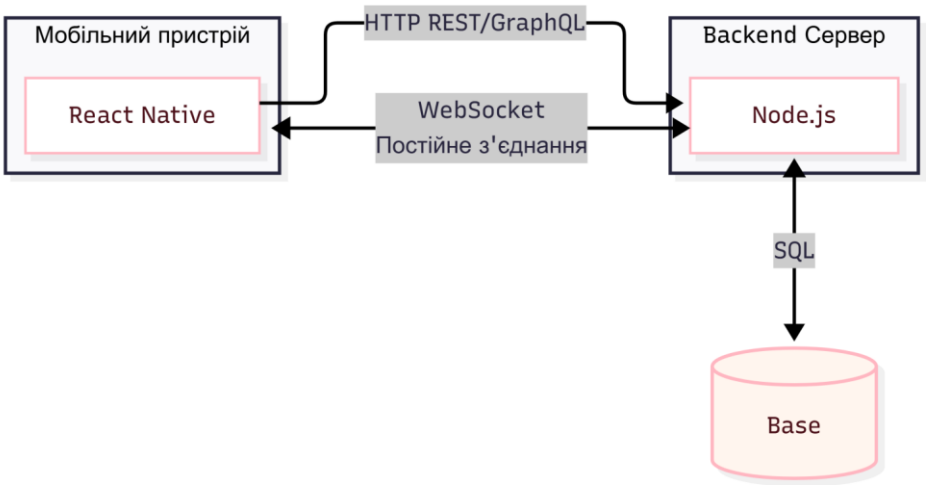

Рисунок 1.3 - Діаграма архітектура між Node.js та React Native

React Native із коробки надає можливість запускати React-код на нативних платформах iOS та Android. Разом із ним надаються найбільш фундаментальні

компоненти: Text, View і TextInput. Однак існує величезна кількість функціоналу, який потрібна майже всім production-додаткам, але не входить до React Native: навігація, надсилання push-сповіщень, використання камери телефону, збереження даних між запусками. Це зроблено навмисно: якби всі нативні можливості були доступні одразу, React Native став би непідтримуваним для невеликої React-команди в Meta.

Оскільки ключова функціональність не входить до ядра, розробники на React Native завжди змушені були покладатися на різноманітні бібліотеки від спільноти і саме в цьому допомагає **Expo**. Expo - це фреймворк для React Native: набір інструментів і сервісів, що заповнює прогалину між тим, що постачається з React Native, і всім іншим, необхідним для створення production-готових застосунків.

Фреймворк у мобільній розробці буде надійним рішенням та хорошим підходом, що забепечить набір програмних інструментів та бібліотек, яке надасть стандартизовану основу для створення додатків. Він значно спрощує процес розробки, оскільки містить готові компоненти та шаблони, які можна використовувати для різних аспектів мобільного додатку [11].

Важливою тахнологією, на яку покладається весь проект це передача та зчитування відео-даних на сервері Node.Js завдяки **FFMPEG**. **FFmpeg** - це провідний мультимедійний фреймворк, призначений для професійної роботи з відео та аудіоданими, а також для організації потокового мовлення. Можливості FFmpeg дозволяють здійснювати декодування, кодування, транскодування, мультиплексування (mux), демультиплексування (demux), стрімінг, фільтрацію та відтворення практично будь-яких медіаформатів. Завдяки своїй багатофункціональності, FFmpeg є універсальним інструментом для передачі медіаконтенту [12].

1.3. Сучасні технології та тенденції, що застосовуються в даній області

Зараз у центрі уваги вже не просто підключення пристроїв - а створення

інтелектуальних, масштабованих і надійних систем, які автономно функціонують у реальних умовах.

Підключені пристрої зазвичай працюють за простим алгоритмом: збирають дані, надсилають їх у хмару, чекають на вказівки. Це працює тільки до того моменту, доки обсяги даних не стали надто великими, мережі - ненадійними, а рішення - надто терміновими, щоб така затримка була прийнятною. Однією з сучасних тенденцій є **Інтернет речей штучного інтелекту(AIoT)**. У підході AIoT замість того, щоб спочатку надсилати дані до хмари, процес «мислення» відбувається на самому пристрої. Датчик або контролер виявляє проблему за мілісекунди та запускає вимкнення або надсилає сповіщення без інтернету та очікування. Хмара все ще відіграє роль, але іншу: вона збирає дані з усіх пристроїв, навчається на основі шаблонів та періодично надсилає розумні оновлення назад на пристрої [13].

Типова структура IoT включає сенсорні технології, передачу даних та хмарні обчислення. Хоча хмара дозволяє обробляти та зберігати величезні обсяги даних, завантаження всієї інформації може перевантажити пропускну здатність мережі та викликати занепокоєння щодо безпеки даних. Швидке зростання IoT та ШІ призводить до зростання проблем для **хмарних обчислень**, головним чином через надмірне навантаження даних. **Периферійні обчислення** стали потужним рішенням для підвищення продуктивності, зменшення затримки та покращення безпеки даних шляхом перенесення деяких завдань обробки з хмари. Оскільки пристрої IoT стають більш інтелектуальними та ресурсоємними, покладаючись виключно на хмарні обчислення, часто призводить до затримок передачі, особливо в чутливих до часу додатках. Обробляючи дані ближче до їх джерела, периферійні обчислення зменшують ці затримки та привернули значну увагу в останніх дослідженнях [14].

Гібридні архітектури підключення відображають один із ключових сучасних трендів у IoT: жодна окрема технологія зв'язку не може задовольнити всі вимоги реального розгортання, тому сучасні IoT-системи все частіше поєднують кілька протоколів одночасно. Наприклад, Wi-Fi для швидкої передачі

даних, Bluetooth для локальної взаємодії, та MQTT або WebSocket для комунікації з сервером [15].

1.3. Висновки по розділу

Аналіз сучасних тенденцій у сфері IoT демонструє, що галузь еволюціонує від простих пристроїв збору даних до комплексних інтелектуальних систем із гібридною архітектурою, автономною навігацією та адаптивним керуванням. Ключовими напрямами розвитку є гібридні моделі підключення, що поєднують декілька протоколів зв'язку, інтеграція алгоритмів машинного навчання безпосередньо на рівні пристрою, а також оптимізація енергоспоживання як на апаратному, так і на програмному рівні.

Розроблена у межах даної роботи система повністю відповідає цим тенденціям: вона реалізує гібридну архітектуру керування з двома режимами роботи, використовує ESP32-S3 як edge-пристрій із можливістю локальної обробки даних, забезпечує передачу даних у реальному часі через WebSocket, а також передбачає інтеграцію алгоритмів злиття сенсорних даних на основі IMU та ультразвукових датчиків. Окремої уваги заслуговує гейміфікований інтерфейс як інноваційний підхід до взаємодії людини з роботизованою системою, що робить керування інтуїтивним.

Таким чином, обрані технології та архітектурні рішення не лише відповідають сучасним вимогам галузі, але й формують основу для подальшого масштабування системи з використанням штучного інтелекту та розширеної сенсорної інтеграції.

РОЗДІЛ 2. АНАЛІЗ ВИМОГ І ПОСТАНОВКА ЗАДАЧІ

2.1. Збір та аналіз вимог користувачів

У розробці програмного забезпечення аналіз вимог зосереджується на завданнях, що визначають потреби або умови, яким має відповідати новий або змінений продукт чи проєкт [16].

Вимоги визначають, що повинна робити система, та обмеження, за яких вона повинна функціонувати. Те, що повинна робити система, належить до категорії **функціональних вимог**, тоді як обмеження описуються **нефункціональними вимогами**. Наприклад, для системи домашнього банкінгу функціональні вимоги включають такі функції, як звітування про баланс рахунку, обробка переказів між рахунками, виконання банківських платежів та скасування дебетових карток. Натомість, нефункціональні вимоги пов'язані з атрибутами якості системи, включаючи продуктивність(кількість транзакцій за секунду, час відгуку, затримка), доступність, безпеку, портативність(здатність програмного забезпечення працювати в різних операційних системах), конфіденційність, використання пам'яті та диска, серед іншого. По суті, нефункціональні вимоги стосуються операційних обмежень [17].

Оскільки система розроблялась як авторський проєкт, вимоги формувались на основі власного досвіду розробки та аналізу потреб цільової аудиторії. В процесі проєктування було визначено три основні групи потенційних користувачів: ентузіасти електроніки та робототехніки, які мають власні пристрої на базі ESP32-S3 і прагнуть розширити їх функціональність; початківці у сфері IoT, для яких важлива інтуїтивність інтерфейсу та низький поріг входу; а також користувачі без технічного background'у, які зацікавлені передусім в ігровому аспекті системи. Отже Use-case діаграма для інтефейсу гри буде мати вигляд як на рисунку 2.1.

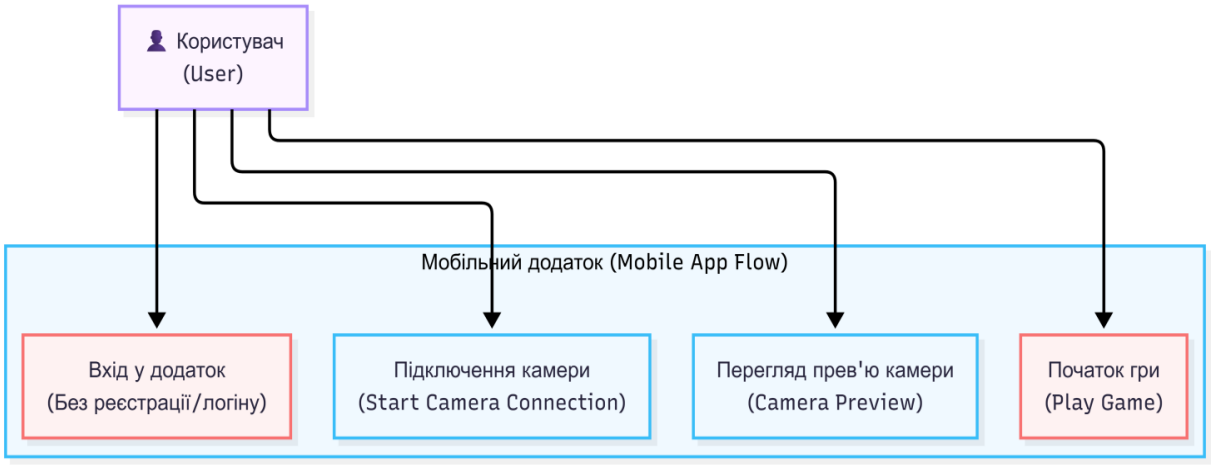

Рисунок 2.1 - Use-case діаграма для інтефейсу гри

2.2. Формулювання функціональних та нефункціональних вимог

Щоб розробити функціональні умови, потрібно почати з user stories(історій користувачів) таблиця 2.1. Я, як один з користувачів хочу підключитись до мікроконтролера через додаток, бачити прев'ю камери та розуміти стабільність чи нестабільність з'єднання, задавати рух пристрою через команди, переключатись між ручним та автономним режимом, грати гру.

Тому **функціональні вимоги** можуть бути такі:

1. Можливість керування роботом у реальному часі через мобільний застосунок
2. Перемикання між ручним та автономним режимами роботи
3. Інтерфейс аркадної піксельної гри із системою XP та штрафами за зіткнення з перешкодами
4. Автоматичне уникнення перешкод в автономному режимі.
5. Передача команд руху вправо, вліво, прямо.
6. Можливість бачити відео з камери на своєму пристої

Нефункціональні вимоги визначають не що система робить, а як вона це робить, тобто характеристики якості, надійності та продуктивності, які безпосередньо впливають на досвід користувача. У системах реального часу,

таких як керування роботом, ці вимоги є критично важливими, оскільки навіть незначні затримки можуть призвести до некоректної поведінки пристрою або втрати керування.

Тому визначено такі **нефункціональні вимоги**:

1. Час підключення до відеопотоку камери не повинен перевищувати 5 секунд, у протилежному випадку це свідчить про проблеми з мережею або з апаратним забезеченням.
2. Затримка між відправкою команди керування та фактичним рухом робота не повинна перевищувати 1 секунди
3. Перемикання між ручним та автономним режимами має відбуватись не довше ніж за 2 секунди
4. Час прийняття рішення в автономному режимі (сканування лівого та правого напрямків, обчислення та вибір маршруту) не повинен перевищувати 3 секунди
5. Навігація між сторінками застосунку, підключення до відеопотоку та перехід до ігрового режиму мають відбуватись миттєво, без відчутних затримок

Таблиця 2.1

Функціональні та нефункціональні вимоги

Функціональна	Відображення відеопотоку з камери у реальному часі	Високий
Функціональна	Сканування QR-кодів та нарахування балів	Середній
Функціональна	Передача команд через WebSocket	Високий
Нефункціональна	Час підключення до відеопотоку ≤ 5 секунд	Високий
Нефункціональна	Затримка команди керування ≤ 2 секунди	Високий
Нефункціональна	Перемикання режимів ≤ 3 секунди	Високий
Нефункціональна	Час прийняття рішення в автономному режимі ≤ 3 секунди	Середній
Нефункціональна	Кросплатформеність (Android та iOS)	Середній
Нефункціональна	Миттєва навігація між екранами застосунку	Середній

Таблиця 2.2

Характеристика проєктів та їх економічні показники

Проект	Необхідні навички	Прибуток	Бюджет
MVP робот (поточна версія)	ESP32, C++, React Native, Node.js, WebSocket, Python	$3,000	$500
Продуктова версія	LiDAR, комп'ютерний зір, sensor fusion, ML, 3D-друк	$15,000	$4,500
Хмарна платформа	DevOps, хмарна інфраструктура, API, мобільна розробка	$25,000	$8,000

2.3. Постановка задачі проєктування системи

На основі зібраних та проаналізованих вимог сформульовано комплексну задачу проєктування системи, яка охоплює апаратну складову, програмну архітектуру,обов’язково розробку власного алгоритму для оминання перешкод та інтерфейс.

Загальна мета полягає у створенні інтегрованої IoT-платформи керування мобільним роботом із двома режимами роботи: ручним(manual) та автономним(smart), із інтерфейсом аркадної піксельної гри, навігацією на основі злиття сенсорних даних у межах єдиної розподіленої архітектури реального часу.

Апаратну основу системи складає мікроконтролер **ESP32-S3-WROOM** із вбудованою камерою, який виконує роль центрального обчислювального та комунікаційного центру. Для виявлення перешкод у трьох напрямках використовуються три **ультразвукові датчики відстані HC-SR04**, . Керування приводами здійснюється через драйвер моторів **DRV8800**. Живлення системи забезпечується трьома літієвими акумуляторами напругою 12В із системою захисту **BMS** та **DC-DC** перетворювачем 12В -> 5В для стабілізації напруги електронних компонентів. Задача проєктування архітектури передбачає розробку розподіленої локальної системи, у якій усі обчислювання виконуються у межах однієї мережі. ESP32-S3 виступає edge-пристроєм, Node.js сервер забезпечує координацію та маршрутизацію команд, Python скрипт на локальному комп’ютері виконує ресурсоємні обчислення з обробки та злиття сенсорних даних, а React Native застосунок є точкою взаємодії з користувачем. Комунікація

між усіма компонентами здійснюється через протокол **WebSocket**, що забезпечує необхідну швидкість передачі даних у реальному часі.

Рисунок 2.2 - Технічні завдання

Задача проєктування прошивки полягає у розробці firmware для ESP32-S3, яка забезпечує паралельну обробку WebSocket-команд, керування драйвером моторів, зчитування даних із сенсорів та трансляцію відеопотоку з мінімальною затримкою та енергоспоживанням.

2.4. Висновки по розділу

У другому розділі було проведено збір та аналіз вимог до системи, визначено цільову аудиторію, сформульовано функціональні та нефункціональні вимоги, а також здійснено повну постановку задачі проєктування. Аналіз показав, що система має охоплювати широкий спектр технічних задач: від електроніки та розподіленої архітектури до створення алгоритмів.

РОЗДІЛ 3. ПРОЄКТУВАННЯ СИСТЕМИ

3.1. Розробка архітектури програмного забезпечення

Вибір правильного архітектурного стилю має вирішальне значення для успішного впровадження систем Інтернету речей (IoT). У зв'язку з тим, що IoT стрімко трансформує галузі та повсякденне життя, розуміння того, які архітектурні стилі найкраще відповідають конкретним потребам, є важливішим, ніж будь-коли. Архітектурні стилі в системах IoT допомагають абстрагувати складнощі підключення різноманітних пристроїв та управління потоком даних у динамічних середовищах, забезпечуючи ефективну роботу системи та відповідність вимогам до якості, таким як доступність, масштабованість та безпека. Кожен архітектурний стиль має свої сильні та слабкі сторони, що робить його придатним для конкретних випадків використання. Наприклад, деякі стилі оптимізовані для обробки даних у реальному часі та високої масштабованості, тоді як інші надають пріоритет безпеці та конфіденційності.

1.**Архітектура, керована подіями (Event-Driven Architecture)** рисунок 3.1, зосереджена на генеруванні, виявленні, обробці та реагуванні на події в режимі реального часу. У сфері Інтернету речей (IoT) EDA використовується для створення інтелектуальних систем, які реагують на події, що генеруються пристроями та датчиками. Ці події обробляються механізмами обробки подій, що дозволяє системі миттєво реагувати на зміни в навколишньому середовищі[18]. Розглянемо систему промислового моніторингу на виробництві. Температурний датчик фіксує перевищення допустимого порогу і генерується подія temperature_exceeded. На цю подію підписані незалежні обробники: система охолодження активується автоматично, модуль логування записує показник із часовою міткою, а сервер сповіщень надсилає тривогу оператору. Жоден із обробників не знає про існування інших, вони лише реагують на спільну подію через шину повідомлень.

З прикладу можемо сказати, що така система забезпечує високу продуктивність, масштабованість та ефективний моніторинг, дозволяючи одночасно обробляти декілька запитів. Однак її тестування може бути складним через її асинхронний характер та потенційне збільшення часу відгуку .

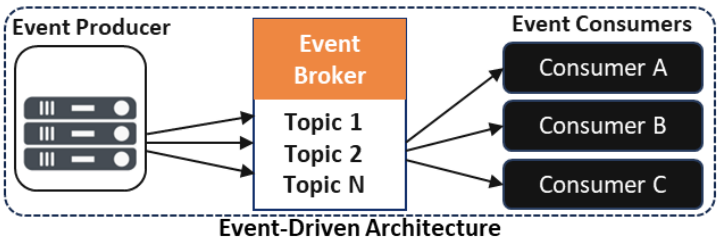

Рисунок 3.1 - Діаграма архітектури, керована подіями

2. **Хмарна архітектура (Cloud-Based Architecture)** рисунок 3.2 використовує хмарні обчислення як основну частину своєї обчислювальної інфраструктури, включаючи такі компоненти, як хмари, пристрої та консолі управління користувачами. Ця архітектура добре підходить для обробки значних обсягів даних у системах IoT, забезпечуючи обробку та зберігання даних у реальному часі. Вона дає масштабованість, гнучкість та адаптивність у різних сценаріях використання Інтернету речей. Однак вимагає надійного підключення до Інтернету, а проблеми з підключенням можуть призвести до зниження продуктивності. Крім того, конфіденційність та безпека даних є серйозними проблемами, оскільки дані зберігаються та обробляються поза межами локальної мережі [18]. Прикладом може бути система моніторингу розумного міста. Тисячі датчиків якості повітря, встановлених по всьому місту, безперервно передають показники на хмарну платформу. Хмарний брокер повідомлень приймає вхідні дані та розподіляє їх між сервісами: модуль аналітики обробляє агреговані показники та виявляє аномалії, сховище даних зберігає історію вимірювань, а API надає доступ до актуальних даних через мобільний застосунок для громадян та адміністративну панель для міської влади.

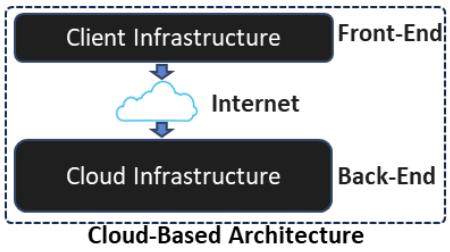

Рисунок 3.2 - Діаграма хмарної архітектури

3. **Архітектура «публікація-підписка» (Publisher-Subscriber)** складається з видавців та підписників. Видавці надсилають повідомлення, класифіковані за темами, не знаючи конкретних підписників, які отримують повідомлення, що відповідають їхнім інтересам [18]. Можна оглянути систему моніторингу розумного будинку. Датчик температури публікує показники у канал home/temperature. Його єдина відповідальність полягає у генерації та публікації даних, тоді як наявність та кількість підписників є для нього повністю прозорою. На цей канал незалежно підписані термостат, який автоматично регулює опалення, модуль логування, який зберігає історію показників, та мобільний застосунок мешканця, який відображає поточну температуру. Додавання нового підписника,наприклад, системи вентиляції не потребує жодних змін ані у логіці датчика, ані в існуючих підписниках. Ця архітектура зменшує взаємозалежність між компонентами, підвищуючи продуктивність та гнучкість системи. Однак вона може страждати від підвищеної затримки та складності через проміжні події, що маршрутизують повідомлення, що може ускладнити налагодження та обробку помилок. Проблеми з доступністю можуть виникнути у разі збою служб подій.

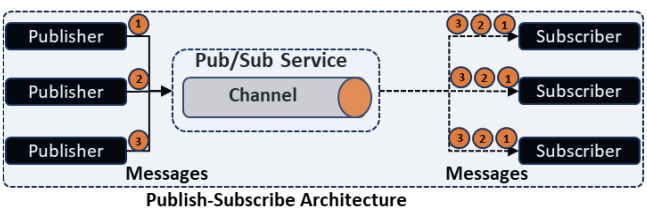

Рисунок 3.3 - Діаграма архітектури Publisher-Subscriber

4. **Архітектура мікросервісів (Microservice Architecture)** рисунок 3.4 є еволюцією СОА(сервісно-орієнтованою архітектурою) і складається з невеликих, слабко зв'язаних сервісів, які взаємодіють за допомогою легких протоколів. Кожен мікросервіс виконує унікальну функцію, що дозволяє командам розробляти та розгортати сервіси незалежно[18]. Є платформа розумного транспорту. Система розділена на незалежні мікросервіси: сервіс геолокації отримує GPS-координати від транспортних засобів, сервіс маршрутизації обчислює оптимальні шляхи, сервіс сповіщень надсилає оновлення пасажирам, а сервіс аналітики агрегує дані про завантаженість маршрутів. Кожен сервіс розгортається, масштабується та оновлюється незалежно , а збій у сервісі сповіщень жодним чином не впливає на роботу геолокації чи маршрутизації.Ця архітектура підтримує великомасштабні системи IoT, підвищуючи стійкість, масштабованість та гнучкість.

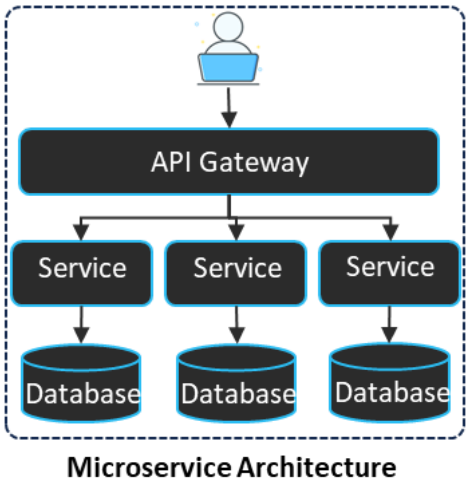

Рисунок 3.4 - Діаграма архітектури мікросервісів

5. **Архітектура REST (Representational State Transfer)** рисунок 3.5- це стиль побудови веб-сервісів, який застосовується при проектуванні систем Інтернету речей (IoT). В архітектурі REST пристрої та додатки IoT взаємодіють через стандартизовані веб-протоколи, часто використовуючи для зв'язку протокол HTTP. REST-сервіси моделюють пристрої IoT як ресурси,

забезпечуючи єдиний інтерфейс для доступу до цих ресурсів та управління ними. Ця архітектура забезпечує масштабованість та можливість повторного використання завдяки своїй слабкій зв'язаності, коли компоненти можуть взаємодіяти без жорстких залежностей. REST також підтримує кешування, що підвищує ефективність за рахунок зменшення навантаження на сервери. Однак проектування RESTful API може бути складним, а через відкритий характер веб-протоколів при обробці конфіденційних даних виникають проблеми безпеки [18].

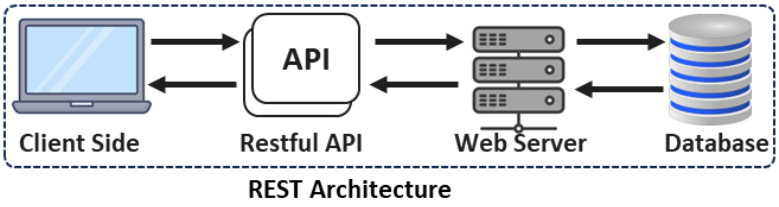

Рисунок 3.5 - Діаграма архітектури **REST (Representational State Transfer)**

6. **Архітектура «клієнт-сервер»** Рисунок 3.6 - це розподілена архітектура, в якій сервери надають ресурси або послуги, а клієнти їх запитують. В IoT сервери можуть виконувати завдання зберігання, обробки або моніторингу, тоді як клієнтами можуть бути різні пристрої IoT. Ця архітектура дозволяє здійснювати централізоване управління та легко модифікувати послуги. Однак при високих навантаженнях трафіку можуть виникати «вузькі місця» на серверах, що потенційно знижує продуктивність та доступність [18].

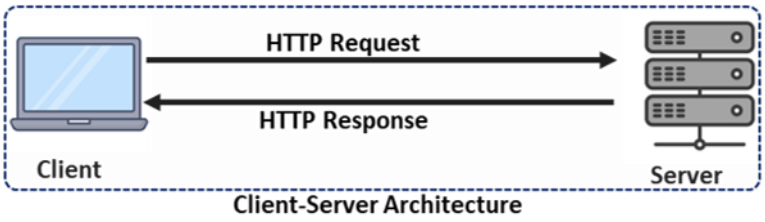

Рисунок 3.6 - Діаграма архітектури **REST (Representational State Transfer)**

7. Сервісно-орієнтована архітектура (SOA) рисунок 3.7 структурує програмне забезпечення у вигляді сервісів, придатних для повторного використання, які взаємодіють через стандартизовані інтерфейси. Кожен сервіс відповідає за конкретну функцію або її частину, що робить SOA придатною для різноманітних систем Інтернету речей (IoT). Вона сприяє модульності, слабкому зв'язку та незалежному керуванню пристроями IoT. Незважаючи на ці переваги, SOA може стикатися з проблемами, пов'язаними з використанням ресурсів, залежністю від окремих технологій та взаємодією між різними сервісами [18].

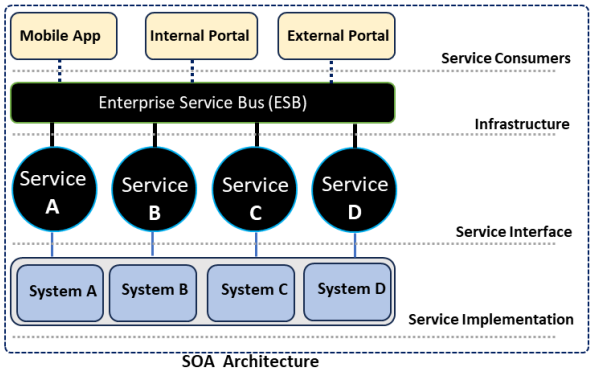

Рисунок 3.7 - Діаграма сервісно-орієнтованої архітектури

7. **Архітектура «пайпи і фільтрів»** (рисунок 3.8) складається з компонентів обробки (фільтрів), з'єднаних ланцюжками обробки даних, які послідовно обробляють дані. Ця архітектура ідеально підходить для систем IoT, що вимагають модульності та лінійної обробки даних, забезпечуючи масштабованість та адаптивність. Фільтри можна розробляти незалежно та повторно використовувати в різних системах. Однак ця архітектура менш придатна для інтерактивних додатків через її пакетний характер обробки, що може призвести до затримок та зниження продуктивності [18].Наприклад, система обробки даних із промислових датчиків якості води, де сирі показники з сенсорів надходять у першу трубу - фільтр очищення даних видаляє шумові викиди та некоректні значення. Очищені дані передаються у наступний фільтр нормалізації, який приводить показники різних датчиків до єдиної шкали. Далі

фільтр агрегації обчислює середні значення за часовими вікнами, після чого фільтр класифікації визначає рівень забруднення та присвоює відповідну мітку. На виході система отримує структуровані, готові до зберігання та аналізу дані.

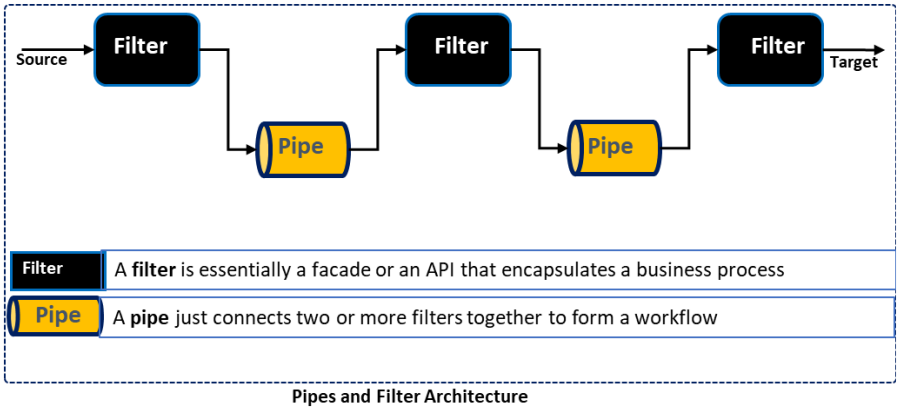

Рисунок 3.8 - Діаграма архітектури «пайпів і фільтрів»

У 2026 році розвиток Інтернету речей (IoT) визначається не тільки різноманітністю пристроїв і відповідною архітектурою дял керування цими пристроями, але і більш суворими вимогами до безпеки та глибокою інтеграцією з хмарними технологіями. Тому розуміння того, які **мови програмування** підходять для кожного рівня системи IoT, є надзвичайно важливим для створення надійних систем, готових до викликів майбутнього.

8. 1. **Java** протягом десятиліть була наріжним каменем розробки масштабного та корпоративного програмного забезпечення, і в 2026 році вона продовжує відігравати важливу роль в екосистемах Інтернету речей (IoT). У багатьох архітектурах IoT Java використовується не на самих найменших пристроях, а в шлюзах, бекенд-платформах та сервісах обробки даних, що розташовані між пристроями та хмарою. Однією з головних переваг Java є її зрілість. Екосистема пропонує надійні фреймворки для створення масштабованих сервісів, обробки обміну повідомленнями та інтеграції з хмарними платформами. Це робить Java поширеним вибором для бекендів IoT, які мають обробляти великі обсяги даних пристроїв, керувати парками пристроїв та надавати API іншим системам. Її сильна типізація та широкий набір

інструментів також допомагають командам підтримувати складні системи з часом, що є особливо важливим у довготривалих проектах IoT, де стабільність та зручність обслуговування мають більше значення, ніж короткострокова швидкість [19].

2. **JavaScript** став однією з найбільш практичних і універсальних мов програмування для розробки в сфері Інтернету речей (IoT), значною мірою завдяки Node.js. У сучасних системах IoT JavaScript зазвичай використовується для створення шлюзів пристроїв, API та хмарних сервісів. Однією з найбільших переваг JavaScript є його керована подіями, неблокуюча модель, яка природно вписується в робочі навантаження IoT, де тисячі пристроїв можуть одночасно надсилати невеликі повідомлення. Це робить його ідеальним для обробки потоків даних у реальному часі, з'єднань WebSocket та легких API. У поєднанні з величезною екосистемою npm команди можуть швидко інтегрувати протоколи, хмарні сервіси та аналітичні інструменти, не винаходячи велосипед. [19].

3.**Python** є однією з найпопулярніших мов програмування для розробки IoT на сьогодні, особливо коли йдеться про обробку даних, автоматизацію та швидке прототипування. Він широко використовується як на периферійних пристроях (таких як Raspberry Pi або подібні плати), так і в хмарних або серверних сервісах, що аналізують та обробляють дані IoT. Найбільша перевага Python є його простота та багатство екосистеми. Існують бібліотеки для роботи з датчиками, комунікації за допомогою поширених протоколів IoT, аналізу даних і навіть застосування машинного навчання до даних пристроїв. Це робить Python чудовим вибором для проектів перевірки концепції, платформ IoT на основі даних та систем, де швидкість експериментування та ітерації є критично важливою [19].

4. **C++** залишається однією з найважливіших мов програмування, що входить до групи Мов опису апарату(DHL - hardware description language) та розробки пристроїв Інтернету речей (IoT), особливо в середовищах, де вирішальне значення мають продуктивність, управління пам'яттю та доступ до апаратного забезпечення. Ця мова широко використовується у вбудованому

програмному забезпеченні, вбудованих системах, контролерах реального часу та компонентах пристроїв IoT, для яких важлива продуктивність.Сильна сторона C++ полягає в його ефективності та можливості низькорівневого управління. Розробники можуть точно налаштовувати використання пам’яті, оптимізувати шляхи виконання та безпосередньо взаємодіяти з периферійними пристроями. Це робить C++ поширеним вибором для промислових контролерів, автомобільних систем та спеціальних апаратних платформ [19].

Відеопотік є одним із найбільш ресурсоємних, але водночас критично важливих елементів сучасних IoT-систем. У контексті робототехніки та дистанційного керування відеозображення з камери пристрою є основним каналом зворьотного зв'язку для оператора - без нього керування роботом у реальному часі стає неможливим. Саме тому вибір протоколу та формату передачі відео безпосередньо впливає на якість взаємодії користувача з системою. Відеопотік у цифровому розумінні є послідовністю кадрів, кожен із яких при передачі через мережу розбивається на пакети даних. У форматі MJPEG кожен кадр є окремим JPEG-зображенням із маркерами початку 0xFF 0xD8 та кінця 0xFF 0xD9, що дозволяє приймальній стороні коректно виокремлювати повні кадри з безперервного байтового потоку.

Для **передачі відео в IoT-системах** існує кілька основних підходів:

1. **HTTP/MJPEG** є найпростішим рішенням, тому що сервер безперервно надсилає JPEG-кадри в межах одного з'єднання, що легко реалізується і широко підтримується, однак є неефективним з точки зору використання пропускної здатності, оскільки кожен кадр передається повністю без урахування змін між ними.

2. **WebSocket** забезпечує двоспрямований канал зв'язку з мінімальними накладними витратами, де відеодані передаються як бінарні фрейми з низькою затримкою, що робить його придатним для систем реального часу.

3. FFmpeg є інструментом обробки відеопотоків на рівні локального обчислювального вузла - він здатний приймати вхідний потік із пристрою,

перекодовувати його у більш ефективний формат, наприклад H.264, та передавати далі зі значно меншим навантаженням на мережу.

4. Secure Reliable Transport (SRT) протокол призначений для передачі високоякісного відео з низькою затримкою через мережі з різним рівнем продуктивності. Він використовує передові методи корекції помилок, щоб забезпечити безперебійну передачу відео навіть у складних умовах, таких як високий трафік, що може спричиняти перевантаження та уповільнення передачі даних, або переривчасте з'єднання, коли мережеве з'єднання є нестабільним або часто обривається. SRT є особливо корисним для додатків Інтернету речей (IoT), оскільки він може впоратися з коливаннями продуктивності мережі, що робить його ідеальним для середовищ, в яких стабільне мережеве з'єднання не завжди доступне. Здатність SRT відновлюватися після втрати пакетів, коли пакети даних не досягають місця призначення, та компенсувати нестабільність, що означає коливання часу прибуття пакетів, гарантує безперебійну передачу відеопотоків високої якості. Крім того, SRT включає потужні функції шифрування, забезпечуючи безпечний канал для передачі відео, що є надзвичайно важливим для додатків, які працюють з конфіденційними даними, таких як системи спостереження та дистанційного моніторингу [20].

5. Протокол потокової передачі в реальному часі (RTSP) - це протокол мережевого управління, який широко використовується в комунікаційних системах для керування серверами потокового медіа. Він ідеально підходить для додатків, що працюють у реальному часі, таких як системи відеоспостереження, оскільки дозволяє безпосередньо керувати функціями відтворення, такими як запуск, пауза та зупинка .Завдяки управлінню процесами підключення та налаштування потоку RTSP забезпечує безперебійний обмін даними між пристроями, що робить його кращим вибором для передачі відео в реальному часі з пристроїв Інтернету речей (IoT). RTSP зазвичай поєднується з протоколом передачі в реальному часі (RTP) для фактичної передачі медіаданих, що підвищує надійність та продуктивність системи. RTP відповідає за передачу

аудіо- та відеоданих, тоді як RTSP керує управлінням та налаштуванням, що забезпечує більш стійку та ефективну систему потокового передавання [20].

6. WebRTC - це універсальний протокол, розроблений компанією Google, який забезпечує зв'язок у режимі реального часу безпосередньо між браузерами та пристроями без використання плагінів. WebRTC є оптимальним вибором для відеодзвінків та інтерактивних застосунків, де критично важлива затримка менше 100 мс, однак його реалізація на рівні мікроконтролера є складною через високі обчислювальні вимоги.Однорангова архітектура WebRTC зменшує потребу в проміжних серверах, що дозволяє знизити витрати та підвищити продуктивність мереж Інтернету речей (IoT). WebRTC підтримує передачу аудіо, відео та даних, пропонуючи комплексне рішення для різних потреб комунікації в Інтернеті речей. Однак архітектура «рівний-рівному» WebRTC може обмежувати масштабованість порівняно з серверними рішеннями, такими як RTSP. Зі збільшенням кількості підключених пристроїв управління прямими з'єднаннями може стати складним, і для роботи з брандмауерами можуть знадобитися методи проходження через мережу, такі як STUN, TURN та ICE [20].

Клієнтський рівень IoT-системи потребує окремого підходу до вибору інструментів розробки, оскільки мобільний застосунок є основною точкою взаємодії користувача з усією платформою. Серед сучасних фреймворків для розробки кросплатформених мобільних застосунків найбільшого поширення набули **React Native та Flutter**.

React Native, розроблений компанією Meta, використовує JavaScript як основну мову програмування та забезпечує взаємодію з нативними компонентами операційної системи через міст (bridge). Це означає, що інтерфейс застосунку використовує реальні нативні елементи платформи, що забезпечує природний вигляд та поведінку на кожній операційній системі окремо. React Native є особливо доцільним у проєктах, де команда вже має досвід із JavaScript або React, а також у системах, що потребують тісної інтеграції з нативними API

пристрою: камерою, геолокацією, Bluetooth або WebSocket-комунікацією у реальному часі.

Flutter, розроблений компанією Google, використовує мову Dart та власний рушій рендерингу Skia, який малює інтерфейс безпосередньо на полотні, минаючи нативні компоненти платформи. Це забезпечує піксельно ідентичний вигляд застосунку на всіх платформах та високу продуктивність анімацій. Flutter є оптимальним вибором для застосунків із складним кастомним дизайном та насиченою графікою, однак інтеграція з низькорівневими нативними API може потребувати додаткових зусиль порівняно з React Native.

Коли ми починаємо розробляти проєкт на новій мові чи з використанням нового фреймворку, одне з перших питань, що виникає, - це як організувати проєкт. Якщо покладатися на офіційну документацію, React не визначає правильного способу організації файлів і залишає це на розсуд розробника, однак вони пропонує кілька прикладів популярних структур. Це структура на основі типів файлів та структура на основі функцій.

Є 4 основні способи організації та структурування проектів React та React Native:

Структура за типами файлів: організація файлів за типами, де більшість файлів розміщується у папках з назвами відповідних типів, розташованих у кореневому каталозі проекту. Іншими словами, на кореневому рівні є папка components для спільних компонентів, а в папці pages кожна сторінка міститиме основний вигляд та компоненти, які використовуються лише на цій сторінці. Така організація дозволяє мати дуже просту структуру для проектів, що тільки починаються. Проте головна проблема полягає в тому, що коли проект починає розростатися, це може перетворитися на справжній хаос, особливо на рівні компонентів, оскільки в найпростішому варіанті можна розмістити їх усі в папці components [21].

Модульна структура або організація за функціональністю. Така структура цікава для проектів більшого масштабу. Основна ідея полягає в тому, що кожен модуль, який визначається, містить весь пов'язаний з ним код, і

імпортується лише код із самого модуля. Коли у нас є кілька модулів, яким потрібен один і той самий фрагмент коду, ми можемо записати його у спільну папку та імпортувати в різні модулі. Основне правило, якого слід дотримуватися, - не імпортувати код між модулями. Великим недоліком цієї структури є те, що визначення того, що таке модуль, може бути складним, а це може мати великий вплив на успіх нашої організації [21].

Структура на основі Atomic Design. Це методологія створення систем дизайну, розроблена Бредом Фростом та Дейвом Олсеном. Вона не є унікальною для React, але дуже добре поєднується з цією бібліотекою завдяки підходу до створення інтерфейсів на основі компонентів. Atomic Design - це не повний метод структурування проєкту, а спосіб організації наших компонентів. Тому це скоріше шаблон, який ми можемо застосувати в рамках нашої існуючої організації. Основна ідея полягає в тому, щоб розділити компоненти на п’ять типів елементів: атоми, молекули, організми, шаблони, сторінки [21].

Структура на основі гексагональної архітектури, також відома як шаблон Ports and Adapters, - це підхід до розробки програмного забезпечення, який сприяє високій модульності та гнучкості. Цю архітектуру задумав Алістер Кокберн, який зауважив, що додатки взаємодіють із зовнішніми системами подібним чином - чи то бази даних, API чи інші сервіси. Ключовий принцип шестикутної архітектури полягає в ізоляції основної бізнес-логіки від зовнішніх залежностей. Щоб вловити принцип гексагональної архітектури, важливо зрозуміти традиційну трирівневу модель, яка складається з: презентаційного шару - користувацька частина додатку(інтерфейс або API), логічний шар - основна бізнес-логіка(обробка даних і рівень даних (зберіганням і отриманням даних через бази даних). Гексагональна архітектура (або архітектура «портів і адаптерів») побудована навколо трьох ключових концепцій - ядро, порти, адаптери. Ядро застосунку містить бізнес-логіку системи, яка є повністю незалежною від зовнішніх систем - баз даних. Ядро не знає нічого про те, звідки надходять дані і куди вони передаються далі. Порти - це інтерфейси, які визначають що ядро очікує на вході і що воно повертає на виході. Адаптери - це

конкретні реалізації портів, які забезпечують взаємодію з зовнішніми системами. Саме адаптери «знають» як спілкуватись із базою даних Архітектура візуалізується у вигляді шестикутника з чітким розділенням на дві сторони. Ведуча сторона (входи) є компонентами, які ініціюють поведінку системи, наприклад API. Ведена сторона (виходи) є компонентами з якими система взаємодіє сама, наприклад бази даних [22].,

3.2. Моделювання бізнес-процесів та системних компонентів

Use Case діаграма системи керування роботом (рисунок 3.9) охоплює чотири актори та три функціональні пакети. Акторами системи є Користувач як основний ініціатор взаємодії, ESP32-S3 як мозок системи, Node.js як основний сервер для передачі команд між клієнтом та мікроконтролером та Python для детекції QR-кодів. Користувач відкриває застосунок, встановлює WebSocket-з'єднання з сервером та підключається до камери ESP32-S3. Основне завдання - почати гру. Цей етап обов'язково включає встановлення з'єднання та підключення до камери. Надсилання команди керування є обов'язковою складовою гри та відбувається через Node.js до мікроконтролера. Отримання -XP та нарахування +XP є розширеннями базового сценарію, що активуються відповідно при зіткненні з перешкодою або успішному маневрі. Сканування QR-коду є необов'язковим розширенням, яке включає нарахування балів та виконується Python-скриптом комп’ютерний зір та бібліотеку OpenCV.Пакет автономного режиму описує сценарій самостійної навігації робота. Зчитування даних сенсорів є обов'язковим кроком, що виконується ESP32-S3, тоді як уникнення перешкоди є розширенням, що активується лише при виявленні об'єкта на шляху руху.

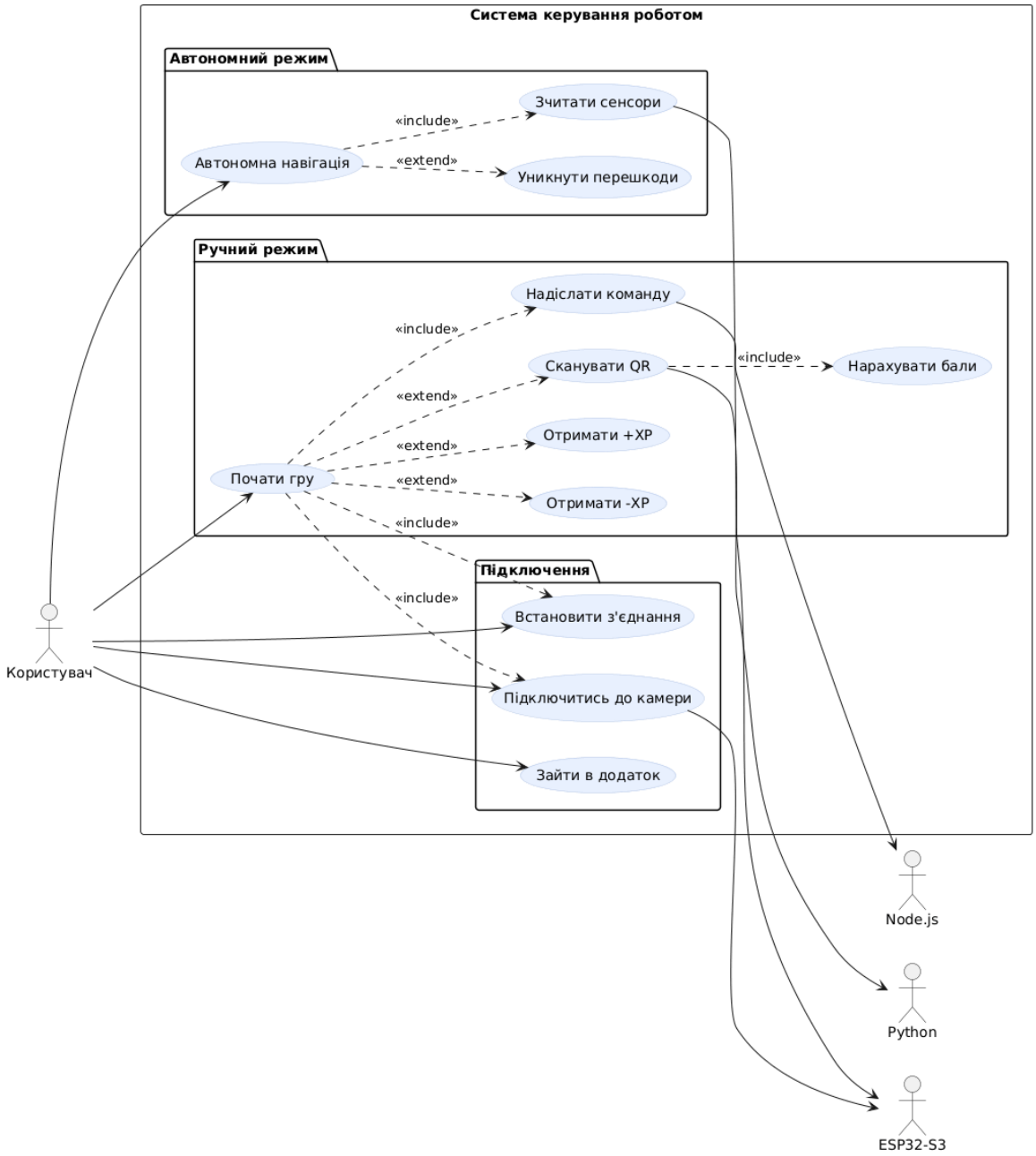

Рисунок 3.9 - Use Case діаграма системи автономного робота між користувачем і акторами

Sequence діаграма сценарію встановлення з'єднання (рисунок 3.10) описує послідовність взаємодії між чотирма учасниками: користувачем, React Native застосунком, Node.js сервером та мікроконтролером ESP32-S3.

Сценарій розпочинається з моменту коли користувач відкриває застосунок, після чого React Native відображає екран StartWIFIConnection. Ініціюється через натискання кнопки Connect і застосунок встановлює повторне

з'єднання з Node.js сервером. Node.js у свою чергу встановлює з'єднання з ESP32-S3, який підтверджує успішне підключення через WiFi.

На другому етапі користувач ініціює підключення до камери. React Native надсилає запит на встановлення WebSocket-з'єднання з камерою до Node.js, який передає його до ESP32-S3. Мікроконтролер підтверджує встановлення WebSocket-з'єднання, сервер повідомляє застосунок про готовність камери, після чого React Native відображає екран PreviewCamera уже з відео. Сценарій завершується моментом коли користувач натискає кнопку початку гри.

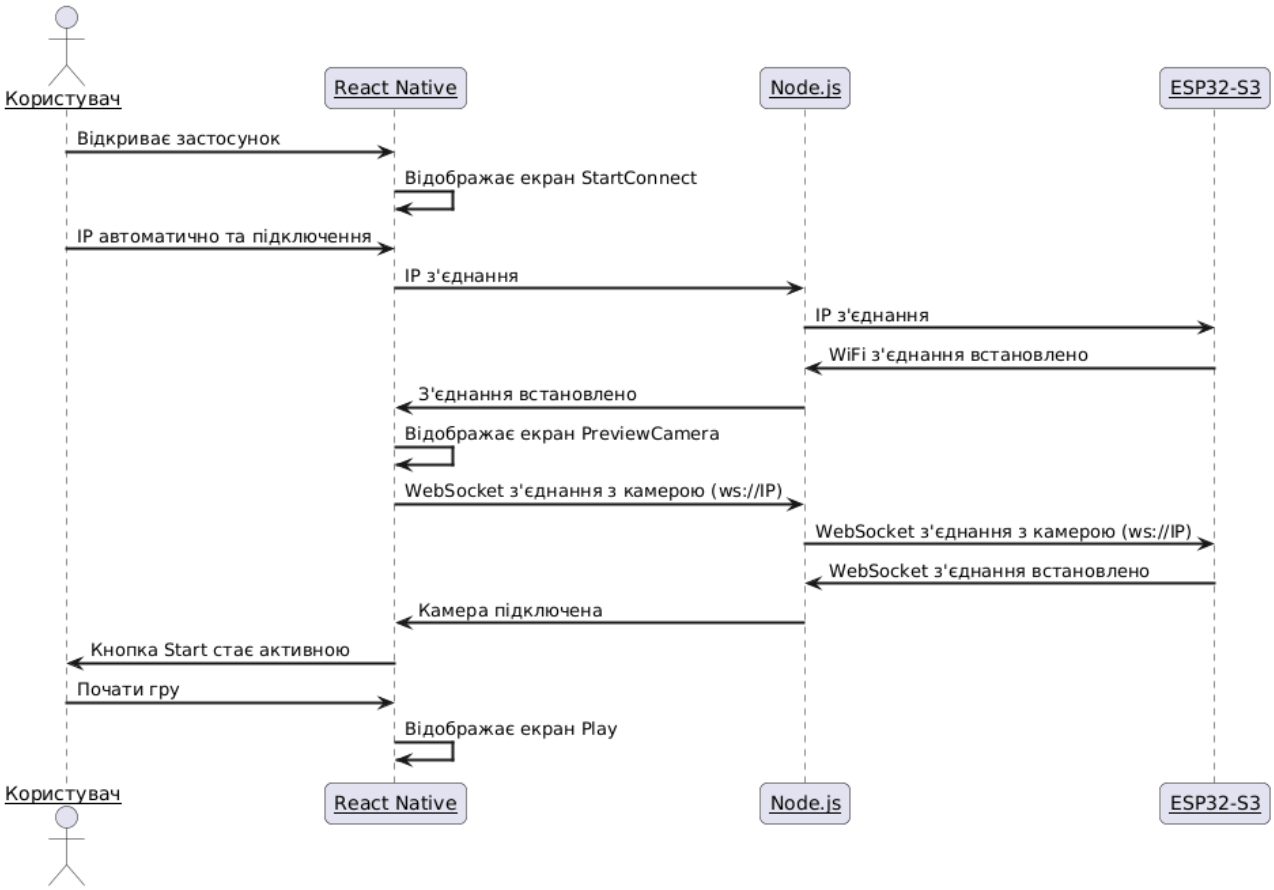

Рисунок 3.10 - Sequence діаграма сценарію для встановлення з'єднання

Sequence діаграма сценарію керування розпочинається з моменту коли користувач натискає кнопку напрямку в ігровому інтерфейсі. React Native застосунок негайно надсилає відповідне WebSocket повідомлення(“GO_LEFT, “GO_RIGHT”, “GO_FORWARD”, “GO_BACKWARDS”, “STOP”) з командою до

Node.js сервера, який відправляє її до мікроконтролера (рисунок 3.11). Отримавши команду, ESP32-S3 обробляє її та передає PWM сигнал драйверу моторів для виконання руху. Паралельно з цим ультразвукові датчики HC-SR04 безперервно сканують простір перед роботом, і у разі зіткнення з перешкодою чип генерує подію OBSTACLE, яка передається через Node.js до React Native застосунку, де користувач отримує штраф -XP.

Незалежно від керування роботом, ESP32-S3 безперервно транслює MJPEG відеопотік до Node.js сервера, який передає його до Python модуля. Python аналізує кожен кадр у пошуку QR-кодів, і при успішній детекції передає результат разом із значенням коду назад до Node.js, який надсилає QR-код до React Native застосунку. Застосунок нараховує відповідну кількість балів та відображає результат користувачу.

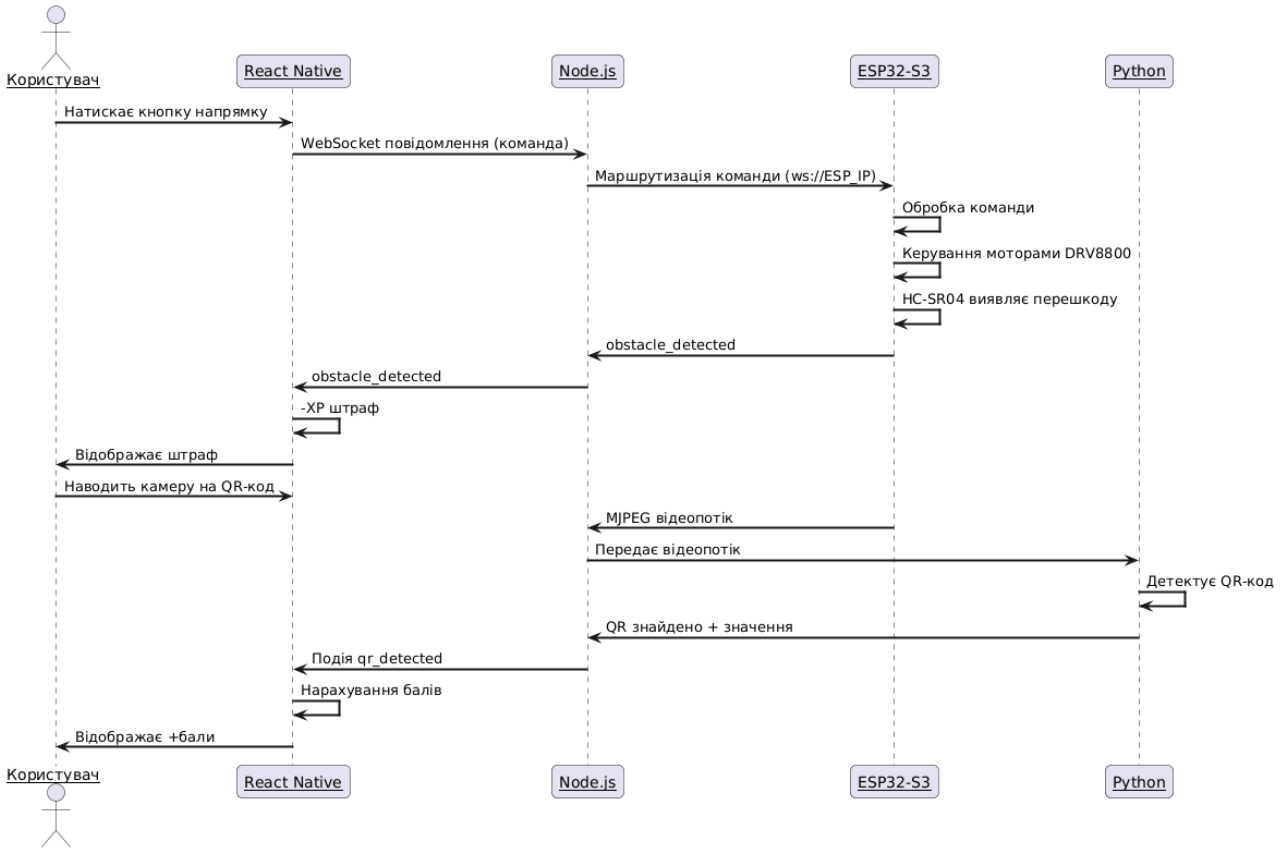

Рисунок 3.11 - Sequence діаграма сценарію для гри

Sequence діаграма сценарію розпочинається з моменту як тільки

користувач перемикає режим роботи на автономний через інтерфейс застосунку. React Native надсилає WebSocket повідомлення до Node.js сервера, який передає відповідну команду до ESP32-S3. Мікроконтролер припиняє обробку ручних команд керування та підтверджує зміну режиму, після чого Node.js повідомляє застосунок, а користувач бачить оновлений статус на екрані.Після активації автономного режиму система переходить у безперервний цикл навігації. На кожній ітерації ESP32-S3 зчитує показники трьох ультразвукових датчиків HC-SR04, отримуючи відстань до перешкод зліва, по центру та справа.

Activity діаграма логіки (рисунок 3.12) автономної навігації описує **алгоритм прийняття рішень мікроконтролером** на основі даних трьох ультразвукових датчиків.

На початку кожної ітерації циклу послідовно зчитує показники відстані з лівого, центрального та правого датчиків із невеликою затримкою між зчитуваннями для уникнення інтерференцій з ультразвуковими хвилями. На основі отриманих значень обчислюються три умови небезпеки: центральна перешкода вважається критичною при відстані менше 30 см, тоді як бічні перешкоди - при відстані менше 20 см. Якщо жодна з умов небезпеки не виконується, робот продовжує рух вперед і цикл повторюється. У разі виявлення перешкоди в будь-якому з напрямків система зупиняє мотори та переходить до вибору напрямку об'їзду. Рішення приймається шляхом порівняння показників лівого та правого датчиків - якщо перешкода зліва є ближчою ніж справа, робот повертає праворуч, в іншому випадку - ліворуч. Після виконання маневру цикл відновлюється з нового зчитування сенсорів.

Рисунок 3.12 - Activity діаграма для алгоритму логіки автономної навігації

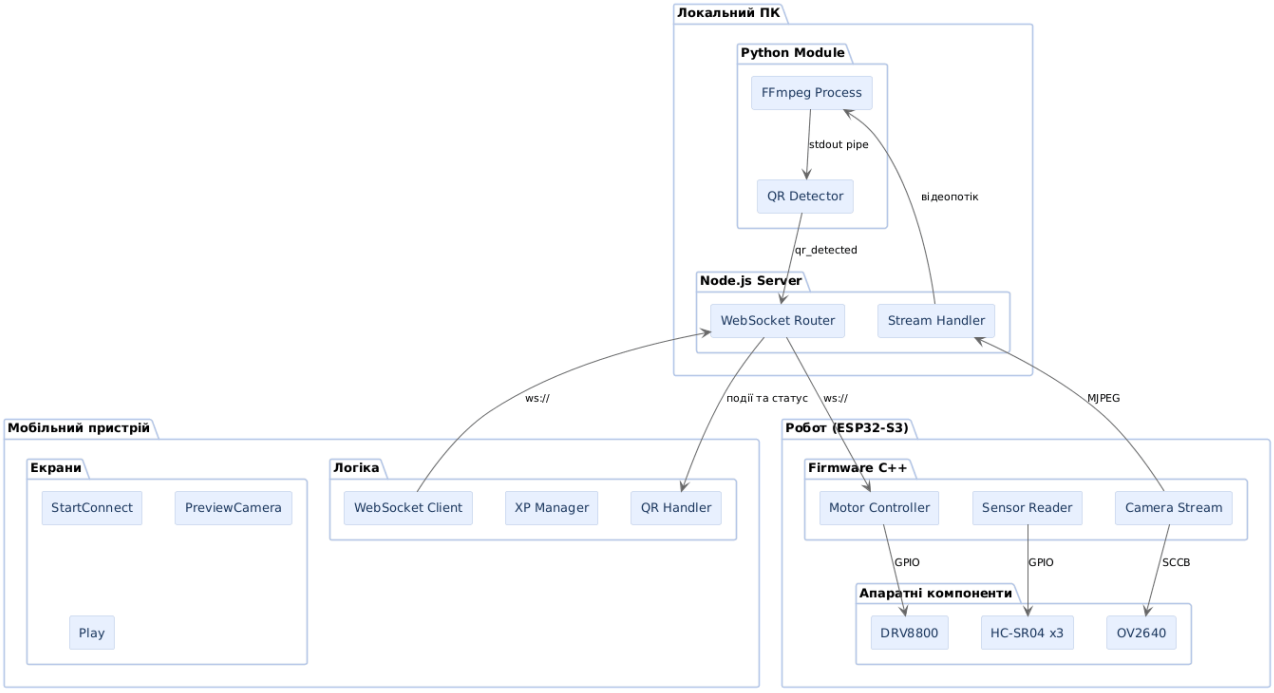

Рисунок 3.13 - Component діаграма всіх елементів системи

Component діаграма (рисунок 3.13) системи відображає архітектуру трьох незалежних, але пов’язаних між собою компонентів.

Перший компонент - мобільний пристрій, що містить дві підгрупи компонентів. Група екранів включає StartConnect для встановлення з'єднання, PreviewCamera для попереднього перегляду камери та Play як основний ігровий екран. Група логіки містить WebSocket Client, який забезпечує комунікацію з сервером у реальному часі, XP Manager, який відповідає за нарахування та списання ігрових балів, та QR Handler, який обробляє QR-коди, що отримує від сервера.

Другий компонент - мій локальний комп’ютер, на якому створені і запущені 2 незалежних модулі. Node.js Server містить WebSocket Router, який відправляє команди з мобільного застосунку до ESP32-S3, та Stream Handler, який приймає MJPEG відео-дані від мікроконтролера та передає його до Python. Python файл містить FFmpeg Process, який декодує вхідне відео та передає окремі кадри через stdout pipe до QR Detector, який аналізує кожен кадр у пошуку QR-кодів та повертає результат детекції до WebSocket Router.

Третій компонент - робот на базі ESP32-S3, який складається з програмної та апаратної частин. Прошивка, написана мовою C++, містить Motor Controller, який отримує команди керування та передає сигнали до драйвера моторів через GPIO, Sensor Reader, який безперервно зчитує дані з ультразвукових датчиків HC-SR04 та Camera Stream, що запускає камеру.

3.3. Вибір та обгрунтування технологій та інструментів розробки

Вибір стеку є одним із ключових завдань, що безпосередньо впливає на продуктивність, та зручність розробки системи. Кожен компонент обирався на основі аналізу альтернатив з урахуванням специфічних вимог проєкту (таблиця 3.1.

Мікроконтролер ESP32-S3-WROOM був обраний як основний апаратний компонент систем. На відміну від Raspberry Pi, який є повноцінним

одноплатним комп'ютером із надлишковими для даної задачі ресурсами та значно вищим енергоспоживанням, ESP32-S3 забезпечує достатню обчислювальну потужність для виконання прошивки, керування моторами та трансляції відеопотоку в компактному форм-факторі з низьким енергоспоживанням. Порівняно з Arduino, ESP32-S3 має вбудований WiFi, значно більший обсяг пам'яті та підтримку апаратного прискорення, що є критично необхідним для бездротової комунікації у реальному часі. Варто зазначити, що початково система розроблялась на базі ESP32-CAM - компактнішого модуля з вбудованою камерою. Однак у процесі розробки виявилось, що ESP32-CAM має критичні обмеження: лише 9 доступних GPIO-пінів, відсутність достатньої кількості ліній для одночасного підключення трьох ультразвукових датчиків, драйвера моторів та інших периферійних пристроїв. З урахуванням вимог до розширюваності системи та необхідності підключення повного набору сенсорів, було прийнято рішення перейти на ESP32-S3-WROOM, який надає суттєво більше можливостей при збереженні компактного форм-фактору.

Таблиця 3.1

Порівняльна характеристика апаратних платформ для розробки IoT-систем.

Характеристика	**ESP32-CAM**	**ESP32-S3-WROOM**	**Raspberry Pi 4**	**Arduino Uno**
Процесор	Xtensa LX6, 240 МГц	Xtensa LX7, 240 МГц	ARM Cortex-A72, 1.8 ГГц	ATmega328P, 16 МГц
RAM	520 КБ + 4 МБ PSRAM	512 КБ + 8 МБ PSRAM	4/8 ГБ	2 КБ
Flash	4 МБ	8 МБ	MicroSD	32 КБ
GPIO піни	9	45	40	14
WiFi	802.11 b/g/n	802.11 b/g/n	802.11 b/g/n/ac	Відсутній
Bluetooth	BT 4.2	BLE 5.0	BT 5.0	Відсутній
Камера	OV2640 (вбудована)	Підтримка через SCCB	Через USB/CSI	Відсутня

Енергоспоживання	~180 мА	~240 мА	~3000 мА	~50 мА
AI прискорення	Відсутнє	Векторні інструкції	Відсутнє	Відсутнє
Вартість	Низька	Середня	Висока	Низька

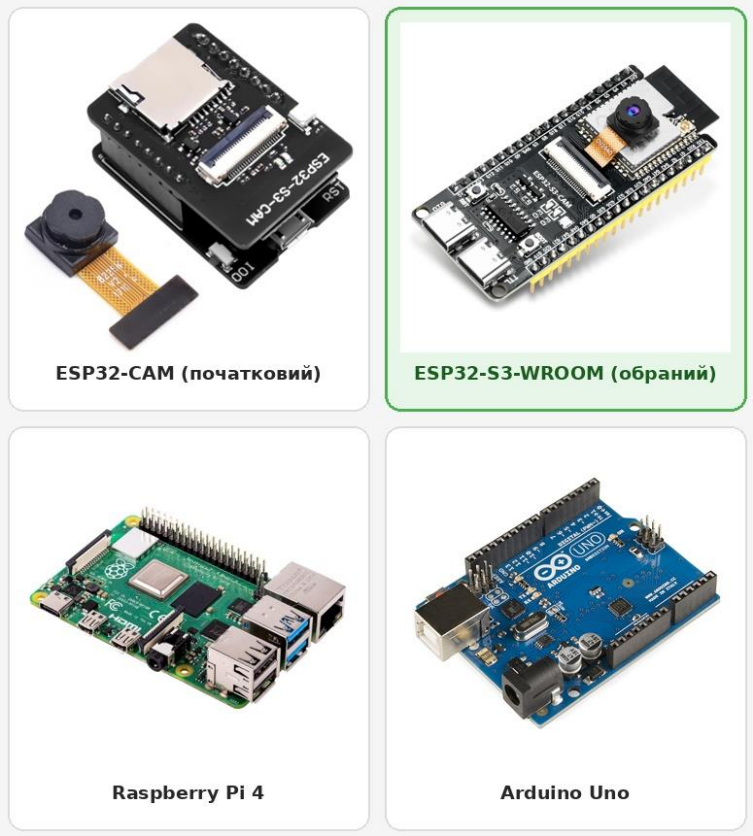

Рисунок 3.13 - Чотири плати

Мова програмування C++ була обрана для розробки прошивки мікроконтролера як стандарт для embedded систем. Вона надає повний контроль над розподілом пам'яті та апаратними ресурсами, що є критично важливим в умовах обмеженого середовища мікроконтролера, де відсутня операційна система та автоматичне управління пам'яттю.

React Native був обраний як фреймворк для розробки клієнтського застосунку замість Flutter з кількох причин. По-перше, React Native використовує JavaScript та архітектурні підходи React, що забезпечує природну інтеграцію з Node.js сервером у межах єдиної JavaScript-екосистеми. По-друге,

кросплатформена природа фреймворку дозволяє розгортати застосунок як на Android, так і на iOS без дублювання коду.

Node.js був обраний як серверна платформа завдяки його подієво-орієнтованій архітектурі та неблокуючій моделі вводу-виводу, що робить його оптимальним для систем із великою кількістю одночасних з'єднань у реальному часі. Вбудована підтримка WebSocket через бібліотеку ws, природна інтеграція з JavaScript-екосистемою React Native, а також ефективна обробка потоків даних від ESP32-S3 та Python модуля роблять Node.js ідеальним координатором у розподіленій архітектурі системи.

WebSocket був обраний як основний протокол комунікації між усіма компонентами системи замість традиційного HTTP. HTTP-модель запит-відповідь є неприйнятною для керування роботом у реальному часі через накладні витрати на встановлення з'єднання при кожному запиті. **WebRTC**, незважаючи на свою здатність забезпечувати мінімальну затримку через peer-to-peer з'єднання, є надмірно складним для реалізації на рівні мікроконтролера через високі обчислювальні вимоги. **RTSP**, орієнтований переважно на передачу медіа у системах відеоспостереження та не є призначеним для двоспрямованої передачі команд керування та потребує окремого транспортного протоколу **RTP** для фактичної доставки даних, що ускладнює архітектуру системи. WebSocket натомість забезпечує постійне двоспрямоване з'єднання після єдиного рукостискання, що мінімізує затримку передачі команд керування та повністю відповідає визначеним нефункціональним вимогам системи.

FFMPEG був обраний як інструмент обробки відео на рівні локальному рівні, завдяки його здатності приймати MJPEG та передавати окремі кадри через stdout pipe до Node.Js і Python для подальшої обробки. Така архітектура дозволяє повністю розділити логіку трансляції відео та обробки зображень, забезпечуючи модульність та незалежність компонентів системи.

Python був обраний як мова реалізації модуля детекції QR-кодів завдяки розвиненій екосистемі бібліотек комп'ютерного зору, зокрема OpenCV та pyzbar,

які забезпечують ефективне декодування QR-кодів із відеопотоку в реальному часі з мінімальною кількістю коду.

3.4. Висновки по розділу

У третьому розділі було проведено комплексний аналіз архітектурних підходів, технологій та інструментів розробки, що склали основу проєктованої системи керування автономним мобільним роботом.

В рамках огляду архітектурних підходів було розглянуто подієво-орієнтовану архітектуру, хмарну архітектуру, патерн публікація-підписка, мікросервісну архітектуру та інші. Аналіз показав, що жоден із підходів не є універсальним — кожен має свою область застосування та компроміси, а розроблена система поєднує елементи кількох архітектурних патернів відповідно до вимог кожного рівня.

Огляд технологій передачі відеопотоку охопив протоколи HTTP/MJPEG, WebSocket, WebRTC, RTSP, SRT, що дозволило обґрунтувати вибір оптимального підходу для трансляції відео в умовах обмежених ресурсів edge-пристрою. Порівняльний аналіз мобільних фреймворків React Native та Flutter визначив переваги першого в контексті інтеграції з JavaScript-екосистемою та нативними API пристрою.

Моделювання системи засобами UML дозволило формалізувати функціональні вимоги та поведінку системи. Use Case діаграма визначила акторів та сценарії взаємодії з відповідними зв'язками include та extend. Три Sequence діаграми детально описали послідовність взаємодії компонентів у сценаріях встановлення з'єднання, ручного керування з гейміфікацією та автономної навігації. Activity діаграма формалізувала алгоритм прийняття рішень мікроконтролера на основі даних ультразвукових датчиків.

Сукупність прийнятих архітектурних та технологічних рішень формує цілісну, систему, готову до реалізації та подальшого розширення.

РОЗДІЛ 4. РЕАЛІЗАЦІЯ ПРОЄКТУ

4.1. Опис розробки програмного коду

Програмний код системи організований у вигляді 4 незалежних модулів, кожен з яких відповідає за окремий рівень архітектури: клієнтський застосунок на React Native, серверна частина на Node.js, апаратна частина на C++ та комп'ютерний зір на Python.

Клієнтський застосунок (рисунок 4.1) було ініціалізовано за допомогою офіційного інструменту Expo CLI командою `npx create-expo-app --template`, яка автоматично завантажує та розгортає базову структуру проєкту з попередньо налаштованим середовищем розробки. У процесі ініціалізації було виявлено попередження про невідповідність версій. Node.js v20.15.0 є нижчою за мінімально рекомендовану для react-native@0.81.5, однак це не вплинуло на коректність збірки та виконання проєкту.

Після ініціалізації було встановлено такі ключові пакети:

`ws` — для WebSocket-комунікації між клієнтом та Node.js сервером у реальному часі.

`react-native-svg` — для рендерингу векторної графіки елементів інтерфейсу.

`expo-router` — для файлової маршрутизації між екранами застосунку.

`buffer` — для декодування бінарних даних відеопотоку з `ArrayBuffer` у формат base64.

Точкою входу є index.tsx, який перенаправляє користувача до екрану StartConnect. Попередній перегляд відео винесено в окремий контекст StartStreaming.tsx, який реалізує патерн Provider та надає дочірнім компонентам доступ до поточного кадру, стану з'єднання та методів керування WebSocket-сесією.

Рисунок 4.1 - Клієнтський додаток

При отриманні бінарного повідомлення від сервера контекст обробляє його у кілька етапів. Відеокадр надходить у форматі ArrayBuffer - сирого бінарного буфера, який браузер та React Native не можуть відобразити напряму як зображення. Для відображення його необхідно перетворити у формат, який розуміє компонент Image. Спочатку ArrayBuffer передається до Buffer.from() з бібліотеки buffer, яка інтерпретує байти як двійкові дані. Далі викликається .toString('base64'), який кодує ці байти у рядок із символів ASCII - це стандартний спосіб представити бінарні дані у текстовому форматі. Отриманий рядок вставляється у data URI формату який є посиланням на зображення: він містить і тип даних (image/jpeg), і самі дані одразу в рядку, без необхідності окремого HTTP-запиту. Саме такий URI передається як source компоненту Image, який відображає кожен новий кадр як статичне зображення, створюючи ілюзію відеопотоку через швидку послідовну заміну кадрів.

```
            ws.onmessage = (event) => {
                setLoading(false);
                const base64data = Buffer.from(event.data).toString('base64');
                setFrameUrl(`${base64data}`);
            };
```

Фрагмент коду 4.1- Перетворення бінарних даних в режим Base64

Основним екраном ігрового процесу є Play.tsx. Компонент підтримує два режими роботи: MANUAL_MODE та SMART_MODE, перемикання між якими здійснюється через відправку відповідної команди на сервер. Керуючі команди GO_LEFT, GO_RIGHT, GO_FORWARD, GO_BACKWARD та STOP надсилаються через WebSocket-з'єднання на порт 8880 при натисканні відповідних кнопок.

```
      <Pressable {...leftButton.pressableProps} style={styles.buttonLeft}
onPress={() => sendCommand("GO_LEFT")} >
                                <Image source={leftButton.imageSource}
style={styles.buttonImage} contentFit="contain" />
                            </Pressable>
                            <Pressable {...stopButton.pressableProps}
style={styles.stopButton} onPress={() => sendCommand("STOP")}>
                                <Image source={stopButton.imageSource}
style={styles.buttonImage} contentFit="contain" />
                            </Pressable>
                            <Pressable {...rightButton.pressableProps}
style={styles.buttonRight} onPress={() => sendCommand("GO_RIGHT")}>
                                <Image source={rightButton.imageSource}
style={styles.buttonImage} contentFit="contain" />
                            </Pressable>

                            <Pressable {...forwardButton.pressableProps}
style={styles.buttonForward} onPress={() => sendCommand("GO_FORWARD")}>
                                <Image source={forwardButton.imageSource}
style={styles.buttonImage} contentFit="contain" />
                            </Pressable>
                            <Pressable {...backwardButton.pressableProps}
style={styles.buttonBackward} onPress={() => sendCommand("GO_BACKWARD")}>
                                <Image source={backwardButton.imageSource}
style={styles.buttonImage} contentFit="contain" />
                            </Pressable>
```

Фрагмент коду 4.2 - компонент Play.tsx

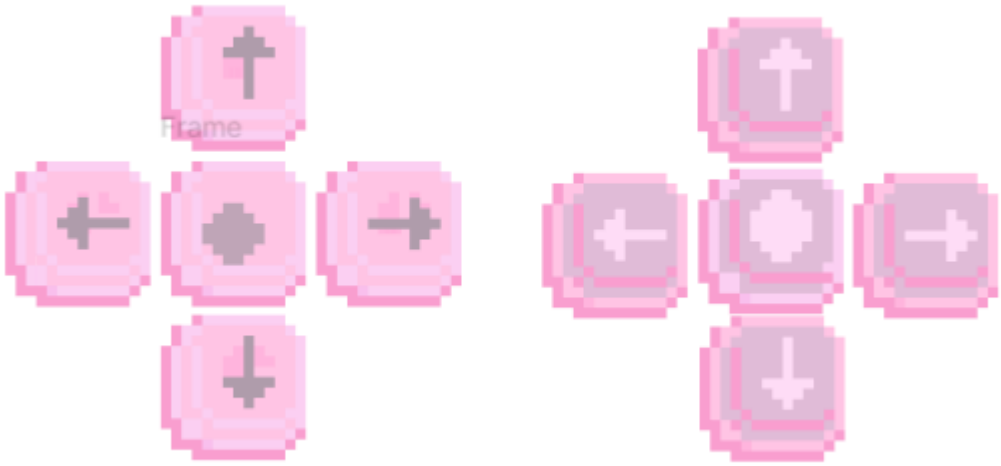

Рисунок 4.2 - Дизайн кнопок у статичному та активному режимах у Figma

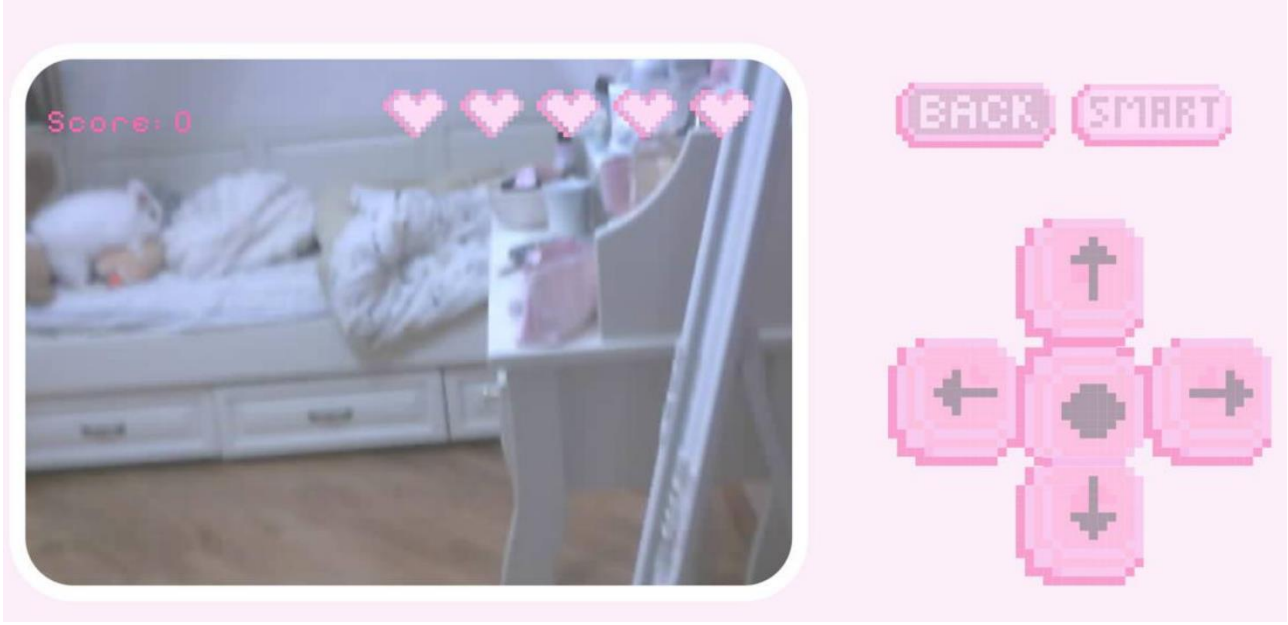

Рисунок 4.3 - Вигляд кнопок у компоненті Play.tsx

Для реалізації інтерактивних кнопок (рисунок 4.2, 4.3) з візуальним станом натиснення розроблено кастомний хук usePressableImage, який через useMemo керує джерелом зображення залежно від стану isPressed.

```
const pressableProps = useMemo(       () => disablePressOut         ? {
                    onPressIn: () => setIsPressed((prev) => !prev),
                }
                : {
                    onPressIn: () => setIsPressed(true),
                    onPressOut: () => setIsPressed(false),
                },
            [disablePressOut]
        );
```

Фрагмент коду 4.4 - кастомний хук usePressableImage

Рисунок 4.3 - Демонстрація роботи хука

Система нарахування балів реалізована через Set<string> - кожне унікальне повідомлення (рисунок 4.3) від Python модуля додається до множини, що унеможливлює подвійне нарахування балів за один QR-код.

Серверна частина складається з двох незалежних WebSocket-серверів. Сервер команд commandServer.ts слухає на порту 8880 та шляху /esp-32. При встановленні з'єднання сервер зчитує параметр role з URL запиту та призначає клієнту відповідну роль - client, esp-32 або python. Логіка маршрутизації в commands.js реалізована через перевірку ролі: команди від клієнта пересилаються лише підключеним ESP32, а повідомлення від Python модуля - лише клієнтським застосункам. Сервер відеопотоку videostreamServer.ts слухає на порту 8888. При підключенні клієнта запускається процес FFmpeg через spawn, який отримує MJPEG потік з ESP32 за адресою

http://192.168.3.148:81/stream та передає оброблені кадри через stdout.

```
    const esp32Url = "http://192.168.3.148:81/stream";
```

```
        python.on('spawn', () => console.log('Python запустився'));
        python.on('error', (err) => console.error('Python помилка:',
err));
        python.stderr.on('data', (data) => console.error('Python
stderr:', data.toString()));
        python.stdout.on('data', (data) => console.log('Python
stdout:', data.toString()));
        const ffmpeg = spawn("ffmpeg", [
            "-f", "mjpeg",
            "-avoid_negative_ts", "make_zero",
       "-fflags", "nobuffer+discardcorrupt",
       "-flags", "low_delay",
       "-i", esp32Url,
            "-r", "15",
     "-f", "mjpeg",
            "-q:v", "2",                              "-"
        ]);
```

Фрагмент коду 4.3 - Інструмент FFMPEG зчитує відеоотік з камери

Сервер накопичує вихідні дані у буфері та виокремлює повні JPEG кадри за маркерами 0xFF 0xD8 та 0xFF 0xD9, після чого надсилає їх підключеним клієнтам у бінарному форматі.

```
    ffmpegProcess.stdout.on("data", (chunk) => {
            buffer = Buffer.concat([buffer, chunk]);

            while (true) {
                const start = buffer.indexOf(Buffer.from([0xff,
0xd8]));
                const end = buffer.indexOf(Buffer.from([0xff, 0xd9]),
start);
                if (start !== -1 && end !== -1) {
                    const frame = buffer.slice(start, end + 2);
                    buffer = buffer.slice(end + 2);
```

Фрагмент коду 4.4 - виокремлення маркерів 0xFF 0xD8 та 0xFF 0xD9

Процесу FFmpeg, аналогічно виокремлює кадри за JPEG маркерами та декодує кожен кадр за допомогою OpenCV. Для детекції QR-кодів використовується

бібліотека pyzbar, яка повертає текстовий вміст та координати знайденого коду.

```
        for bc in codes:
                        x, y, w, h = bc.rect

                        cv2.rectangle(image, (x, y), (x + w, y + h), (255, 0, 0), 3)

                        barcode_text = bc.data.decode('utf-8')
                        barcode_type = bc.type
```

Фрагмент коду 4.5 - Детекції QR-кодів

Рисунок 4.5 - Детекції QR-коду у Python

Рисунок 4.6 - Детекції QR-коду у додатку

При успішній детекції модуль відправляє текст QR-коду через WebSocket-з'єднання до сервера команд із роллю python, який відправляє його до клієнтського застосунку. WebSocket-з'єднання Python модуля виконується в окремому потоці через threading.Thread для уникнення блокування основного циклу обробки відеопотоку.

Прошивка мікроконтролера ESP32-S3 написана мовою C++ у середовищі Arduino IDE. Ініціалізація апаратних компонентів виконується у функції setup(), де послідовно налаштовуються піни моторів (MOTOR_IN1–MOTOR_IN4, STBY), піни ультразвукових датчиків (TRIG_PIN, ECHO_PIN_L, ECHO_PIN_C, ECHO_PIN_R), камера OV2640. Функція setup() викликається під час першого запуску коду. Використовується якраз таки для ініціалізації змінних, режимів виведення виводів, початку використання бібліотек тощо. Функція setup() виконуватиметься лише один раз, після кожного ввімкнення або перезавантаження плати Arduino [24].

```
    Serial.begin(115200);
    Serial.setDebugOutput(true);

    pinMode(MOTOR_IN1, OUTPUT);
    pinMode(MOTOR_IN2, OUTPUT);
    pinMode(MOTOR_IN3, OUTPUT);
    pinMode(MOTOR_IN4, OUTPUT);
    pinMode(STBY, OUTPUT);
    pinMode(TRIG_PIN, OUTPUT);
    pinMode(ECHO_PIN_L, INPUT);
    pinMode(ECHO_PIN_C, INPUT);
    pinMode(ECHO_PIN_R, INPUT);
```

Фрагмент коду 4.6 - Ініціалізація апаратних компонентів

Конфігурація камери реалізована з адаптивною логікою: при наявності PSRAM система встановлює роздільну здатність SVGA, якість JPEG на рівні 10 та подвійну буферизацію з режимом CAMERA_GRAB_LATEST для плавного відеопотоку. У разі відсутності PSRAM система автоматично знижує роздільну здатність до VGA, збільшує стиснення та обмежується одним буфером у DRAM,

що запобігає переповненню пам'яті.

Керування моторами через драйвер DRV8800 реалізоване через ШІМ-сигнали analogWrite на чотирьох пінах. Кожна функція руху (goForward, goBackwards, goLeft, goRight, stop) безпосередньо встановлює рівні сигналів на відповідних пінах та керує піном STBY для активації або деактивації драйвера.

```
void stop() {
  analogWrite(MOTOR_IN1, 0);
  analogWrite(MOTOR_IN2, 0);
  analogWrite(MOTOR_IN3, 0);
  analogWrite(MOTOR_IN4, 0);
  digitalWrite(STBY, LOW);
  Serial.println("stop");
}

void goForward() {
  analogWrite(MOTOR_IN1, 0);
  analogWrite(MOTOR_IN2, MAX_SPEED);
  analogWrite(MOTOR_IN3, 0);
  analogWrite(MOTOR_IN4, MAX_SPEED);
  digitalWrite(STBY, HIGH);
  Serial.println("go");
}
void goBackwards() {
  analogWrite(MOTOR_IN1, MAX_SPEED);
  analogWrite(MOTOR_IN2, 0);
  analogWrite(MOTOR_IN3, MAX_SPEED);
  analogWrite(MOTOR_IN4, 0);
  digitalWrite(STBY, HIGH);
  Serial.println("backwards");
}
void goLeft(int speed = MAX_SPEED) {
  analogWrite(MOTOR_IN1, MAX_SPEED);
```

```
  analogWrite(MOTOR_IN2, 0);
  analogWrite(MOTOR_IN3, 0);
  analogWrite(MOTOR_IN4, MAX_SPEED);
  digitalWrite(STBY, HIGH);
  Serial.println("left");
}
void goRight(int speed = MAX_SPEED) {
  analogWrite(MOTOR_IN1, 0);
  analogWrite(MOTOR_IN2, MAX_SPEED);
  analogWrite(MOTOR_IN3, MAX_SPEED);
  analogWrite(MOTOR_IN4, 0);
  digitalWrite(STBY, HIGH);
  Serial.println("right");
}
```

Фрагмент коду 4.7 - Ініціалізація апаратних компонентів

Після створення функції setup(), яка ініціалізує та встановлює початкові значення, функція loop() виконує саме те, що випливає з її назви, і послідовно виконує цикл, дозволяючи програмі реагувати [23]. Цикл loop() є мінімалістичним: викликає webSocket.loop() для обробки мережевих подій та умовно запускає функцію AUTONOMOUS() залежно від поточного режиму роботи.

```
// ====================
// Loop
// ====================
void loop() {
  webSocket.loop();

  if (autonomousMode) {
    AUTONOMOUS();
  }
}
```

Фрагмент коду 4.8 - Ініціалізація апаратних компонентів

4.2. Використані алгоритми та структури даних

Ключовим алгоритмом клієнтської частини є алгоритм унікалізації QR-балів на основі структури даних `Set`. На відміну від масиву, множина автоматично відхиляє дублікати, що гарантує нарахування балів лише за унікальні QR-коди незалежно від кількості їх зчитувань камерою.

Центральним алгоритмом серверної та Python частини є алгоритм виокремлення JPEG кадрів з безперервного байтового потоку. Алгоритм підтримує змінний буфер, до якого послідовно дописуються вхідні чанки даних. На кожній ітерації виконується пошук маркера початку кадру `0xFF 0xD8`, після чого пошук маркера кінця `0xFF 0xD9`. За наявності обох маркерів виокремлюється повний кадр, буфер зсувається на позицію після кінцевого маркера, і цикл повторюється до вичерпання доступних повних кадрів.

Система команд реалізована через патерн Command Map std::map<std::string, CommandHandler>, де ключем є рядкова назва команди, а значенням - вказівник на відповідну функцію-обробник. При отриманні WebSocket-повідомлення функція callEachCommand виконує пошук у мапі та викликає відповідний обробник, що забезпечує розширюваність системи без модифікації логіки маршрутизації.

```
commandMap["GO_FORWARD"] = goForward;
commandMap["STOP"] = stop;
commandMap["GO_LEFT"] = []() {
  goLeft();
};
commandMap["GO_RIGHT"] = []() {
  goRight();
};
commandMap["GO_BACKWARDS"] = goBackwards;
commandMap["RESTART"] = restart;
commandMap["SMART_MODE"] = toggleSmartMode;
commandMap["MANUAL_MODE"] = toggleManualMode;
```

Фрагмент коду 4.9 - Command Map std::map<std::string, CommandHandler>

У даній системі використовуються три датчики HC-SR04, розміщені в лівому, центральному та правому напрямках, що дозволяє роботу отримувати тривимірну картину навколишнього простору.

Ультразвуковий датчик відстані HC-SR04 (рисунок 4.7) є одним із найпоширеніших сенсорів у робототехніці та IoT. Принцип його роботи базується на вимірюванні часу поширення ультразвукової хвилі: датчик випромінює короткий імпульс частотою 40 кГц через передавач (Trig), який відбивається від перешкоди та повертається до приймача (Echo). Час між відправкою та отриманням імпульсу пропорційний відстані до об'єкта, яка обчислюється за формулою, де

t — час у мікросекундах

0.034 — швидкість звуку в см/мкс, а ділення на 2 враховує двосторонній шлях сигналу.

$$d = t \times 0.034 / 2$$

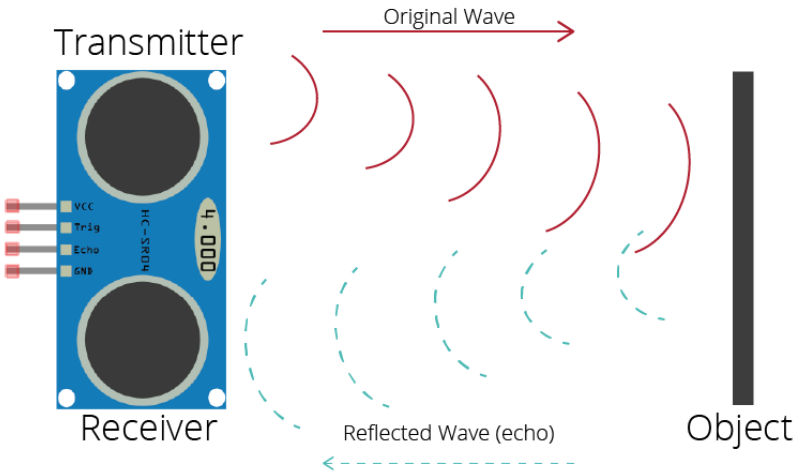

Рисунок 4.7 - Принцип роботи HC-SR04 [25]

У процесі розробки алгоритму автономної навігації, було суттєве переопрацювання. Початкова реалізація використовувала складну багатокрокову стратегію сканування: при виявленні перешкоди робот зупинявся, відїжджав назад, повертав ліворуч для сканування, повертався до центру, потім повертав праворуч для повторного сканування, знову повертався до центру і

лише після цього порівнював суми відстаней обох напрямків для прийняття рішення. Така реалізація містила численні асинхронні виклики функцій delay() із загальною тривалістю понад 4 секунди на один цикл прийняття рішення, що повністю блокувало обробку WebSocket-подій та унеможливлювало своєчасну реакцію на команди перемикання режиму. Крім того, алгоритм демонстрував нестабільну поведінку при значеннях датчиків рівних 0 або 250, які інтерпретувались як відсутність відлуння.

Код першої версії алгоритму:

```
void AUTONOMOUS() {
          Serial.println("SMARTMODE");
          long fwd = readDistance(ECHO_PIN_C);
          sensorCenter = fwd;
          delay(30);
          long sideLeft = readDistance(ECHO_PIN_L);
          sensorLeft = sideLeft;
          delay(30);
          long sideRight = readDistance(ECHO_PIN_R);
          sensorRight = sideRight;
          delay(30);

          Serial.println("F:" + String(fwd) + " L:" + String(sideLeft) + " R:" + String(sideRight));
          if (fwd == 0 || fwd == 250) {
            stop();
            return;
          }
          if (fwd > 40 && sideLeft > 15 && sideRight > 15) {
            goForward();
            return;
          }
          stop();
          delay(500);
          goBackwards();
          delay(500);
          stop();
          goLeft(200);
          delay(600);
          stop();
          delay(500);
          long fwdScan1;
            fwdScan1 = readDistance(ECHO_PIN_C);
            sensorCenter = fwdScan1;
            delay(30);
          long sideLeftScan1;
            sideLeftScan1 = readDistance(ECHO_PIN_L);
            sensorLeft = sideLeftScan1;
            delay(30);
          long sideRightScan1;
            sideRightScan1 = readDistance(ECHO_PIN_R);
            sensorRight = sideRightScan1;
```

```
 delay(30)
  Serial.println("Left side scan L:" + String(sideLeftScan1));
  Serial.println("Left side scan C:" + String(fwdScan1));
  Serial.println("Right side scan R:" + String(sideRightScan1));
  goRight(200);
  delay(600);
  stop();
  delay(1000);
  goRight(200);
  delay(600);
  stop();
  delay(200);
  long fwdScan2;
  for (int i = 0; i < 2; i++) {
    fwdScan2 = readDistance(ECHO_PIN_C);
    sensorCenter = fwdScan2;
    delay(30);
  }
  long sideLeftScan2;
  for (int i = 0; i < 2; i++) {
    sideLeftScan2 = readDistance(ECHO_PIN_L);
    sensorLeft = sideLeftScan2;
    delay(30);
  }
  long sideRightScan2;
  for (int i = 0; i < 2; i++) {
    sideRightScan2 = readDistance(ECHO_PIN_R);
    sensorRight = sideRightScan2;
    delay(30);
  }
  Serial.println("Left scan:" + String(sideLeftScan2));
  Serial.println("Center scan:" + String(fwdScan2));
  Serial.println("Right scan:" + String(sideRightScan2));
  goLeft(200);
  delay(600);
  stop();
  long distanceSum1 = (fwdScan1 + sideLeftScan1 + sideRightScan1);
  long distanceSum2 = (fwdScan2 + sideLeftScan2 + sideRightScan2);
  if (fwd >= fwdScan1 && fwd >= fwdScan2 && fwd > 40) {
    goForward();
  } else if (distanceSum1 >= distanceSum2 && distanceSum1 > 60) {
    goLeft(200);
    delay(600);
    goForward();
  } else if (distanceSum2 > distanceSum1 && distanceSum2 > 60) {
    goRight(200);
    delay(600);
    goForward();
  } else {
    goBackwards();
    delay(700);
    stop();
  }
}
```

Фрагмент коду 4.9 - Перша версія алгоритму оминання перешкод

Переопрацьований алгоритм базується на безпосередньому порівнянні показників трьох датчиків без зворотнього руху та сканування. Рішення приймається миттєво на основі булевих умов небезпеки з порогами 30 см для центрального датчика та 20 см для бічних, а вибір напрямку повороту визначається простим порівнянням leftDanger > rightDanger. Це дозволило скоротити час циклу навігації до 150 мс та зберегти реактивність WebSocket-з'єднання.

Код другої версії алгоритму:

```
void AUTONOMOUS() {
  sensorLeft = readDistance(ECHO_PIN_L);
  delay(15);
  sensorCenter = readDistance(ECHO_PIN_C);
  delay(15);
  sensorRight = readDistance(ECHO_PIN_R);
  delay(15);

  Serial.print("F:");
  Serial.print(sensorCenter);
  Serial.print(" L:");
  Serial.print(sensorLeft);
  Serial.print(" R:");
  Serial.print(sensorRight);
  bool centerDanger = (sensorCenter > 0 && sensorCenter < 30);
  bool leftDanger = (sensorLeft > 0 && sensorLeft < 20);
  bool rightDanger = (sensorRight > 0 && sensorRight < 20);
  if (centerDanger || leftDanger || rightDanger) {
    Serial.println(" -> stop");
    stop();
    if (leftDanger > rightDanger) {
      goLeft(250);
      return;
    } else {
      goRight(250);
      return;
    }
  } else {
    Serial.println(" -> go");
    goForward();
  }
}
```

Фрагмент коду 4.10 - Друга версія алгоритму оминання перешкод

QR-код є двовимірною матрицею, що представляє бінарну сітку чорних та білих кольорів. Структура QR-коду включає три маркери позиціонування у

кутах, що утворюють характерний L-подібний патерн, часові доріжки для визначення масштабу та орієнтації, блок інформації про формат, а також область даних із кодами корекції помилок.

Процес детекції починається з бінаризації кадру: перетворення зображення у двійкову матрицю за пороговою фільтрацією. Кожен піксель зі значенням яскравості нижче порогу отримує значення 0 (чорний модуль), вище — 1 (білий модуль). Часові доріжки QR-коду — послідовності модулів, що чергуються — дозволяють алгоритму адаптивно визначити оптимальний поріг бінаризації залежно від умов освітлення.

Ключовою математичною основою QR-кодів є корекція помилок **Ріда-Соломона** над скінченним полем GF(256) — полем із 256 елементами, де арифметика виконується за модулем примітивного полінома

$$p(x) = x^8 + x^4 + x^3 + x^2 + 1$$

Множення в GF(256) реалізується як множення поліномів із подальшим діленням на p(x) та взяттям залишку, що гарантує збереження результату в діапазоні 0–255.

Генераторний поліном для t кодових слів корекції помилок визначається як:

$$g(x) = (x - \alpha^0)(x - \alpha^1)(x - \alpha^2)\cdots(x - \alpha^{(t-1)})$$

де α є примітивним елементом поля GF(256). При кодуванні даних обчислюється поліном повідомлення m(x), і кодове слово формується як:

$$c(x) = m(x)\cdot x^t + (m(x)\cdot x^t \bmod g(x))$$

де залишок від ділення утворює кодові слова корекції помилок.

4.3. Модульне тестування та налагодження

Для перевірки коректності ключових алгоритмів серверної та клієнтської частин системи було розроблено набір Unit-тестів із використанням фреймворку Jest. Тестування охоплює шість незалежних сценаріїв, кожен з яких перевіряє окремий функціональний модуль системи.

Перший тест перевіряє що всі визначені команди керування: GO_FORWARD, GO_BACKWARDS, GO_LEFT, GO_RIGHT, STOP, SMART_MODE, MANUAL_MODE. Це присутні у словнику команди, що гарантує коректність маршрутизації на рівні сервера.

```
describe('Command Routing', () => {
        test('valid command should be found in commandMap', ()
=> {
            const validCommands = [
                'GO_FORWARD', 'GO_BACKWARDS', 'GO_LEFT',
                'GO_RIGHT', 'STOP', 'SMART_MODE',
'MANUAL_MODE'
            ];
            validCommands.forEach(cmd => {
                expect(validCommands.includes(cmd)).toBe(true)
;
            });
        });
```

Фрагмент коду 4.11 - Перевірка визначених команд керування

Другий тест перевіряє що невалідна команда, яка не входить до визначеного набору, коректно відхиляється системою без виклику обробника.

```
    test('invalid command should not be in commandMap', () => {
            const invalidCommand = 'FLY_AWAY';
            const validCommands = ['GO_FORWARD', 'STOP',
'GO_LEFT', 'GO_RIGHT'];
            expect(validCommands.includes(invalidCommand)).toB
e(false);
        });
```

Фрагмент коду 4.11 - Перевірка невизначених команд керування

Третій тест перевіряє функцію парсингу ролі клієнта з URL-параметра WebSocket-з'єднання. Очікується що ролі esp-32, client та python коректно витягуються з рядка запиту, а за відсутності параметра повертається значення unknown.

```
    test('role parsing from URL query', () => {
```

```
            const parseRole = (url: string): string => {
                const params = new
URLSearchParams(url.split('?')[1]);
                return params.get('role') || 'unknown';
            };
            expect(parseRole('/esp-32?role=esp-
32')).toBe('esp-32');
            expect(parseRole('/esp-
32?role=client')).toBe('client');
            expect(parseRole('/esp-
32?role=python')).toBe('python');
            expect(parseRole('/esp-32')).toBe('unknown');
        });
```

Фрагмент коду 4.12 - Перевірка функції парсингу ролі клієнта з URL-параметра

Четвертий тест перевіряє порогові значення алгоритму автономної навігації — функції визначення небезпечної відстані для центрального датчика (менше 30 см) та бічних датчиків (менше 20 см), а також коректну обробку нульових значень як відсутності відлуння.

```
    test('obstacle detection thresholds', () => {
```

```
            const isCenterDanger = (dist: number) => dist > 0
&& dist < 30;
            const isSideDanger = (dist: number) => dist > 0 &&
dist < 20;

            expect(isCenterDanger(25)).toBe(true);
            expect(isCenterDanger(35)).toBe(false);
            expect(isCenterDanger(0)).toBe(false);
            expect(isSideDanger(15)).toBe(true);
            expect(isSideDanger(25)).toBe(false);
        });
```

Фрагмент коду 4.13 - Перевірка функції парсингу ролі клієнта з URL-параметра

П'ятий тест перевіряє механізм зчитування QR-балів через структуру даних Set: повторне додавання одного й того самого QR-коду не має збільшувати розмір множини.

```
    test('QR score deduplication via Set', () => {
            const scores = new Set<string>();
            scores.add('QR_001');
            scores.add('QR_001');
            scores.add('QR_002');
            expect(scores.size).toBe(2);
        });
```

Фрагмент коду 4.14 - Перевірка механізму зчитування QR-балів

Шостий тест перевіряє коректність визначення маркерів початку 0xFF 0xD8 та кінця 0xFF 0xD9 JPEG-кадру у бінарному буфері, що є основою алгоритму виокремлення кадрів із відеопотоку.

```
        test('JPEG frame markers detection', () => {
            const startsWithJpeg = (buf: number[]) =>
                buf[0] === 0xFF && buf[1] === 0xD8;
            const endsWithJpeg = (buf: number[]) =>
                buf[buf.length - 2] === 0xFF && buf[buf.length
- 1] === 0xD9;
            const fakeFrame = [0xFF, 0xD8, 0x00, 0x01, 0xFF,
0xD9];
            expect(startsWithJpeg(fakeFrame)).toBe(true);
            expect(endsWithJpeg(fakeFrame)).toBe(true);
        });
```

Фрагмент коду 4.14 - Перевірка коректності маркерів JPEG-кадру у бінарному буфері

Усі шість тестів були успішно виконані командою npx jest, що підтверджує коректність реалізації ключових алгоритмів системи.

4.4. Висновки по розділу

У четвертому розділі реалізовано повний програмний продукт та систему автономного мобільного робота, що охоплює збірку апартних компонентів, прошивку мікроконтролера, серверну частину, підбірку та впровадження декількох мережевих протоколів для обробки відеопотоку та двохстороннього з’єднання, клієнтський застосунок. Описано ключові алгоритми та структури даних, обґрунтовано архітектурні рішення, а також задокументовано еволюцію алгоритму автономної навігації в процесі розробки. Проведене модульне тестування підтвердило коректність реалізації критичних компонентів системи.

РОЗДІЛ 5. ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ

5.1.Стратегії та методи тестування

У даному розділі описано методологію тестування чотириколісного автономного робота на основі ESP32-S3 WROOM. Метою тестування є перевірка коректності роботи двох режимів: ручного (Manual) та автономного (Autonomous), а також надійності апаратної та програмної частин системи**.**

Для перевірки функціональності робота були застосовані такі методи тестування:

1. **Функціональне тестування:** перевірка відповідності кожної функції системи заявленим вимогам.

2. **Тестування ручного режиму:** перевірка керування роботом через зовнішній інтерфейс (додаток / пульт).

3. **Тестування автономного режиму:** перевірка самостійного руху та обходу перешкод.

4. **Тестування з'єднання:** перевірка стабільності бездротового зв'язку (Wi-Fi/Bluetooth).

В таблиці 5.1 наведено тест-кейси для перевірки ручного режиму керування роботом.

Таблиця 5.1

Тест-кейси для перевірки ручного режиму керування роботом.

№ тесту	Опис тесту	Очікуваний результат	Фактичний результат
1	Рух вперед	Робот рухається вперед без відхилення	Робот рухається з дуже мінімальним відхиленням через специфічну механіку коліс
2	Рух назад	Робот рухається назад без відхилення	Робот рухається назад з поворотои навколо своєї осі.

№ тесту	Опис тесту	Очікуваний результат	Фактичний результат
3	Поворот ліворуч	Робот повертає ліворуч на команду	Робот повертає ліворуч на команду та може зробити повний оберт навколо своєї осі проти годинникової стрілки
4	Поворот праворуч	Робот повертає праворуч на команду	Робот повертає праворуч на команду та може зробити повний оберт навколо своєї осі проти годинникової стрілки
5	Зупинка	Робот зупиняється при відпусканні команди	Робот одразу зупиняється при відпусканні команди
6	Перемикання режиму	Успішне перемикання з Manual на Auto	Робот перемикання з Manual на Smart режим
7	Затримка відповіді	Затримка між командою і реакцією < 200 мс	Так

В таблиці 5.2 наведено тест-кейси для перевірки автономного режиму роботи.

Таблиця 5.2

Мест-кейси для перевірки автономного режиму

№ тесту	Опис тесту	Очікуваний результат	Фактичний результат
1	Виявлення перешкоди спереду	Робот зупиняється і змінює напрямок	Робот змінює напрямок, щоб оминути перешкоду перед собою, але не завжди справляється з цим завданням через фізичні обмеження звукових хвиль, а також вразливість до умов навколишнього середовища.
2	Обхід перешкоди зліва	Робот успішно об'їжджає перешкоду зліва	Робот змінює напрямок, щоб оминути перешкоду зліва, але не завжди справляється з цим завданням через фізичні обмеження звукових хвиль, а також вразливість до умов навколишнього середовища.
3	Обхід перешкоди справа	Робот успішно об'їжджає перешкоду справа	Робот змінює напрямок, щоб оминути перешкоду перед собою, але не завжди справляється з цим завданням через фізичні

№ тесту	Опис тесту	Очікуваний результат	Фактичний результат
			обмеження звукових хвиль, а також вразливість до умов навколишнього середовища.
4	Тупик (перешкоди з трьох сторін)	Робот розвертається і знаходить вихід	
5	Рух по відкритому простору	Робот рухається без зупинок по вільному маршруту	Робот рухається без зупинок по вільному маршруту

Таблиця 5.3

№ тесту	Опис тесту	Очікуваний результат	Фактичний результат
1	Підключення на відстані 5 м	Стабільний зв'язок, команди виконуються	Стабільний зв'язок, команди виконуються
2	Підключення на відстані 10 м	Стабільний зв'язок без затримок	Стабільний зв'язок, команди виконуються
3	Втрата з'єднання під час руху	Робот зупиняється, не виконує хаотичних дій	Робот зупиняється, не виконує хаотичних дій
4	Повторне підключення	З'єднання відновлюється автоматично або вручну	З'єднання краще відновлюється вручну ніж автоматично

5.2. Результати тестування системи

За результатами функціонального тестування ручний режим керування продемонстрував стабільну роботу в усіх перевірених сценаріях. Затримка між відправкою команди та реакцією робота не перевищує 200 мс, що відповідає визначеним нефункціональним вимогам. Рух вперед виконується з мінімальним відхиленням, зумовленим механічними особливостями конструкції коліс, що є прийнятним для даного типу шасі. Поворот та рух назад виконуються через обертання навколо власної осі, що є характеристикою диференційного приводу. Перемикання між ручним та автономним режимами відбувається коректно в

межах визначеного часового порогу.

Автономний режим продемонстрував задовільну поведінку у відкритому просторі: робот впевнено рухається без зупинок за відсутності перешкод. При виявленні перешкод спереду та з боків система коректно реагує та змінює напрямок руху. Разом з тим виявлено обмеження, пов'язані з фізичною природою ультразвукових датчиків HC-SR04, тому що їхня чутливість до кута відбиття, поглинаючих поверхонь та акустичних завад навколишнього середовища призводить до нестабільних показників у певних умовах. Відсутність датчика заднього огляду унеможливлює автономний вихід із тупикової ситуації, що є відомим обмеженням поточної конфігурації апаратного забезпечення, тому у таких випадках рекомендується перемикання у ручний режим керування.

Тестування з'єднання підтвердило стабільну роботу WebSocket-комунікації на відстані до 10 метрів. При втраті з'єднання робот коректно зупиняється без хаотичних дій. Автоматичне відновлення з'єднання працює менш надійно порівняно з ручним перепідключенням, що є напрямком для подальшого вдосконалення серверної логіки.

5.3 Аналіз ефективності впровадження

Розроблена система демонструє практичну придатність як дослідницька IoT-платформа. Її першочергова цінність полягає не стільки в досконалості механіки чи навігації, скільки у дослідженні способів комунікації між пристроями, а саме протоколів передачі відео, організації двостороннього та багатостороннього зв’язку між трьома вузлами системи одночасно. Саме це є ключовим науковим внеском даної роботи та у **розвиток Інтернет роботизованих речей.**

Водночас розробка виявила широкий спектр перспектив для подальшого вдосконалення системи. По-перше, заміна ультразвукових датчиків на LiDAR забезпечить суттєво вищу точність та стабільність навігації незалежно від умов навколишнього середовища. Впровадження серйознішого мікроконтролера

відкриє можливості для мультизадачного комп’ютерного зору безпосередньо на пристрої, а використання якіснішої камери підвищить точність детекції об’єктів у реальному часі. Повна заміна механічної бази, а саме шасі, коліс та моторів дозволить усунути відхилення при русі та підвищити загальну надійність системи.

На програмному рівні перспективним напрямком є інтеграція повноцінного алгоритму злиття сенсорних даних IMU та LiDAR, впровадження алгоритмів машинного навчання для розпізнавання об’єктів та вдосконалення механізму автоматичного відновлення з’єднання.

5.4. Висновки по розділу

У п'ятому розділі проведено комплексне тестування системи керування автономним мобільним роботом. Функціональне тестування підтвердило коректність роботи ручного режиму та стабільність WebSocket-комунікації в межах визначених нефункціональних вимог. Автономний режим продемонстрував задовільну поведінку в стандартних умовах із виявленими обмеженнями, зумовленими фізичною природою ультразвукових датчиків та поточною конфігурацією апаратного забезпечення. Отримані результати підтверджують працездатність системи та визначають конкретні напрямки для її подальшого вдосконалення.

ВИСНОВКИ

У цій бакалаврській роботі представлено проектування та реалізацію інтегрованої платформи Інтернету речей (IoT) та робототехніки, створеної повністю з нуля: від паяння апаратних компонентів і налаштування мікроконтролера до розробки розподіленого бекенду, комп'ютерного зору та мобільного додатка з ігровими елементами. Цінність цієї роботи полягає не в окремому компоненті, а в продуманій побудові цілісної багатошарової системи, де кожне архітектурне рішення базувалося на реальних інженерних обмеженнях, а не на теоретичних припущеннях.

Дослідження зробило значний внесок у розуміння протоколів зв'язку в розподілених вбудованих системах. Завдяки дослідженню та порівнянню численних підходів до передачі даних у реальному часі, включаючи потокове передавання HTTP, WebSocket, WebRTC, RTSP, SRT. У цій роботі зроблено практичні, обґрунтовані висновки щодо того, які протоколи є придатними для двостороннього зв'язку між трьома вузлами: мікроконтролером, локальним обчислювальним сервером та мобільним клієнтом. Це не тривіальна проблема, і розроблені тут рішення відображають справжнє дослідження меж того, що можуть витримати полегшені протоколи в умовах обмежень реального часу.

Багаторівнева архітектура системи, що охоплює прошивку на C++, рівень Node.js, модуль обробки зображень на Python та інтерфейс React Native демонструє, що робототехнічні платформи не обов’язково потребують спеціалізованого промислового обладнання для реалізації складної розподіленої поведінки. Було продемонстровано, що ESP32-S3-WROOM, який зазвичай вважається модулем Інтернету речей для споживачів, може виконувати функції периферійного обчислювального вузла у поєднанні з добре розробленою програмною архітектурою. Цей висновок ставить під сумнів традиційне припущення, що для повноцінної автономної робототехніки необхідне високопродуктивне апаратне забезпечення.

Окрім технічних досягнень, цей проєкт демонструє можливість того, що один розробник може одночасно проектувати, збирати та інтегрувати повнофункціональну вбудовану систему, що охоплює апаратний, бекенд- , фронтенд- та мережевий рівні.

Випробування в реальних умовах підтвердили експлуатаційну готовність системи, а виявлені обмеження, такі як чутливість ультразвукового датчика, відсутність функції виявлення перешкод позаду та чіткої орієнтації в просторі дають чіткий і об’єктивний план подальшого розвитку, що включає інтеграцію LiDAR, об’єднання даних датчиків з інерційною вимірювальною системою MPU-6050 та сприйняття на основі машинного навчання та повного вдосконалення механіки.

Ця робота підтверджує, що Інтернет роботизованих речей (IoT) є родючим ґрунтом для досліджень у галузі програмної інженерії,роботехніки, автономних систем, а найповніші системи часто створюються в умовах обмежень, коли кожне рішення має бути обґрунтованим, а архітектура має витримувати все навантаження реальної фізичної системи, що працює у непередбачуваному світі.

СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ

1. Інтернет речей: перспективи розвитку та можливості використання. URL: https://duikt.edu.ua/ua/news-1-0-11324-internet-rechey-perspektivi-rozvitku-ta-mozhlivosti-vikoristannya (дата звернення: 02.03.2026).
2. Smart Robotics 101. Symmetry Electronics. URL: https://www.symmetryelectronics.com/blog/smart-robotics-101/ (дата звернення: 05.03.2026).
3. IoT and Robotics: a synergy. ResearchGate. URL: https://www.researchgate.net/publication/344989839_IoT_and_Robotics_a_synergy (дата звернення: 09.03.2026).
4. Інтернет речей (IoT) у цифровій трансформації бізнес-процесів IT компаній. ResearchGate. URL: https://www.researchgate.net/publication/380669216_Internet_recej_IoT_u_cifrovij_transformacii_biznes-procesiv_IT_kompanij (дата звернення: 12.03.2026).
5. IoT Analytics Market. Fortune Business Insights. URL: https://www.fortunebusinessinsights.com/iot-analytics-market-114206 (дата звернення: 15.03.2026).
6. IoT Predictions for 2026. GTIA. URL: https://gtia.org/blog/11-iot-predictions-for-2026 (дата звернення: 18.03.2026).
7. Розробка платформи віддаленого управління інфраструктурою Інтернет речей. ResearchGate. URL: https://www.researchgate.net/publication/354184727_Rozrobka_platformi_viddalenogo_upravlinna_infrastrukturou_Internet_recej (дата звернення: 22.03.2026).
8. What Describes the Relationship Between Edge Computing and Cloud Computing? Intellisoft. URL: https://intellisoft.io/what-describes-the-relationship-between-edge-computing-and-cloud-computing/ (дата звернення: 25.03.2026).
9. Edge Computing vs Cloud: Latency Impact. Firecell. URL: https://firecell.io/edge-computing-vs-cloud-latency-impact/ (дата звернення: 28.03.2026).

10. Architecture of Internet of Things (IoT). GeeksforGeeks. URL: https://www.geeksforgeeks.org/computer-networks/architecture-of-internet-of-things-iot/ (дата звернення: 02.04.2026).

11. What is FFmpeg? Ant Media. URL: https://antmedia.io/what-is-ffmpeg/ (дата звернення: 06.04.2026).

12. Expo Documentation. URL: https://expo.dev/ (дата звернення: 10.04.2026).

13. IoT Trends to Watch: 2026. LinkedIn. URL: https://www.linkedin.com/pulse/iot-trends-watch-2026-denys-hukov-d0fdf/ (дата звернення: 14.04.2026).

14. Design and Implementation of IoT System Based on ESP32-S3. MDPI Sensors. URL: https://www.mdpi.com/1424-8220/25/6/1656 (дата звернення: 18.04.2026).

15. IoT Trends 2026. Metadesk Global. URL: https://metadeskglobal.com/iot-trends-2026/ (дата звернення: 22.04.2026).

16. Requirements Analysis. Wikipedia. URL: https://en.wikipedia.org/wiki/Requirements_analysis (дата звернення: 26.04.2026).

17. Valente M. T. Engenharia de Software Moderna. 2020. URL: https://softengbook.org/chapter3 (дата звернення: 01.05.2026).

18. Navigating IoT System Architectures: Styles and Recommendations. URL: https://blog.ptidej.net/navigating-iot-system-architectures-styles-and-recommendations/ (дата звернення: 05.05.2026).

19. Top IoT Languages Every Developer Should Learn in 2026. Digiscorp. URL: https://digiscorp.com/top-iot-languages-every-developer-should-learn-2026/ (дата звернення: 10.05.2026).

20. Video Streaming Protocols for IoT. Nabto. URL: https://www.nabto.com/video-streaming-protocols-for-iot/ (дата звернення: 15.05.2026).

21. Folder Structures to Organize React Project. Reboot Studio. URL: https://reboot.studio/blog/folder-structures-to-organize-react-project (дата звернення: 20.05.2026).

22. Hexagonal Architecture: Guide to Decoupled Software Design. LinkedIn. URL: https://ua.linkedin.com/pulse/hexagonal-architecture-guide-decoupled-software-design-karthik-rana-sobbc (дата звернення: 25.05.2026).

Arduino loop() Function Reference. Arduino Documentation. URL: https://docs.arduino.cc/language-reference/en/structure/sketch/loop/ (дата звернення: 28.05.2026).

23. Arduino setup() Function Reference. Arduino Documentation. URL: https://docs.arduino.cc/language-reference/en/structure/sketch/setup/ (дата звернення: 28.05.2026).

24. Complete Guide for Ultrasonic Sensor HC-SR04. Random Nerd Tutorials. URL: https://randomnerdtutorials.com/complete-guide-for-ultrasonic-sensor-hc-sr04/ (дата звернення: 30.05.2026).

ДОДАТКИ

ДОДАТОК А

Посилання на репозиторій з проєктом

Посилання на github-репозиторій з повним лістингом коду, файлом

https://github.com/luvthenika/Smart-RoboCar

ДОДАТОК Б

Посилання на фрагменти дизайну (включно із дизайн-системою) у додатку Figma

https://www.figma.com/design/gGQEYMcXzklHcm2JHDYC4y/VINTAGE-RETRO-SET-OF-PIXEL-ART-ICONS-GAME-UI-PINK--Community-?node-id=2003-762

БІБЛІОГРАФІЧНА ДОВІДКА

Тема дипломної роботи: «Розробка програмного забезпечення системи автономної навігації з розпізнаванням візуальних міток та веб-інтерфейсом керування»

Обсяг пояснювальної записки: 80 аркушів

Дата закінчення роботи 09 червня 2025 р.

Підпис _____