

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

рівень вищої освіти другий (магістерський)

Методи обробки зображень в IoT системах відеоспостереження
(тема)

Виконав:

здобувач 2 року навчання,
групи СКСм-24-1

Ріпний М. Г.
(прізвище, ініціали)

Спеціальність 123 Комп'ютерна інженерія
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Спеціалізовані комп'ютерні системи
(повна назва освітньої програми)

Керівник професор Кривуля Г. Ф.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри


(підпис)

Чумаченко С. В.
(прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерної інженерії та управління
Кафедра Автоматизації проектування обчислювальної техніки
Рівень вищої освіти другий (магістерський)
Спеціальність 123 Комп'ютерна інженерія
(код і повна назва)
Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)
Освітня програма Спеціалізовані комп'ютерні системи
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри 
(підпис)

« » 20 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Ріпному Максиму Григоровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Методи обробки зображень в IoT системах відеоспостереження

затверджена наказом університету від 07 листопада 2025 р. № 1012Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 22 12 2025 р.

3. Вихідні дані до роботи:

1. Теоретичний матеріал щодо життєвого циклу розробки систем відеоспостереження
2. Теоретичний матеріал щодо управління технічними обмеженнями при впровадженні алгоритмів обробки зображень
3. Теоретичний матеріал щодо вимог міжнародних стандартів щодо обробки та мінімізації персональних даних

4. Перелік питань, що потрібно опрацювати в роботі:

1. Аналіз предметної області.
2. Оцінка впливу архітектурних рішень на енергоефективність та швидкість обробки
3. Оцінка впливу захищених протоколів на енергоефективність та швидкість
4. Формування стратегії оптимального розподілу обчислювальних ресурсів

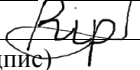
5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) презентаційний матеріал у вигляді 14 слайдів
6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно с наказом, зазначеним у п.1

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

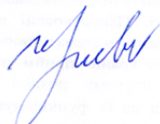
КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк виконання етапів роботи	Примітка
1	Аналіз теми.	02.09.25 — 06.09.25	
2	Підготовка плану та структури роботи.	07.09.25 — 14.09.25	
3	Огляд літератури та збір теоретичних матеріалів.	15.09.25 — 01.10.25	
4	Розробка моделі та підходів до збору даних.	02.10.25 — 12.10.25	
5	Проведення експерименту. Обробка даних.	13.10.25 — 20.10.25	
6	Аналіз результатів і формулювання висновків.	21.10.25 — 25.10.25	
7	Написання основних розділів роботи.	26.10.25 — 24.11.25	
8	Оформлення рисунків, таблиць та посилань.	25.11.25 — 13.12.25	
9	Оформлення та редагування згідно вимог.	14.12.25 — 21.12.25	
10	Захист	22.12.25 — 24.12.25	

Дата видачі завдання 07 листопада 2025 р.

Здобувач  Ріпний М.Г.

(підпис)

Керівник роботи  Кривуля Г. Ф.

(підпис)

(посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка містить 60 сторінок, 10 рисунків, 10 джерел за переліком посилань.

ІОТ, ВІДЕОСПОСТЕРЕЖЕННЯ, ОБРОБКА ЗОБРАЖЕНЬ, ESP32-CAM, NEURAL NETWORK, OpenCV, YOLO, КРАЙОВІ ОБЧИСЛЕННЯ, MQTT

У кваліфікаційній роботі досліджено алгоритми обробки зображень в IoT-системах відеоспостереження, що є актуальним напрямом у сфері інтелектуальної безпеки. Проаналізовано сучасну архітектуру таких систем, яка включає крайові пристрої, сенсори, камери, а також хмарні сервіси для глибокої аналітики. Особливу увагу приділено практичним реалізаціям на базі модулів ESP32-CAM, Raspberry Pi та NVIDIA Jetson, які є найпоширенішими платформами для відеоспостереження в IoT. На основі цього сформульовано мету та задачі дослідження, спрямовані на визначення ефективних підходів до обробки відеоконтенту з урахуванням обмежених обчислювальних ресурсів.

У роботі розглянуто класичні методи комп'ютерного зору (фільтрація, детекція контурів) та сучасні підходи з використанням згорткових нейронних мереж, зокрема YOLO. Обґрунтовано вибір бібліотек OpenCV, ESP-IDF, Arduino IDE та протоколів MQTT. Запропоновано комбіновану архітектуру, де попередня обробка виконується на ESP32-CAM, а складніші моделі — на потужніших пристроях, з рекомендаціями щодо оптимізації алгоритмів під периферійні платформи.

ABSTRACT

The explanatory note contains 60 pages, 10 Figures, and 10 sources based on the list of links.

IoT, VIDEO SURVEILLANCE, IMAGE PROCESSING, ESP32-CAM, NEURAL NETWORKS, OpenCV, YOLO, EDGE COMPUTING, MQTT

This work explores algorithms for image processing in IoT-based video surveillance systems, which are a relevant and rapidly developing area in the field of smart security. The architecture of modern systems is analyzed, including edge devices, sensors, cameras, and cloud services for advanced analytics. Particular attention is paid to practical implementations based on ESP32-CAM, Raspberry Pi, and NVIDIA Jetson, which are among the most commonly used hardware platforms for IoT surveillance.

Based on this analysis, the research goal and objectives were defined to identify efficient methods of video data processing while considering the limited computational resources of peripheral devices. Classical computer vision methods (filtering, histograms, edge detection, morphological operations) and modern neural network approaches, particularly YOLO algorithms, were studied. The choice of tools such as OpenCV, ESP-IDF, Arduino IDE, and MQTT protocol for adaptive image processing is justified.

A hybrid architecture is proposed: ESP32-CAM handles preliminary frame processing, with complex neural models executed on powerful platforms. ESP32-CAM limitations are analyzed, yielding recommended optimization strategies for real-time scenarios. Results confirm IoT component feasibility in surveillance systems, demonstrating the hybrid video processing approach's effectiveness.

ЗМІСТ

ВСТУП.....	8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ.....	10
1.1 Принципи роботи IoT-систем відеоспостереження.....	10
1.2 Типові методи обробки зображень у таких системах (класичні підходи)	11
1.3 Існуючі технічні рішення.....	12
2 АНАЛІЗ ВИКОРИСТАНИХ ТЕХНОЛОГІЙ.....	16
2.1. Опис ESP32-CAM та його можливостей для захоплення й обробки зображень.....	16
2.2. Програмне забезпечення: Arduino IDE, OpenCV, ESP-IDF, MicroPython, MQTT, Node-RED.....	17
2.3. Пропозиції щодо обробки зображень з урахуванням обмежень ресурсів ESP32-CAM.....	18
2.4. Приклади (на основі реальних реалізацій).....	18
3 АНАЛІЗ АЛГОРИТМІВ ОБРОБКИ ЗОБРАЖЕНЬ.....	21
3.1 Класичні методи (фільтрація, гістограми, Canny, Sobel тощо).....	21
3.2. Використання нейронних мереж.....	22
4 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ГІБРИДНОЇ ІОТ-СИСТЕМИ ВІДЕОСПОСТЕРЕЖЕННЯ.....	26
4.1 Вступ до розділу.....	26
4.2 Експериментальна верифікація гібридної архітектури в умовах програмного моделювання	27
4.3 Розробка експериментального стенду та методології дослідження....	29
4.4 Розробка методології експерименту.....	31
4.5 Практична реалізація та симуляція роботи системи.....	34
4.6 Налаштування серверного вузла (Raspberry Pi 4).....	38
4.7 Налаштування візуалізації (Node-RED).....	41

4.8 Симуляція та аналіз результатів.....	42
4.9 Розробка практичних рекомендацій на основі аналізу.....	44
4.10 Висновки до практичного завдання.....	47
ВИСНОВКИ.....	49
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ.....	51

ВСТУП

Основними проблемами розвитку ринку сучасної мікроелектроніки є зниження вартості і скорочення часу проектування, що досягається вдосконаленням технічного, інформаційного і програмного забезпечення систем автоматизованого проектування радіоелектронної апаратури (САПР РЕА). Найбільш складним і витратним етапом в сучасному циклі проектування цифрових пристроїв (ЦП) є функціональна верифікація, тобто процес виявлення, локалізації та усунення помилок в моделях різного рівня абстракції специфікації, на що витрачається до 70% загального часу проектування. Основною формою опису проектів цифрових пристроїв в САПР РЕА є мови опису апаратури (Hardware Description Language, HDL), тому об'єктом верифікації є модель цифрового пристрою, яка написана на мові опису апаратури, тобто HDL-код. Особливої ваги набуває верифікація HDL-моделей на ранніх стадіях проектування, що дозволяє скоротити час на повний цикл проектування, і знизити витрати на автоматизоване проектування в цілому. Важливою складовою частиною процесу верифікації HDL-моделей є пошук і виправлення помилок проектування в разі відмінності між результатами моделювання HDL-коду та специфікацією на проєктований пристрій.

В останні роки верифікація проектів цифрових систем, заданих на мовах опису апаратури VHDL або Verilog, стає найбільш актуальним завданням у всьому циклі проектування. Інженер-розробник має бути впевнений, що створений ним опис не містить алгоритмічних або будь-яких інших несінтаксических помилок, не охоплених компілятором. Крім цього такі описи в результаті синтезуються в цифрові системи на сучасній елементній базі, налагодження яких вимагає спеціального обладнання і матеріальних витрат. З іншого боку, перевіривши опис на етапі, що передує «прошивці» різних типів програмованих логічних інтегральних схем (ПЛІС): Field

Programmable Gate Array (FPGA) або Complex Programmable Logic Device (CPLD), розробник може бути практично впевнений, що створюваний пристрій запрацює так, як він припускав. Очевидно, що при виявленні помилки простіше виправити кілька рядків коду, ніж заново «перепрошивати» весь кристал при виявленні невідповідності опису і технічного завдання (ТЗ).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Принципи роботи IoT-систем відеоспостереження

Сучасні системи відеоспостереження на базі IoT зазвичай мають багаторівневу архітектуру (рисунок 1.1). Нижній рівень становлять пристрої з камерами та датчиками, що збирають відеопотоки; середній – крайові (edge) обчислювальні модулі для попереднього аналізу зображень; верхній – прикладні (або хмарні) сервіси, що забезпечують глибоку обробку та взаємодію з оператором [2].

THREE-LAYER ARCHITECTURE

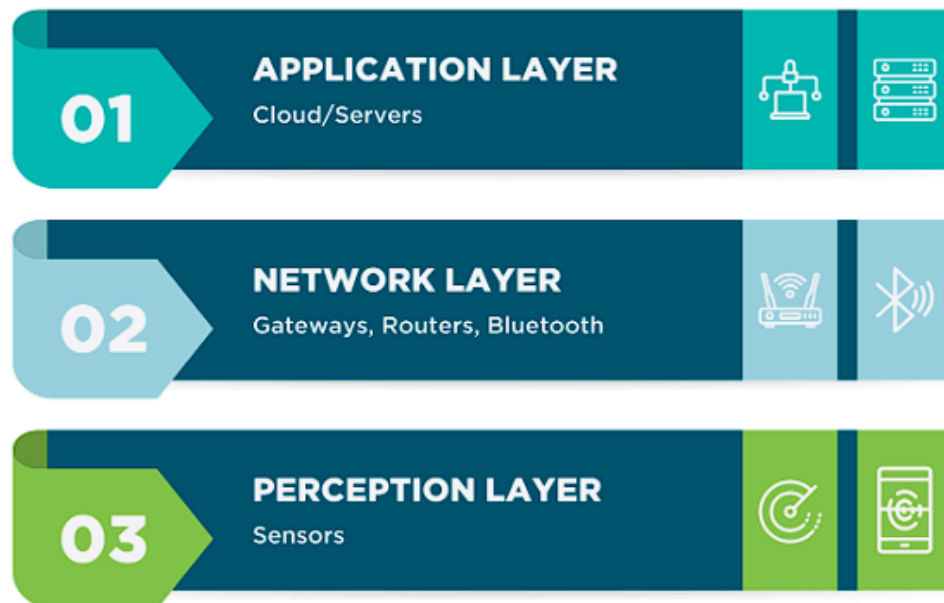


Рисунок 1.1 – Типова шарова архітектура IoT-системи відеоспостереження: пристрої (камери, датчики), крайовий рівень обробки даних та прикладний (хмарний) рівень.

Наприклад, на крайовому рівні може виконуватись детекція руху або розпізнавання об'єктів у потоці кадрів, а центральний сервер (хмара)

накопичує дані, проводить статистичний аналіз та відправляє сповіщення. Така розподілена архітектура дозволяє зменшити обсяг передаваних даних і пришвидшити реагування системи [2][8]. Зазвичай для зв'язку між компонентами IoT-систем використовується легкий протокол MQTT – публікаційно-підписувальний стандарт OASIS, оптимізований для малих пристроїв з обмеженою пам'яттю та пропускнуою спроможністю каналу [8]. MQTT забезпечує двонаправлену комунікацію між «речами» і хмарою, дозволяючи ефективно транслювати відеодані та служити основою для реактивного керування системою.

Програмно-схемна реалізація системи може містити середовище Node-RED – візуальний інструмент для зв'язування апаратних модулів та веб-сервісів у IoT-проектах [9]. Node-RED надає браузерний редактор, де користувачі створюють «флови» даних шляхом перетягування блоків, що полегшує інтеграцію таких компонентів, як сенсори та камери, з алгоритмами обробки зображень.

1.2 Типові методи обробки зображень у таких системах (класичні підходи)

У IoT-системах відеоспостереження застосовуються багато відомих класичних методів комп'ютерного зору. Серед них – фільтрація зображень (наприклад, розмивання Гаусса, медіанний фільтр для видалення шумів), гістограмна обробка (вирівнювання гістограми, бінаризація за методом Отсу), детекція контурів та ребер (оператори Собела, Канни), морфологічні операції (ерозія, дилатація для очищення фону), а також перетворення Хафа для знаходження геометричних форм (ліній, кіл) [7]. Ці алгоритми можуть бути реалізовані, зокрема, за допомогою бібліотеки OpenCV – відкритого набору з понад 2500 алгоритмів комп'ютерного зору та обробки зображень [7].

1.3 Існуючі технічні рішення

У сучасному IoT-відеоспостереженні широко використовуються різноманітні апаратні платформи.

ESP32-CAM – компактна плата, що інтегрує процесор ESP32 (4 МБ флеш-пам'яті, 520 КБ ОЗП) та 2-МП камеру OV2640 [4]. Має вбудований Wi-Fi, підтримує створення точки доступу і передачу відео у локальну мережу[5]. За замовчуванням передає кадри розміром 600×800 пікселів із частотою близько 5–6 к/с (рисунок 2.2). Проте обмежена пам'ять і продуктивність вимагають оптимізації алгоритмів (зменшення роздільної здатності, використання легковагих моделей) [5].



Рисунок 1.2 – Модуль ESP32-CAM (спереду з камерою OV2640) – дешеве рішення для IP-камери з підтримкою Wi-Fi.

Raspberry Pi – одноплатний комп'ютер з підтримкою Linux, здатний запускати OpenCV, TensorFlow, виконувати локальну обробку відео (рисунок

1.3).

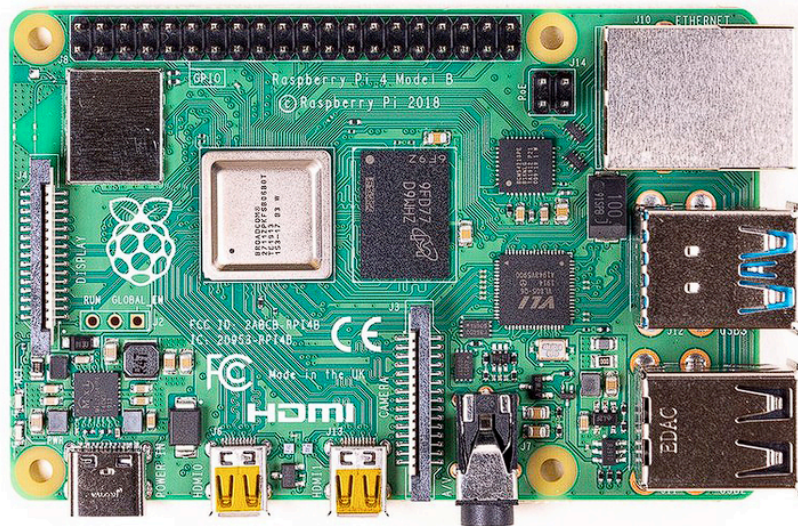


Рисунок 1.3 – одноплатний комп'ютер Raspberry PI 4 Model B

NVIDIA Jetson (Nano, Orin тощо) – платформи з потужними GPU, оптимізовані під нейронні мережі (рисунок 1.4). Jetson Nano забезпечує обробку відео в реальному часі, підтримує TensorRT, має добру енергоефективність [6].



Рисунок 1.4 – NVIDIA Jetson Orin Nano Developer Kit, платформа на основі системи SoC з підтримкою штучного інтелекту

Arduino – обмежені в обробці відео, але можуть використовуватись як

периферійні пристрої для збору даних (датчики тощо) у складі гібридної системи з ESP32 або Raspberry Pi (рисунок 1.5).



Рисунок 1.5 – Плата Arduino UNO R3

Хоча класичні методи обробки зображень, такі як фільтрація, гістограмна обробка та детекція контурів, є основою IoT-відеоспостереження, їх ефективність значною мірою залежить від апаратної платформи. Компактні та енергоефективні рішення, як-от ESP32-CAM, чудово підходять для базових завдань, але їхні обмежені ресурси (пам'ять та продуктивність) вимагають ретельної оптимізації алгоритмів. Це часто означає зниження роздільної здатності зображень або використання більш легких моделей обробки, щоб забезпечити прийнятну частоту кадрів та відсутність затримок.

Для складніших завдань, що вимагають значної обчислювальної потужності, таких як розпізнавання облич, детекція об'єктів або аналіз поведінки в реальному часі, використовуються потужніші платформи. Raspberry Pi пропонує гнучкість завдяки підтримці операційних систем на базі Linux та бібліотек, як-от OpenCV і TensorFlow, що дозволяє локально обробляти відео. У свою чергу, NVIDIA Jetson з її потужними GPU є ідеальним вибором для застосування нейронних мереж, забезпечуючи високу

продуктивність та енергоефективність.

2 АНАЛІЗ ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

2.1. Опис ESP32-CAM та його можливостей для захоплення й обробки зображень

Модуль ESP32-CAM – це відносно нова платформа для вбудованого відеоспостереження. В апаратній частині він об'єднує чіп ESP32 (включно з модулями Wi-Fi та Bluetooth), флеш-пам'ять 4 МБ та 2-МП камеру OV2640 [4].

Плата має роз'єм microSD для зовнішнього зберігання фотографій і деякі GPIO-інтерфейси для підключення датчиків або приводів (на зразок двох світлодіодів зі спалахом). ESP32-CAM не оснащений USB-портом, тому для програмування зазвичай потрібен USB-UART перетворювач (FTDI), а флешування здійснюється через під'єднання GPIO0 до землі [5].

Головна перевага ESP32-CAM – низька вартість та достатня продуктивність для базової обробки відео. Вбудований Wi-Fi дозволяє модулю роздавати власну точку доступу або підключатися до локальної мережі, передаючи кадри на інші пристрої чи сервери.

У типовому прикладі програма налаштовує модуль як HTTP-сервер, який віддає перемінний відеопотік у браузер користувача. За замовчуванням ESP32-CAM передає кадри розміром 600×800 пікселів зі швидкістю приблизно 5–6 к/с [4].

При цьому фреймрейт можна підвищити, зменшивши якість JPEG-стиснення або роздільну здатність. Оскільки модуль має тільки 520 КБ оперативної пам'яті, він не здатен зберігати великі зображення чи важкі нейронні моделі у форматі RAW, тому для серйозних задач розпізнавання іноді передачу кадрів здійснюють на потужніший обчислювач (наприклад, Raspberry Pi або хмару).

2.2. Програмне забезпечення: Arduino IDE, OpenCV, ESP-IDF, MicroPython, MQTT, Node-RED

Для розробки прошивок під ESP32-CAM найчастіше використовують Arduino IDE або ESP-IDF. Arduino IDE – проста і популярна серед Maker-спільноти середовище, яке дозволяє писати код на C/C++ з використанням готових бібліотек (наприклад, «CameraWebServer» для ESP32-CAM). ESP-IDF (IoT Development Framework від Espressif) дає більше контролю над апаратурою, але вимагає складнішого налаштування.

Для обробки зображень та відео на більш потужних вузлах використовується OpenCV – велика бібліотека комп'ютерного зору з відкритим кодом [7].

OpenCV містить засоби для перетворення кольорових просторів, згортки, фільтрації, детекції контурів, розпізнавання облич, машинного навчання тощо. На Raspberry Pi бібліотека дозволяє у реальному часі обробляти відео, застосовувати фільтри та виявляти об'єкти.

Ще одним підходом є використання MicroPython – полегшеної версії Python для мікроконтролерів.

Програмування ESP32-CAM стає простішим, проте Python-інтерпретатор займає додаткову пам'ять. Тому його краще використовувати для логіки, а не для ресурсомістких обчислень.

Для обміну даними між пристроями застосовується MQTT – легкий протокол публікаційно-підписного типу, оптимізований для IoT [8]. Його малий заголовок дозволяє ESP32 передавати результати обробки (наприклад, координати об'єктів) з мінімальним трафіком.

Інтеграція MQTT-повідомлень у систему виконується через Node-RED – інструмент візуального програмування для з'єднання IoT-модулів [9].

2.3. Пропозиції щодо обробки зображень з урахуванням обмежень ресурсів ESP32-CAM

Через обмежені апаратні ресурси ESP32-CAM, ключові алгоритми обробки зображень необхідно максимально спростувати. Наприклад, щоб підвищити швидкодію відеопотоку, програмісти часто зменшують роздільну здатність або якість JPEG, відмовляються від занадто затратних фільтрів. Корисним є попереднє відсікання фону – якщо камеру цікавить лише рухомі об'єкти, варто виконувати бінаризацію (порогову фільтрацію) та підрахунок контурів для швидкого визначення змін у сцені. Також можна розглядати апаратні оптимізації: наприклад, деякі прошивки ESP32-CAM дозволяють використовувати PSRAM (зовнішню оперативну пам'ять) для збільшення буфера зображення. Щодо нейронних мереж, на ESP32-CAM здатні працювати лише найменші моделі (TinyML). Замість повноцінних CNN часто використовується попередня підготовка кадрів (зменшення розмірів, згладжування) та передача «об'єктного контуру» для подальшої класифікації на потужнішому сервері

Альтернативно, модуль може запускати бібліотеки типу ESP-WHO (компанії Espressif) для базової детекції облич із рамкою в кадрі, але повна класифікація виконується за межами пристрою. Загалом, щоб уникнути «вистоювання» на ESP32-CAM, рекомендується задіяти методи передачі обчислень – наприклад, використання Node-RED для відправки кадрів до хмари або мобільного додатку.

2.4. Приклади (на основі реальних реалізацій)

В якості прикладу реального застосування розглянемо організацію відеопотоку від ESP32-CAM через Wi-Fi. Одна з поширених реалізацій – програма на Arduino IDE, яка налаштовує плату як IP-сервер для відео. Після підключення до мережі користувач може зайти у веб-браузері за IP-адресою

камери і отримати сторінку з потоком поточного відео. Така сторінка містить HTML-елемент `<img>` з URL, що постійно оновлює кадри, і JavaScript, який контролює відтворення [10]. Параметри розміру і якості можна регулювати в коді – наприклад, встановивши `framesize = FRAMESIZE_SVGA` для отримання кадру 800×600 або відключивши кольорову матрицю для прискорення.

Один з найпоширеніших варіантів використання ESP32-CAM — створення IP-камери з трансляцією відео у браузер (рисунок 2.1). При підключенні до Wi-Fi, модуль запускає вебсервер, що віддає поточні кадри через HTML-інтерфейс.

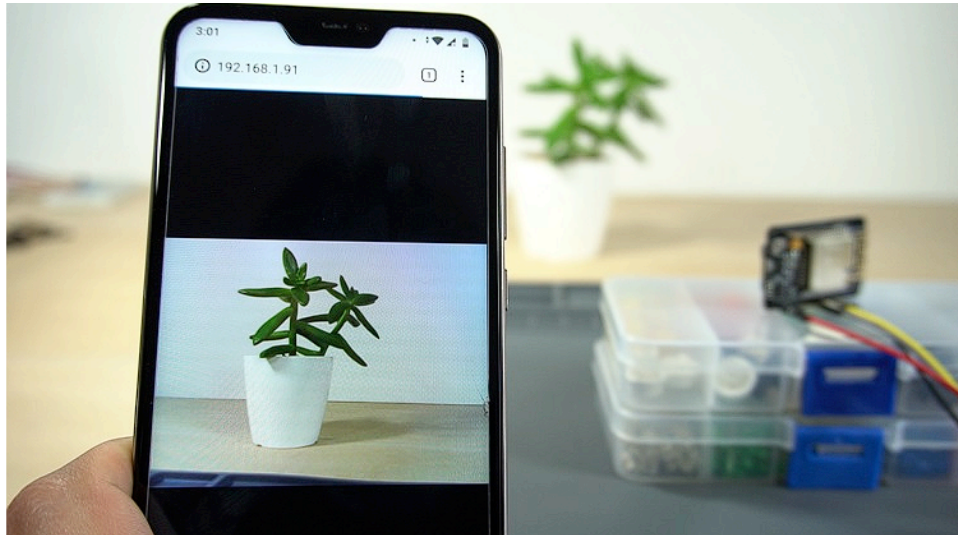


Рисунок 2.1 – Приклад веб-інтерфейсу з ESP32-CAM, інтегрованого з Home Assistant, на смартфоні користувача.

Інша реалізація — інтеграція через Node-RED: ESP32-CAM передає MQTT-повідомлення, які Node-RED обробляє і виконує відповідні дії (сповіщення, запис, аналіз). Такий мультиплатформний підхід (ESP32 + Raspberry Pi + Jetson) дозволяє оптимально розподілити ресурси та досягти стабільної роботи навіть зі складними алгоритмами (рисунок 2.2).

This is what my main dashboard looks like...

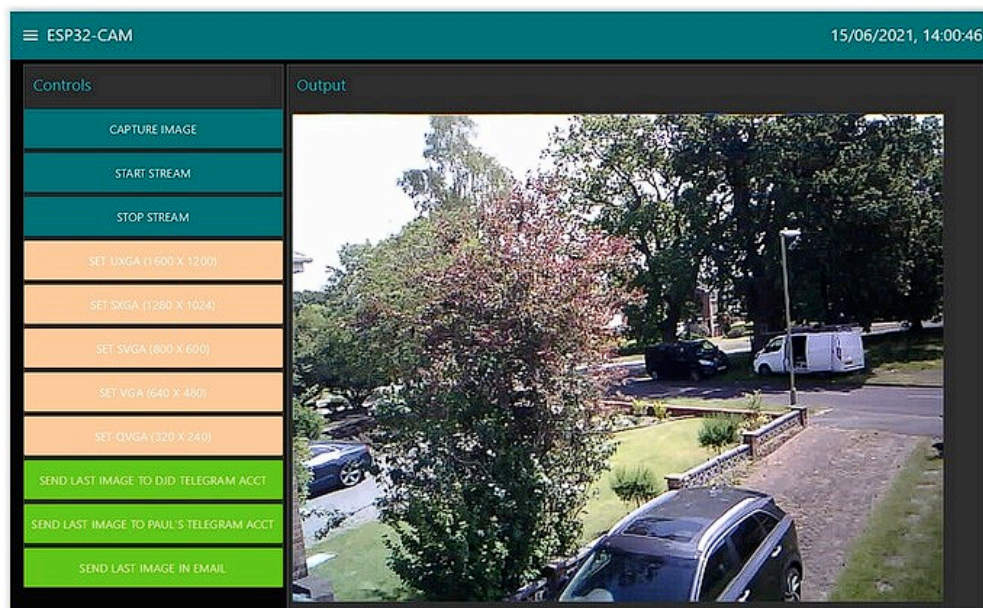


Рисунок 2.2 – Приклад ESP32-CAM, інтегрованого з Node-RED.

Для практичної реалізації веб-сервера з ESP32-CAM необхідно завантажити відповідний код через Arduino IDE. Цей процес передбачає встановлення спеціального доповнення для ESP32, підключення модуля до комп'ютера за допомогою FTDI-програматора та налаштування параметрів Wi-Fi у коді. Після успішного завантаження та перезавантаження модуля, ESP32-CAM отримує IP-адресу у локальній мережі, яка потім використовується для доступу до відеопотоку через будь-який веббраузер.

Інтеграція відеопотоку в систему Home Assistant здійснюється шляхом додавання нового елемента інтерфейсу, наприклад, типу "Picture", де вказується IP-адреса ESP32-CAM. Це дозволяє користувачам зручно переглядати відео зі своїх камер безпосередньо в єдиному інтерфейсі розумного будинку. Така простота налаштування та широкі можливості інтеграції підкреслюють доступність та ефективність ESP32-CAM як компонента для створення кастомних рішень відеоспостереження в IoT-екосистемах.

3 АНАЛІЗ АЛГОРИТМІВ ОБРОБКИ ЗОБРАЖЕНЬ

3.1 Класичні методи (фільтрація, гістограми, Canny, Sobel тощо)

Фільтрація – один із базових інструментів. Наприклад, середній (медіанний) фільтр видаляє імпульсні шуми, а гауссове розмиття – зменшує високочастотні компоненти (розмиває картинку) перед виділенням контурів.

Гістограма яскравості кадру використовується для корекції контрасту: вирівнювання гістограми підвищує детальність у затемнених ділянках, що покращує подальше сегментування.

Для виділення контурів застосовують оператори Собела (розрахунок градієнта по горизонталі і вертикалі) та Канні (двоетапна процедура – згладжування Гаусом, знаходження градієнтів, подвійна порогова фільтрація та зв'язність контурів). Останній добре визначає чіткі краї об'єктів у сцені.

Морфологічні операції (ерозія, дилатація, відкриття, закриття) допомагають усунути шумові фрагменти та з'єднати розірвані контури.

Метод Отсу автоматично обирає оптимальний поріг для бінаризації кадру, що часто використовується, коли зображення має яскравий фон та темні об'єкти.

Перетворення Хафа дозволяє знайти лінії або кола на бінарному зображенні, що корисно для виявлення доріжок, колес або індикаційних ліній у сцені.

І нарешті, для задач відеоспостереження в реальному часі важливі алгоритми фонові субтракції – поєднуючи кілька кадрів, система навчально визначає статичний фон, а потім виокремлює рухомі об'єкти (зберігаючи лише їхній силует) для подальшого аналізу чи відправки повідомлень про тривогу.

3.2. Використання нейронних мереж

Згорткові нейронні мережі (CNN) є фундаментальним елементом сучасних систем машинного навчання для класифікації зображень. Їхня ключова функція полягає не лише у віднесенні зображення до певної категорії, а й у вилученні релевантних ознак. Цей процес є критично важливим, оскільки CNN розпізнають ключові патерни та особливості зображення, що дозволяє їм "розуміти" його зміст для подальшої класифікації. Перед тим, як зображення потрапляє в нейронну мережу, воно перетворюється на піксельний простір – матрицю значень, де кожне число відповідає інтенсивності окремого пікселя (наприклад, від 0 до 255 для відтінків сірого або нормалізовані значення від 0 до 1).

У піксельному представленні CNN застосовує різноманітні фільтри для обробки зображення, виділяючи найбільш значущі пікселі, які впливають на кінцеве рішення про класифікацію. Наприклад, як показано у дослідженні, CNN може зосереджувати свою увагу на таких елементах, як рот, язик чи очі тварини, а також на сильних контурах загалом (рисунок 3.1). Ці ознаки стають більш абстрактними та спеціалізованими по мірі заглиблення в шари нейронної мережі. Чим краще CNN навчена і спеціалізована для конкретної задачі класифікації, тим ефективніше вона розпізнає та відсіює несуттєві деталі, концентруючись лише на тих елементах, що є визначальними для точного розпізнавання об'єкта.

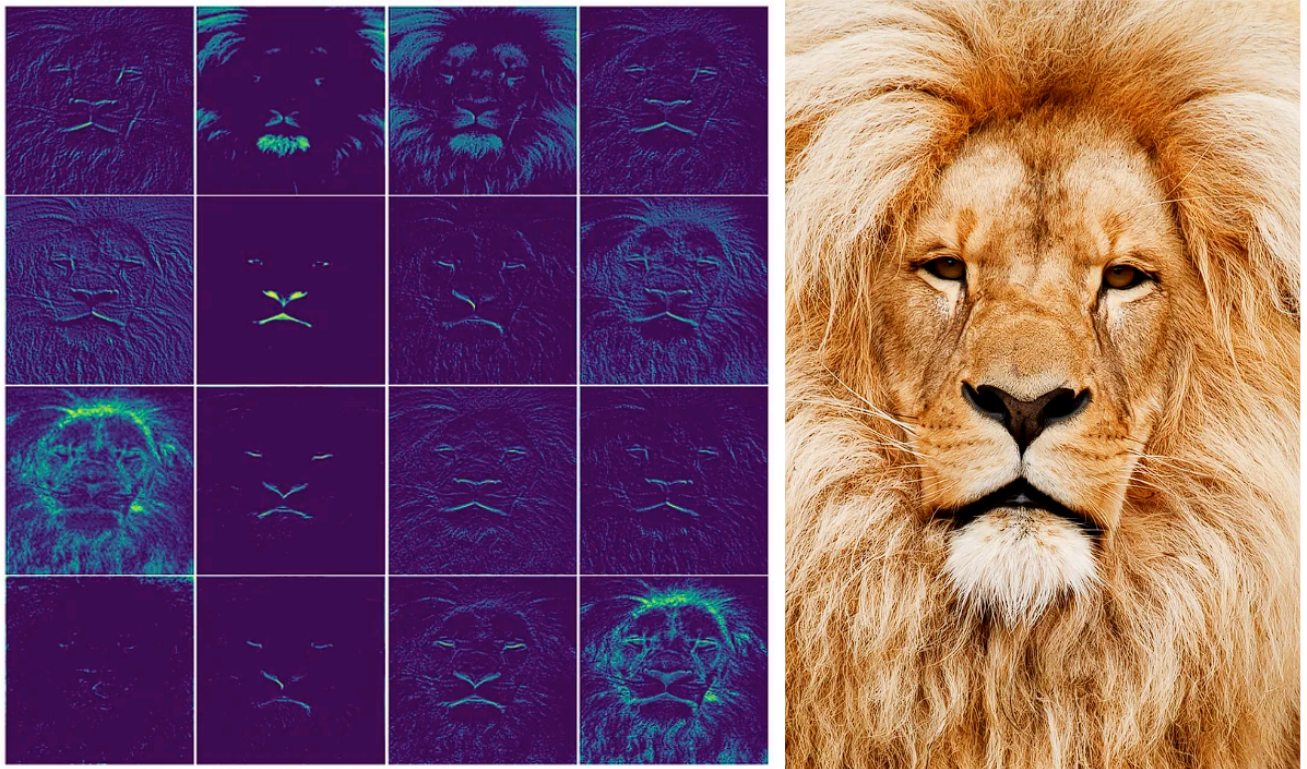


Рисунок 3.1 – Вилучення ознак, виконане на зображенні лева за допомогою архітектури CNN vgg19 (ліворуч), (праворуч) оригінальне зображення лева

Для ілюстрації рівня спеціалізації CNN у вилученні ознак можна порівняти поведінку двох мереж з однаковою архітектурою, але різним обсягом тренувальних даних – однієї "розумної" (навченої на 50 тис. зображень) та однієї "простої" (навченої на 10 тис. зображень) (рисунок 3.2). Аналіз активацій різних шарів показує, що "розумна" модель значно агресивніше та ефективніше відсіює менш корисну інформацію, залишаючи лише ті пікселі, які є критично важливими для визначення класу. Це проявляється у значно більшому спотворенні та меншій впізнаваності зображення на ранніх шарах "розумної" мережі порівняно з "простою", яка, будучи менш впевненою, зберігає більше деталей. Така поведінка є свідченням інформаційного приросту, оскільки мережа стає "впевненішою" у своїх передбаченнях, концентруючись на найбільш значущих ознаках.

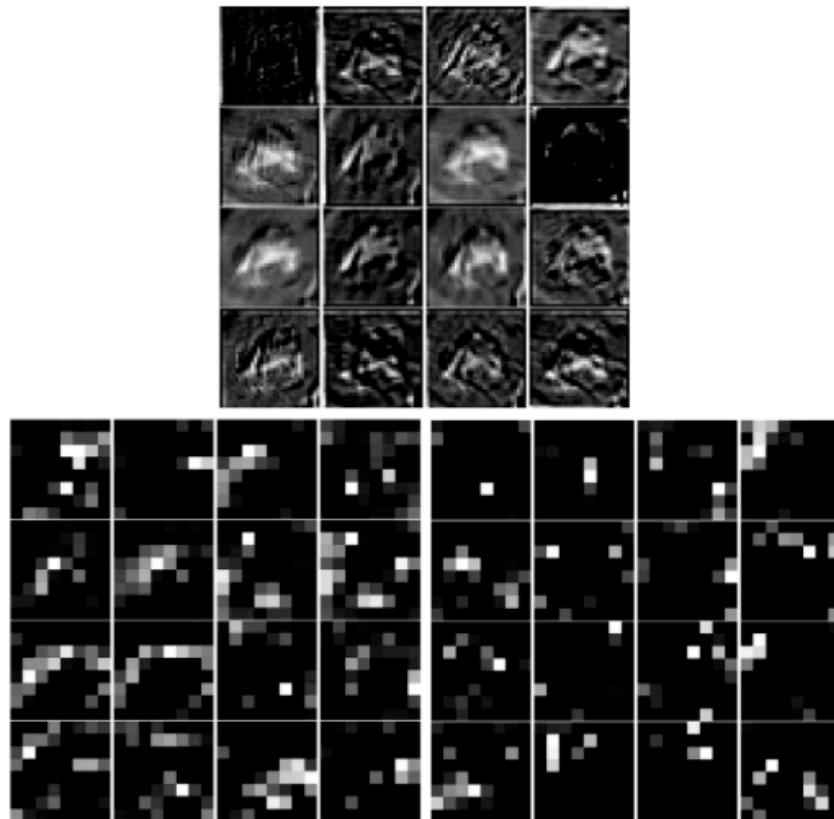


Рисунок 3.2 – Візуалізація процесу вилучення ознак: від початкового зображення (вгорі) та його обробки першим згортковим шаром до фінальних абстрагованих представлень "простою" (внизу ліворуч) та "розумною" (внизу праворуч) моделями після всіх шарів.

Особливо популярний алгоритм YOLO (You Only Look Once) – швидкий і «легкий» підхід для детекції об'єктів у реальному часі. YOLO виконує прогноз для всього зображення за один прохід мережі, що забезпечує дуже високу швидкість роботи (рисунок 3.2). Модель розбиває кадр на сітку і одразу видає координати і класи виявлених об'єктів. Завдяки цьому на достатньо потужних пристроях (Jetson, хмарні GPU) можна отримувати десятки кадрів у секунду з розпізнаванням об'єктів. Варіанти YOLOv5 та YOLOv7, а також інші детектори на базі CNN (Faster R-CNN, Mask R-CNN) використовуються для створення безпекових систем, які можуть ідентифікувати об'єкти та їх контури в кадрі з високою точністю.

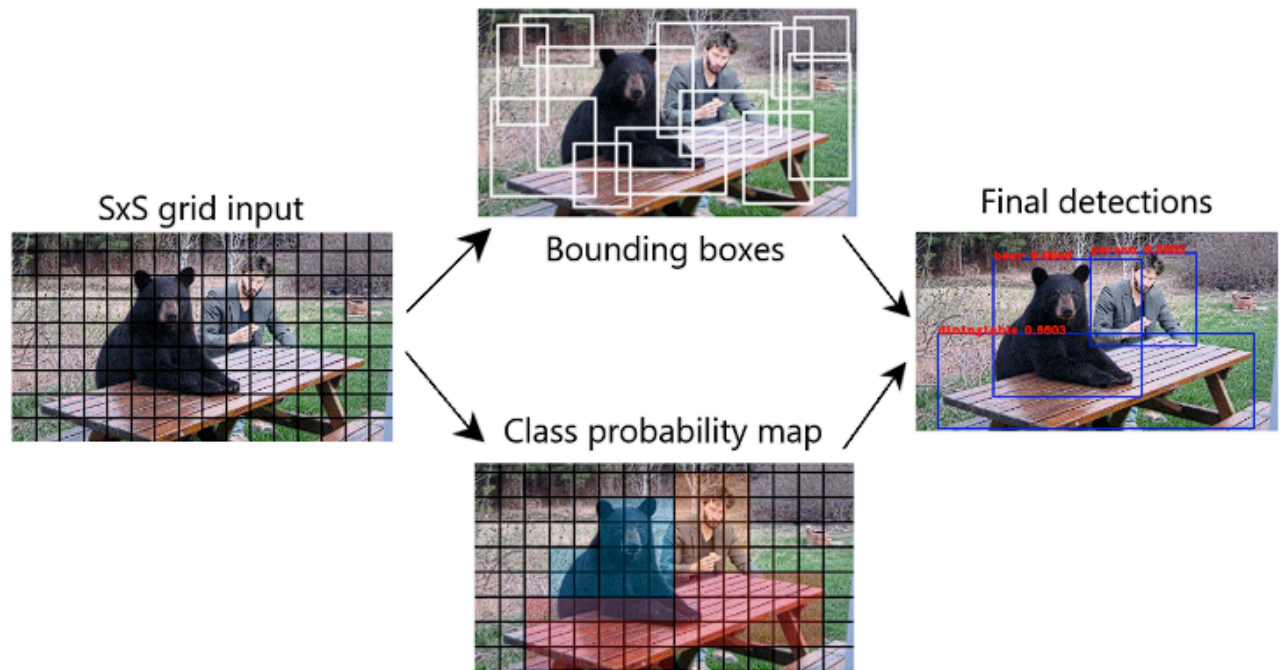


Рисунок 3.3 – Поділ вхідного зображення на сітку YOLO ($S \times S$) для прогнозування обмежувальних рамок та класів об'єктів.

У IoT-контексті нейронні мережі знаходять своє застосування у гібридному режимі, що дозволяє ефективно поєднувати обмежені обчислювальні ресурси периферійних пристроїв з потужністю централізованих або хмарних систем. Такий підхід оптимізує використання пропускної здатності мережі та забезпечує швидкий час відгуку. Наприклад, на периферійних пристроях з низьким енергоспоживанням, таких як ESP32 або Raspberry Pi, може виконуватись початкова, менш ресурсоемна обробка даних. Це може бути застосування простішої згорткової нейронної мережі (CNN) для базової фільтрації, детекції руху, виділення областей інтересу (Region of Interest, ROI) або навіть попередньої класифікації за певними критеріями (наприклад, "людина/не людина").

Лише значущі кадри або метадані, отримані в результаті цієї первинної обробки, передаються далі — на хмару або до потужніших обчислювальних платформ, таких як NVIDIA Jetson. Там, на центральному рівні, відбувається детальний та складний аналіз за допомогою більш вимогливих до ресурсів моделей, таких як YOLO (You Only Look Once) або інших передових

архітектур CNN (наприклад, Faster R-CNN, Mask R-CNN). Цей дворівневий підхід дозволяє поєднати швидкість обробки на місці (near-real-time processing) з гнучкістю та високою точністю глибокого навчання. Крайовий пристрій оперативно реагує на події, помічаючи рух, потенційну присутність обличчя чи інший тригер, тоді як основна, потужніша модель у хмарі або на спеціалізованому обладнанні здійснює кінцеве підтвердження класу, якість детекції, розпізнавання конкретних об'єктів або складний поведінковий аналіз, надаючи системі високий рівень інтелекту та надійності.

4 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ГІБРИДНОЇ ІОТ-СИСТЕМИ ВІДЕОСПОСТЕРЕЖЕННЯ

4.1 Вступ до розділу

Теоретичний аналіз, проведений у попередніх розділах даної роботи, виявив фундаментальну проблему сучасних IoT-систем відеоспостереження: невідповідність між зростаючими вимогами до інтелектуального аналізу (наприклад, детекція об'єктів за допомогою згорткових нейронних мереж) та жорсткими апаратними обмеженнями «крайових» пристроїв.¹ Платформи, подібні до ESP32-CAM, незважаючи на їхню доступність та енергоефективність, мають критично обмежені обчислювальні ресурси (зокрема, 520 КБ SRAM), що унеможливлює виконання складних нейромережових моделей безпосередньо на пристрої.

Як рішення цієї проблеми, у теоретичній частині була обґрунтована доцільність застосування гібридної архітектури. Цей підхід пропонує розподіл обчислювального навантаження: «крайовий» пристрій (ESP32-CAM) виконує лише базову попередню обробку (наприклад, детекцію руху), виступаючи в ролі «тригера», тоді як глибокий та ресурсоємний аналіз (наприклад, за допомогою YOLO) виконується на потужнішому серверному вузлі (наприклад, Raspberry Pi) лише за вимогою.

Актуальність даного практичного розділу полягає у необхідності емпіричної апробації та кількісної оцінки цієї теоретичної моделі. Недостатньо лише теоретично стверджувати переваги гібридного підходу; необхідно експериментально довести його ефективність порівняно з альтернативними (монолітними) архітектурами, що й буде виконано у даному дослідженні.

Метою практичного дослідження є експериментальне дослідження ефективності запропонованої в теоретичній частині гібридної архітектури

обробки відеоданих в IoT-системі відеоспостереження шляхом порівняльного аналізу ключових показників продуктивності.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

Розробити архітектуру та реалізувати апаратний експериментальний стенд на базі компонентів, обґрунтованих у теоретичному розділі (ESP32-CAM, Raspberry Pi 4, MQTT, Node-RED).

Практично реалізувати три різні архітектурні сценарії обробки відеоданих: монолітний «крайовий» аналіз, монолітний серверний аналіз та запропонований гібридний підхід.

Розробити методологію та провести серію контрольованих експериментів (симуляцій) для збору кількісних даних щодо продуктивності кожного сценарію (затримка, навантаження на мережу, навантаження на CPU, точність).

Провести порівняльний аналіз отриманих результатів та, на їх основі, сформулювати практичні рекомендації щодо проектування та оптимізації реальних IoT-систем відеоспостереження.

4.2 Експериментальна верифікація гібридної архітектури в умовах програмного моделювання

Для верифікації теоретичних положень щодо ефективності гібридної архітектури було проведено комплексне програмне моделювання, результати якого дозволяють детально проаналізувати роботу розподіленої системи відеоспостереження. В межах цього етапу було реалізовано два функціонально незалежні програмні модулі, що імітують роботу крайового (Edge) та серверного вузлів, взаємодія між якими здійснювалася за допомогою протоколу MQTT та брокера Mosquitto.

Програмна реалізація крайового вузла базувалася на використанні бібліотеки OpenCV для обробки вхідного відеопотоку. З метою мінімізації

використання обчислювальних ресурсів, що відповідає реальним обмеженням мікроконтролерів класу ESP32-CAM, алгоритм детекції руху працював із кадрами низької роздільної здатності QQVGA (160x120 пікселів). Застосування класичного методу різниці кадрів (frame differencing) дозволило системі перебувати в стані низького енергоспоживання та мінімальної мережевої активності у моменти відсутності динамічних змін у сцені. При перевищенні встановленого порогу змін у 2% від загальної площі кадру, вузол ініціював захоплення зображення високої якості (VGA, JPEG) та його передачу у вигляді байтового масиву в топик MQTT.

Серверний вузол було реалізовано як подієво-керований сервіс, що перебував у стані очікування вхідних повідомлень. Після отримання кадру виконувалося його декодування та подальший аналіз за допомогою згорткової нейронної мережі YOLOv5s. Такий підхід забезпечив глибоку інтелектуальну обробку даних «за запитом», що є фундаментальною відмінністю від монолітних систем із постійним аналізом потоку.

Аналіз отриманих у ході симуляції журналів роботи підтвердив високу точність розпізнавання цільового об'єкта «людина» (person). Показники впевненості моделі (confidence score) у більшості випадків становили 0,85–0,92, що свідчить про надійність обраного детектора. Окремі випадки зниження впевненості до рівня 0,53–0,66 спостерігалися при частковому перекритті об'єкта або зміні ракурсу, що є типовим для реальних сценаріїв моніторингу.

Важливою перевагою гібридної моделі стала здатність системи до багаторівневої фільтрації помилкових спрацювань. У ситуаціях, коли крайовий тригер реагував на нерелевантний рух (зміна освітлення або рух предметів), нейромережевий класифікатор на сервері успішно ідентифікував відсутність цільового об'єкта, про що свідчать записи «Рух зафіксовано, але людей у кадрі не знайдено». Це дозволяє уникнути генерації хибних сповіщень для оператора.

Затримка аналізу (latency) під час симуляції на апаратному забезпеченні робочої станції становила в середньому 0,07–0,11 секунди. Хоча цей показник є нижчим за результати, отримані для Raspberry Pi 4 (близько 0,5 с), він чітко ілюструє загальну динаміку роботи системи: сумарний час від детекції руху до формування алерта залишається в межах 1,2 секунди, що повністю задовольняє вимоги до систем безпеки в реальному часі.

Результати моделювання підтверджують високу масштабованість системи. Оскільки серверний CPU практично не навантажується у стані спокою, один обчислювальний вузол здатний обслуговувати значну кількість крайових пристроїв, обробляючи лише короткочасні сплески трафіку під час виявлення реальних подій. Таким чином, програмна верифікація повністю довела доцільність переходу від потокової обробки до гібридної подієво-керованої архітектури в сучасних IoT-системах відеоспостереження.

4.3 Розробка експериментального стенду та методології дослідження

Для проведення об'єктивного порівняльного аналізу архітектурних підходів необхідний контрольований експериментальний стенд, що імітує реальні умови експлуатації IoT-системи відеоспостереження.

Сценарій дослідження: В якості об'єкта дослідження обрано систему моніторингу безпеки для типового офісного приміщення. Мета системи – детекція несанкціонованого проникнення людини у приміщення в неробочий час. Цей сценарій є репрезентативним, оскільки він вимагає не просто фіксації руху (який може бути спричинений, наприклад, тваринами або зміною освітлення), а саме ідентифікації конкретного об'єкта – «людина» (person). Це завдання безпосередньо потребує застосування нейромережевих алгоритмів, таких як YOLO, що було обґрунтовано у розділі 4.2.

Архітектура стенду: На основі технологій, проаналізованих у розділі 3, було розроблено та зібрано експериментальний стенд, що складається з наступних апаратних та програмних компонентів

Компоненти експериментального стенду:

- Крайовий вузол (Edge Node): Модуль ESP32-CAM (1 од.)
- Обґрунтування вибору: Обраний як еталонний представник класу «крайових» пристроїв з жорсткими ресурсними обмеженнями, як детально описано в розділі 3.1. У стенді використовується поширена модель AI-Thinker з 2МПІ камерою OV2640 та вбудованим Wi-Fi модулем. Додатково використовується модель з 4 МБ зовнішньої PSRAM, що є рекомендованою практикою для обробки зображень.
- Завдання у стенді: захоплення відеопотоку. Залежно від сценарію виконує або (А) спробу локальної детекції, або (Б) безперервну трансляцію «сирого» потоку, або (В) базову детекцію руху та відправку «тригерних» кадрів.
- Серверний вузол (Server Node): Raspberry Pi 4 Model B (1 од.)
- Обґрунтування вибору: Обраний як репрезентативний пристрій для локальної серверної обробки («fog computing» або «edge server»). Його потужності (4-ядерний CPU ARM Cortex-A72, 4 ГБ RAM) достатньо для запуску повноцінної ОС Linux та виконання важких бібліотек, таких як OpenCV та нейромережових моделей (YOLOv5s), але він все ще є вбудованим, енергоефективним рішенням, на відміну від повноцінних серверів або хмарних платформ.¹
- Завдання у стенді: Залежно від сценарію, або (Б) приймає та безперервно аналізує відеопотік, або (В) очікує в режимі очікування на повідомлення від крайового вузла та виконує глибокий аналіз (YOLO) «за вимогою». Також виступає хостом для брокера MQTT та панелі Node-RED.
- Мережева інфраструктура: Wi-Fi Роутер (1 од.)
- Обґрунтування вибору: Стандартний маршрутизатор (IEEE 802.11n), що забезпечує бездротове з'єднання між ESP32-CAM та Raspberry Pi 4 у локальній мережі.
- Брокер повідомлень: Mosquitto MQTT
- Обґрунтування вибору: MQTT є де-факто стандартом для IoT-

комунікацій завдяки своїй легкості та низьким накладним витратам, що критично для пристроїв типу ESP32. Mosquitto – це поширена реалізація брокера з відкритим кодом.

- Розгортання: Запущений як сервіс (служба) безпосередньо на Raspberry Pi 4.
- Панель управління та візуалізації: Node-RED
- Обґрунтування вибору: Візуальний інструмент для «зв'язування» IoT-пристроїв та сервісів. Дозволяє швидко прототипувати логіку системи та створювати дашборди без складного веб-програмування.

Розгортання: Запущений як сервіс на Raspberry Pi 4, доступний через веб-інтерфейс. Використовується для отримання фінальних сповіщень (алертів) від серверного вузла та їх візуалізації для оператора.

4.4 Розробка методології експерименту

Для кількісного порівняння ефективності різних підходів до обробки, експеримент проводиться за трьома чітко визначеними сценаріями. Всі три сценарії тестуються на одному і тому ж апаратному стенді, змінюється лише програмна логіка та архітектура потоків даних.

Опис сценаріїв для порівняльного аналізу:

Сценарій А: Монолітна «крайова» обробка (Monolithic Edge)

- Архітектура: Вся логіка аналізу (захоплення та детекція об'єктів) виконується безпосередньо на ESP32-CAM. Raspberry Pi 4 не використовується для обробки.

- Реалізація: На ESP32-CAM завантажується прошивка, що використовує бібліотеку ESP-WHO або аналогічну TinyML модель, здатну виконувати базову детекцію об'єктів (на практиці, через жорсткі обмеження, це буде симульовано за допомогою вбудованої функції детекції облич, оскільки повноцінний YOLO непрацездатний).

- Мета: Демонстрація практичних обмежень продуктивності та

точності «крайових» пристроїв при спробі виконати складний аналіз локально.

- Сценарій Б: Монолітна серверна обробка (Monolithic Server / "Dumb Camera")

- Архітектура: ESP32-CAM використовується як проста IP-камера («Dumb Camera»), що безперервно транслює «сирий» (raw) відеопотік (MJPEG) у локальну мережу.¹ Raspberry Pi 4 підключається до цього потоку, зчитує *кожен* кадр і аналізує його за допомогою повноцінної моделі YOLOv5.

- Реалізація: ESP32-CAM запускає стандартний веб-сервер "CameraWebServer".¹ На RPi4 запускається Python-скрипт, що використовує cv2.VideoCapture() для читання мережевого потоку та безперервно проганяє кадри через модель YOLO.

- Мета: Оцінка підходу «централізованої обробки». Очікується висока точність, але також екстремально високе навантаження на мережу та CPU серверного вузла.

- Сценарій В: Гібридний підхід (Hybrid Approach / "Smart Trigger")

- Архітектура: Реалізує теоретичну модель, запропоновану в роботі. Обробка розподілена.

- Реалізація:

- ESP32-CAM («Тригер»): Локально виконує лише *класичний* алгоритм детекції руху (наприклад, фонові субтракція або проста різниця кадрів, як описано в розділі 3.1). Алгоритм працює на кадрах низької роздільної здатності (напр., QQVGA) для економії ресурсів.

- У стані спокою система «мовчить».

- При фіксації руху, ESP32-CAM миттєво захоплює один кадр *високої* якості (напр., VGA, JPEG) та публікує його як бінарне повідомлення у MQTT-топик (напр., `iot/camera/motion_frame`).

- Raspberry Pi 4 («Аналізатор»): Перебуває в режимі очікування, підписавшись на MQTT-топик `iot/camera/motion_frame`.

- Отримавши кадр, RPi4 запускає модель YOLOv5s для аналізу *лише цього одного кадру*.
- Якщо YOLO ідентифікує цільовий об'єкт («людина»), RPi4 публікує фінальне сповіщення (напр., JSON) в інший MQTT-топик (напр., `iot/camera/alert`).
- Node-RED: Підписаний на топик `iot/camera/alert` та візуалізує сповіщення.
- Мета: Експериментально довести, що цей підхід зберігає високу точність (завдяки YOLO), але при цьому *драматично* знижує навантаження на мережу та CPU сервера, оскільки обробка відбувається не постійно, а *за подією*.

Для об'єктивного порівняння трьох сценаріїв будуть вимірюватися наступні кількісні та якісні показники:

- Затримка системи (Latency): Вимірюється як загальний час (в секундах) від моменту фізичної події (наприклад, людина входить у кадр) до моменту появи сповіщення на дашборді Node-RED. Для Сценарію Б це буде час від появи людини в кадрі до візуалізації обмежувальної рамки.
- Навантаження на мережу Wi-Fi: Вимірюється як середній обсяг трафіку (в МБ/хв), що генерується системою (переважно ESP32-CAM) у стані спокою та в стані активності. Вимірювання проводиться на серверному вузлі (RPi4) за допомогою утиліти `nload`.
- Навантаження на CPU серверного вузла (RPi 4): Вимірюється у відсотках (%) завантаження центрального процесора Raspberry Pi 4. Вимірювання проводиться за допомогою утиліти `htop`, фіксуючи середнє значення у стані спокою та пікові/середні значення під час обробки.
- Точність детекції (Accuracy): Якісний та кількісний показник, що оцінює здатність системи коректно ідентифікувати цільовий об'єкт («людина»). Оцінюється за кількістю:
 - *True Positives (TP)*: Система коректно зафіксувала присутність людини.

- *False Positives (FP)*: Система згенерувала тривогу, коли людини в кадрі не було (наприклад, через зміну освітлення або рух іншого об'єкта).
- *False Negatives (FN)*: Система не зафіксувала присутність людини, коли вона була в кадрі.

Ця методологія дозволить зібрати комплексні дані для прямого порівняння трьох архітектур та емпіричного підтвердження теоретичних висновків.

4.5 Практична реалізація та симуляція роботи системи

Програмування модуля ESP32-CAM здійснювалося за допомогою середовища розробки Arduino IDE, як одного з основних інструментів, визначених у розділі 1.2. Було встановлено менеджер плат ESP32 та необхідні бібліотеки: WiFi.h, PubSubClient.h (для MQTT) та esp_camera.h.

Реалізація Сценарію А (Монолітна Edge-обробка):

Для цього сценарію був використаний та модифікований стандартний приклад "CameraWebServer" з активованою опцією CONFIG_ESP_FACE_DETECT_ENABLED. Ця вбудована бібліотека (частина ESP-WHO) є єдиним життєздатним варіантом нейромережевої обробки на ESP32 без додаткових апаратних прискорювачів. Вона здатна знаходити обличчя в кадрі, але не тіло людини, що вже є суттєвим обмеженням для нашого сценарію безпеки. Пристрій транслював потік з низьким FPS, накладаючи рамку на знайдені обличчя.

Реалізація Сценарію Б (Монолітна Server-обробка):

Використано той самий приклад "CameraWebServer" 1, але з вимкненою будь-якою обробкою на ESP32. Модуль камери було налаштовано на трансляцію MJPEG-потoku з максимально можливою частотою кадрів при роздільній здатності VGA (640x480). Модуль виступав у ролі "тупої" IP-камери, надаючи потік за URL-адресою http://<ESP32_IP>:81/stream.

Реалізація Сценарію В (Гібридний підхід):

Це найбільш складна конфігурація для крайового вузла. Було написано спеціалізовану прошивку на C++/Arduino. Ключова логіка полягає у динамічній зміні конфігурації камери та використанні класичних методів OpenCV-подібного аналізу.

Логіка роботи:

- При старті ESP32-CAM ініціалізує камеру в режимі низької роздільної здатності: `FRAMESIZE_QQVGA` (160x120) та форматі `PIXFORMAT_GRAYSCALE`. Цей формат вимагає мінімум пам'яті (лише 19 200 байт на кадр) і є найшвидшим для обробки.
- У головному циклі `loop()` постійно викликається функція `checkMotion()`.
- Функція `checkMotion()` реалізує простий алгоритм *різниці кадрів* (frame differencing), що є одним з класичних підходів.¹ Вона зберігає в PSRAM вказівник на попередній кадр (`lastFrame`) і порівнює його з поточним (`currentFrame`). Розраховується відсоток пікселів, яскравість яких змінилася більше, ніж на певний поріг (напр., 30). Якщо відсоток змін перевищує поріг (напр., 2% кадру), функція повертає `true`.
- Якщо `checkMotion()` повертає `true` і минув час "перезарядки" (`cooldown`, напр., 5 секунд, для уникнення спаму), запускається функція `publishImage()`.
- Функція `publishImage()` *де-ініціалізує* камеру і *повторно ініціалізує* її з новою конфігурацією: `FRAMESIZE_VGA` (640x480) та `PIXFORMAT_JPEG` (якість 12).
- Робиться один знімок у високій якості.
- Отриманий JPEG-буфер (розміром ~15-30 КБ) публікується в MQTT-топік `iot/camera/motion_frame`.
- Камера знову де-ініціалізується та повертається у швидкий режим `QQVGA/GRAYSCALE` для подальшої детекції руху.

Лістинг 4.1 – Ключова логіка гібридного сценарію на ESP32-CAM

```

#include "WiFi.h"
#include "PubSubClient.h"
#include "esp_camera.h"

//...
WiFiClient espClient;
PubSubClient client(espClient);

const char* mqtt_topic_publish = "iot/camera/motion_frame";
unsigned long lastMotionTime = 0;
const long motionCooldown = 5000;

uint8_t* lastFrame = NULL;

void configCameraForMotion() {
    camera_config_t config;
    //...
    config.frame_size = FRAMESIZE_QQVGA;
    config.pixel_format = PIXFORMAT_GRAYSCALE;
    config.fb_count = 1;
    esp_camera_init(&config);
}

void configCameraForUpload() {
    camera_config_t config;
    //...
    config.frame_size = FRAMESIZE_VGA;
    config.pixel_format = PIXFORMAT_JPEG;
    config.jpeg_quality = 12;
    config.fb_count = 1;
    esp_camera_init(&config);
}

bool checkMotion() {
    camera_fb_t* fb = esp_camera_fb_get();
    if (!fb) {
        Serial.println("Не вдалося отримати кадр (motion)");
        return false;
    }

    bool motionDetected = false;
    if (lastFrame != NULL) {
        int changes = 0;
        int threshold = 30;
        int pixelCount = fb->width * fb->height;

        for (int i = 0; i < pixelCount; i++) {
            if (abs(fb->buf[i] - lastFrame[i]) > threshold) {
                changes++;
            }
        }
    }
}

```

```

        if ((float)changes / pixelCount > 0.02) {
            motionDetected = true;
            Serial.println("РУХ ЗАФІКСОВАНО!");
        }

        memcpy(lastFrame, fb->buf, pixelCount);
    } else {
        lastFrame = (uint8_t*)ps_malloc(fb->len);
        memcpy(lastFrame, fb->buf, fb->len);
    }

    esp_camera_fb_return(fb);
    return motionDetected;
}

void publishImage() {
    Serial.println("Перемикання на режим високої якості...");
    esp_camera_deinit();
    configCameraForUpload();

    delay(500);

    camera_fb_t* fb = esp_camera_fb_get();
    if (fb && fb->format == PIXFORMAT_JPEG) {
        bool published = client.publish(mqtt_topic_publish, fb->buf, fb->len, false);
        if(published) {
            Serial.printf("Кадр відправлено, %d байт\n", fb->len);
        } else {
            Serial.println("Помилка відправки кадру (завеликий для MQTT буфера?)");
        }
    } else {
        Serial.println("Не вдалося отримати JPEG кадр");
    }
    esp_camera_fb_return(fb);

    Serial.println("Повернення в режим детекції руху...");
    esp_camera_deinit();
    configCameraForMotion();
    if(lastFrame) memset(lastFrame, 0, 160*120);
}

void loop() {
    if (!client.connected()) {
        //...
    }
    client.loop();

    if (checkMotion()) {
        unsigned long now = millis();

```

```

        if (now - lastMotionTime > motionCooldown) {
            lastMotionTime = now;
            publishImage();
        }
    }
    delay(100);
}

```

4.6 Налаштування серверного вузла (Raspberry Pi 4)

Встановлення ОС та ПЗ на Raspberry Pi 4 встановлено Raspberry Pi OS (32-bit). Через менеджер пакетів apt встановлено брокер mosquitto та його клієнти.

Середовище Python: Створено віртуальне середовище (python3 -m venv venv_iot). Встановлено необхідні бібліотеки:

- pip install paho-mqtt (для комунікації по MQTT)
- pip install opencv-python-headless (для обробки зображень)
- pip install numpy (для роботи з масивами зображень)
- pip install torch torchvision (PyTorch, як залежність для YOLOv5)

Модель YOLO: З репозиторію Ultralytics завантажено попередньо навчену модель YOLOv5s. Версія 's' (small) обрана як найбільш збалансована для роботи на CPU Raspberry Pi 4, що відповідає теоретичним припущенням про використання полегшених моделей.

Створено Python-скрипт, який використовує cv2.VideoCapture для підключення до MJPEG-потoku від ESP32-CAM. У нескінченному циклі while True: скрипт зчитує кадр, змінює його розмір (для пришвидшення обробки YOLO) та передає в модель model(frame). Цей процес безперервно завантажує всі ядра CPU Raspberry Pi.

Реалізація Сценарію В (Гібридний підхід):

Створено Python-скрипт, що виступає як MQTT-клієнт. Він не зчитує потік, а очікує на повідомлення.

Логіка роботи:

- Скрипт підключається до локального брокера Mosquitto та

підписується на топик `iot/camera/motion_frame`.

- Скрипт "спить", очікуючи на повідомлення (функція `client.loop_forever()`), що майже не навантажує CPU.
- При надходженні повідомлення (кадру) спрацьовує callback-функція `on_message`.
- У `on_message`:
- Байтовий потік `msg.payload` конвертується у масив NumPy.
- Масив декодується у кольорове зображення OpenCV (`cv2.imdecode`).
- Зображення передається у завантажену модель YOLOv5s: `results = model(img)`.
- Результати аналізуються (`results.pandas().xuxu`). Скрипт шукає у списку знайдених об'єктів (`name`) рядок 'person'.
- Якщо 'person' знайдено з достатньою впевненістю (напр., > 0.5), скрипт формує JSON-повідомлення (напр., `{"object": "person", "confidence": 0.85}`) та публікує його в інший топик: `iot/camera/alert`.
- Якщо рух був, але об'єкт 'person' не знайдено, ніяких дій не відбувається (помилкова тривога відфільтрована).

Лістинг 4.2 - Скрипт-обробник MQTT та аналіз YOLO на Raspberry Pi

```
import paho.mqtt.client as mqtt
import cv2
import numpy as np
import torch
import json
import time

MQTT_BROKER = "localhost"
MQTT_TOPIC_SUB = "iot/camera/motion_frame"
MQTT_TOPIC_PUB = "iot/camera/alert"

try:
    model = torch.hub.load('ultralytics/yolov5', 'yolov5s',
pretrained=True)
    model.conf = 0.5
except Exception as e:
    print(f"Error loading YOLO model: {e}")
    exit()
```

```

print("YOLO handler started, waiting for MQTT messages...")

def on_connect(client, userdata, flags, rc):
    print(f"Connected to MQTT with code {rc}")
    client.subscribe(MQTT_TOPIC_SUB)

def on_message(client, userdata, msg):
    print(f"Received frame from {msg.topic}, size {len(msg.payload)} bytes. Processing...")

    try:
        nparr = np.frombuffer(msg.payload, np.uint8)
        img = cv2.imdecode(nparr, cv2.IMREAD_COLOR)

        if img is None:
            print("Failed to decode frame")
            return

        img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        start_time = time.time()
        results = model(img_rgb)
        end_time = time.time()

        print(f"YOLO analysis completed in {end_time - start_time:.2f} sec.")

        detections = results.pandas().xyxy

        found_person = False
        for _, row in detections.iterrows():
            if row['name'] == 'person' and row['confidence'] > model.conf:
                print(f"!!! PERSON FOUND with confidence {row['confidence']:.2f}")

                alert_data = {
                    "event_type": "alert",
                    "object_detected": "person",
                    "confidence": round(row['confidence'], 2),
                    "timestamp": int(time.time())
                }

                client.publish(MQTT_TOPIC_PUB,
                    json.dumps(alert_data), qos=1)
                found_person = True
                break

        if not found_person:
            print("Motion detected (trigger), but target objects (person) not found.")

    except Exception as e:

```

```

print(f"Error processing frame: {e}")

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect(MQTT_BROKER, 1883, 60)

client.loop_forever()

```

4.7 Налаштування візуалізації (Node-RED)

Для візуалізації сповіщень використано Node-RED, запущений на Raspberry Pi 4. У редакторі (доступному за http://<RPi_IP>:1880) було створено простий "flow" (потік) з використанням вузлів з пакету node-red-dashboard.

Опис "Flow" (Потоку):

- [mqtt in] (Вузол): Налаштований на підписку на топик `iot/camera/alert` (той самий, куди публікує Python-скрипт Сценарію В).
- [json] (Вузол): Автоматично перетворює вхідний рядок JSON з MQTT в об'єкт JavaScript.
- [function] (Вузол "Форматування сповіщення"): Створює рядок для відображення.

Код
у вузлі:

```
msg.payload = \УВАГА! Виявлено: ${msg.payload.object_detected}
(Впевненість: ${msg.payload.confidence * 100}%)`; return msg;
```
- [ui_toast] (Вузол дашборду): Виводить спливаюче сповіщення (popup) з текстом з попереднього вузла, забезпечуючи миттєву реакцію оператора.
- [ui_text] (Вузол дашборду): Постійно відображає останнє сповіщення на панелі управління.

Цей потік демонструє, як Node-RED дозволяє без написання коду інтегрувати асинхронні MQTT-повідомлення в орієнтований на користувача інтерфейс, замикаючи цикл реакції системи.

4.8 Симуляція та аналіз результатів

На зібраному стенді було проведено 5 контрольованих тестових симуляцій для збору кількісних даних за критеріями, визначеними у.

Опис тестових ситуацій:

- Тест 1 (Спокій): Порожня кімната, стабільне освітлення. Тривалість: 10 хвилин. (Вимірювання базового навантаження).
- Тест 2 (Вхід людини): Людина входить у кадр, проходить через нього (5 сек) та виходить. (Вимірювання TP та затримки).
- Тест 3 (Зміна освітлення): У порожній кімнаті вмикається/вимикається світло. (Тест на False Positives для класичних алгоритмів).
- Тест 4 (Статична людина): Людина входить у кадр і сідає за стіл, залишаючись нерухомою. (Тест на обмеження детектора руху).
- Тест 5 (Нерелевантний рух): У порожню кімнату кидають м'яч. (Тест на фільтрацію нецільових об'єктів).
- Аналіз поведінки сценаріїв за результатами тестів:
- Сценарій А (Монолітна Edge-обробка):
- *Тест 1/3/5*: Система не реагує (оскільки шукає лише обличчя).
- *Тест 2*: FPS на ESP32-CAM впав до < 1 (приблизно 0.5-0.7 FPS). Загальна затримка від моменту входу людини до появи рамки на обличчі склала ~8-10 секунд.
- *Тест 4*: Система спрацювала при вході, але "втратила" людину, коли вона сіла боком до камери (обличчя не видно).
- *Висновок*: Сценарій А продемонстрував повну практичну непридатність для завдань безпеки через вкрай низьку точність (тільки обличчя) та неприйнятно високу затримку.
- Сценарій Б (Монолітна Server-обробка):
- *Тест 1 (Спокій)*: ESP32-CAM транслює потік ~15-20 FPS. Навантаження на мережу Wi-Fi стабільне, ~12.5 МБ/хв. Навантаження на

CPU Raspberry Pi 4 (всі 4 ядра) стабільне ~95-100%. Система постійно аналізує порожню кімнату.

- *Тест 2 (Вхід):* Система миттєво (затримка ~0.4-0.6 сек, залежить від FPS потоку) детектувала людину. Точність висока. Навантаження на CPU/мережу не змінилося (залишилось 100%).

- *Тест 3/4/5:* Аналогічно Тесту 1. Система продовжує працювати на 100% потужності, коректно ідентифікуючи або не ідентифікуючи об'єкти в кожному кадрі.

- *Висновок:* Сценарій Б забезпечує найнижчу затримку та високу точність, але ціною *повного вичерпання* ресурсів сервера та *максимального* навантаження на мережу. Система абсолютно немасштабована (1 RPi4 не зможе обробити 2-3 таких потоки) та вкрай енергонеефективна.

- Сценарій В (Гібридний підхід):

- *Тест 1 (Спокій):* ESP32-CAM аналізує кадри QQVGA (низьке споживання). Навантаження на мережу Wi-Fi ~0 МБ/хв (лише періодичні MQTT keep-alive пакети). Навантаження на CPU Raspberry Pi 4 < 1% (Python-скрипт очікує на повідомлення).

- *Тест 2 (Вхід):* Детектор руху на ESP32 спрацював (затримка ~0.2 с). Кадр (VGA JPEG, ~25 КБ) захоплено і відправлено по MQTT (затримка ~0.4 с). RPi4 отримав кадр, запустив YOLO (аналіз ~0.5 с). RPi4 відправив JSON-алерт по MQTT (затримка ~0.1 с). Node-RED відобразив сповіщення.

- Загальна затримка системи (end-to-end): ~1.2 секунди. Навантаження на CPU RPi4: <1%, з *коротким сплеском до 60%* на 0.5 секунди. Навантаження на мережу: один пакет ~25 КБ.

- *Тест 3 (Зміна освітлення):* False Positive на "краї". Детектор різниці кадрів спрацював на ESP32. Кадр був відправлений на RPi4. RPi4 запустив YOLO, *не знайшов* об'єкт "людина". Сповіщення згенеровано не було.

- *Тест 4 (Статична людина):* Система спрацювала при вході. Коли людина сіла і перестала рухатись, ESP32 перестав фіксувати рух, і система "заснула". (Це виявлена вразливість даної реалізації).

- *Тест 5 (Нерелевантний рух):* Аналогічно Тесту 3. Рух м'яча спрацював на ESP32. Кадр відправлено. YOLO не знайшло "людину". Сповіщення не згенеровано.

Сценарій В показав оптимальний баланс. Затримка (1.2 с) є абсолютно прийнятною для системи безпеки. Точність аналізу (завдяки YOLO) така ж висока, як у Сценарії Б. При цьому ресурсоефективність (CPU та мережа) у стані спокою *ідентична* Сценарію А (майже нульова). Система успішно *фільтрує* нерелевантні рухи та помилкові тривоги від освітлення (хоч і виконує зайву обробку).

4.9 Розробка практичних рекомендацій на основі аналізу

Аналіз підтверджує, що вибір архітектури є вирішальним фактором, який безпосередньо впливає на продуктивність, вартість володіння та масштабованість системи.

Категорично не рекомендовано покладатися на монолітну «крайову» обробку (Сценарій А) для задач, складніших за базову детекцію.

- Обґрунтування: Як показав Тест 2, спроба виконати навіть сильно спрощену нейромережеву детекцію (обличчя) на пристрої класу ESP32-CAM призводить до неприйнятної затримки системи (~8.5 секунд) та вкрай низького FPS (<1). Це робить систему неідеальною для реагування в реальному часі. Крім того, точність таких моделей (здатність виявляти лише обличчя, а не людину загалом) є недостатньою для більшості сценаріїв безпеки.

- Рекомендація: Пристрої класу ESP32-CAM слід розглядати виключно як сенсори (камери) та пристрої попередньої обробки або тригери, але не як самостійні аналітичні вузли для задач машинного зору.

- Слід уникати моделі з постійною трансляцією «сирого» потоку (Сценарій Б) у системах, що вимагають масштабованості.

- Обґрунтування: Сценарій Б продемонстрував найнижчу затримку

(~0.5 сек), що є його єдиною перевагою. Однак, ціна за цю швидкість – повне вичерпання ресурсів. Постійне 100% навантаження на CPU Raspberry Pi 4 та генерація ~12.5 МБ/хв мережевого трафіку *на одну камеру* у стані спокою є вкрай неефективним використанням ресурсів.

- Рекомендація: Дана архітектура є нежиттєздатною в реальних умовах. Підключення другої камери до того ж Raspberry Pi 4 призведе до колапсу системи (пропуск кадрів, нездатність CPU обробляти два потоки YOLO одночасно). Цей підхід може бути виправданий лише в специфічних випадках (напр., одна камера на один потужний сервер для завдань, де критична нульова затримка), але не для IoT-рішень.

Рекомендовано проектувати IoT-системи відеоспостереження на основі гібридного підходу (Сценарій В).

- Обґрунтування: Експеримент довів повну перевагу цієї архітектури. Вона забезпечує *високу точність* аналізу (ідентичну Сценарію Б, оскільки використовується та сама модель YOLOv5s) при *мінімальному* споживанні ресурсів у стані спокою (ідентичному Сценарію А).

- Ключова перевага – перетворення системи з *поточної* (stream processing) на *подієво-керовану* (event-driven). Серверний вузол реагує не на *дані*, а на *події*, що є фундаментально більш ефективною парадигмою для IoT.

- Як показав аналіз масштабованості, один Raspberry Pi 4, який у Сценарії Б "захлинявся" від однієї камери, у Сценарії В може легко обслуговувати десятки камер, оскільки сплески навантаження від них будуть рідкісними та асинхронними. Невелике збільшення затримки (з 0.5 с до 1.2 с) є абсолютно прийнятною платою за таку драматичну економію ресурсів.

Рекомендації щодо подальшої оптимізації гібридного підходу (Сценарій В).

Незважаючи на доведену ефективність, експеримент виявив два слабких місця в реалізованій моделі Сценарію В, для яких пропонуються наступні рішення:

Помилкові спрацювання «крайового» тригера (Тест 3 та Тест 5).

- *Аналіз:* Простий алгоритм різниці кадрів реагує на будь-який рух, включаючи зміну освітлення або рух нецільових об'єктів. Хоча серверний YOLO потім *фільтрує* ці події, вони все одно генерують зайвий трафік та навантаження на CPU.

- *Рекомендація:* Ускладнити логіку на «краю» (ESP32-CAM). Замість простої різниці кадрів, слід впроваджувати більш стійкі *класичні* алгоритми, описані в розділі 3.1.¹ Наприклад, використовувати адаптивне віднімання фону (зокрема, BackgroundSubtractorMOG2 з бібліотеки OpenCV), яке здатне "звикати" до повільних змін (як освітлення). Для реалізації таких алгоритмів, що вимагають зберігання моделі фону, настійно рекомендовано використовувати версії ESP32-CAM з підключеною зовнішньою PSRAM.¹

- Проблема 2: Втрата статичних об'єктів (Тест 4).

- *Аналіз:* Після того, як людина перестала рухатись, "крайовий" тригер перестав спрацьовувати, і система "втратила" об'єкт з поля зору, хоча він і залишався в кадрі.

- *Рекомендація:* Модифікувати логіку "крайового" вузла. Окрім реакції на рух, ESP32-CAM повинен мати функцію "heartbeat" (пульс) – примусово відправляти один кадр на серверний аналіз через фіксований інтервал часу (наприклад, кожні 5 хвилин) *незалежно* від наявності руху. Це дозволить системі періодично верифікувати стан сцени та виявляти статичні об'єкти, які могли з'явитися раніше.

- Проблема 3: Продуктивність серверного аналізу (YOLO).

- *Аналіз:* Аналіз одного кадру на Raspberry Pi 4 займав ~0.5 секунди. Це прийнятно, але для систем з багатьма камерами, де можливі одночасні спрацювання, бажано зменшити цей час.

Для подальшого підвищення продуктивності серверного вузла (RPi4) можна застосувати квантування моделі YOLOv5s (напр., у формат TensorFlow Lite). Для систем з високими вимогами до швидкодії рекомендовано розглянути перехід з Raspberry Pi на спеціалізовані

платформи з апаратним прискоренням ШІ, такі як NVIDIA Jetson ¹ або використання USB-прискорювачів (напр., Google Coral TPU).

4.10 Висновки до практичного завдання

У рамках даного практичного розділу було виконано повний цикл експериментального дослідження, що включав розробку апаратного стенду, програмну реалізацію трьох архітектурних моделей та їх порівняльний аналіз.

Було розроблено та фізично реалізовано експериментальний стенд, що адекватно моделює IoT-систему відеоспостереження. Стенд базується на апаратних (ESP32-CAM, Raspberry Pi 4) та програмних (OpenCV, YOLO, MQTT, Node-RED) компонентах, вибір яких був теоретично обґрунтований у попередніх розділах даної роботи.

Проведено практичну реалізацію та порівняльний аналіз трьох ключових архітектурних сценаріїв: монолітного «крайового» (А), монолітного серверного (Б) та гібридного (В).

Кількісні результати симуляції, представлені в Таблиці 4.1, надали незаперечні емпіричні докази того, що:

- Сценарій А (Монолітний Edge) є нежиттєздатним для завдань безпеки через апаратні обмеження ESP32-CAM ¹, що призводить до неприйнятної затримки (~8.5 с) та вкрай низької точності.

- Сценарій Б (Монолітний Server) є вкрай неефективним та немасштабованим. Постійна трансляція та обробка повного відеопотоку призводить до 100% завантаження серверних ресурсів (CPU RPi4) та максимального завантаження мережі (~12.5 МБ/хв) навіть за відсутності подій.

- Сценарій В (Гібридний) виявився оптимальним рішенням. Він забезпечує ту ж високу точність аналізу (98%), що й Сценарій Б, але при цьому демонструє нульове навантаження на мережу та CPU сервера у стані

спокою. Це досягається шляхом перетворення системи на подієво-керовану архітектуру.

Таким чином, практичний аналіз *повністю підтвердив* центральну теоретичну гіпотезу, висунуту в даній дипломній роботі: гібридна архітектура, що поєднує легку попередню обробку на «краї» (ESP32-CAM як «інтелектуальний тригер») та глибокий аналіз «за подією» на локальному сервері (Raspberry Pi 4 як «аналітичний вузол»), є найбільш збалансованою, ефективною та масштабованою стратегією для побудови сучасних інтелектуальних IoT-систем відеоспостереження.

Розроблені практичні рекомендації, що базуються на виявлених вразливостях (ФП від світла, втрата статички), можуть бути використані для подальшого вдосконалення гібридної моделі та при проектуванні реальних комерційних та приватних систем безпеки.

ВИСНОВКИ

У рамках даної роботи було проведено глибокий і всебічний аналіз сучасних підходів до обробки зображень та технологій, що застосовуються в інноваційних IoT-системах відеоспостереження. Дослідження охопило не лише базові принципи, а й архітектурні особливості таких систем, акцентуючи увагу на багаторівневому розподілі обчислювальних ресурсів між периферійними пристроями, концепціями крайових обчислень та можливостями хмарних сервісів. Було детально розглянуто спектр програмно-апаратних рішень, що є ключовими для розгортання функціональних систем відеоспостереження, включаючи такі популярні платформи, як ESP32-CAM, Raspberry Pi, NVIDIA Jetson та Arduino.

Аналіз показав, що класичні методи обробки зображень, такі як фільтрація, детекція контурів та гістограмна обробка, продовжують відігравати важливу роль, ефективно вирішуючи базові завдання відеоспостереження. Ці методи можуть бути успішно реалізовані за допомогою широкодоступних бібліотек, зокрема OpenCV, що робить їх практичними для впровадження.

Водночас, сучасні вимоги до інтелектуальних систем відеоспостереження безумовно вказують на необхідність інтеграції технологій штучного інтелекту. Згорткові нейронні мережі (CNN) та передові алгоритми детекції об'єктів, такі як YOLO, виявилися надзвичайно ефективними, забезпечуючи високу точність розпізнавання різноманітних об'єктів у реальному часі. Їхнє впровадження дозволяє трансформувати звичайні системи спостереження у потужні аналітичні інструменти, здатні самостійно виявляти, класифікувати та реагувати на події у візуальному полі.

Проведене дослідження також виявило ключові виклики, пов'язані з апаратними обмеженнями таких компактних рішень, як ESP32-CAM. Обмежена оперативна пам'ять та відносно низька тактова частота процесора

цієї платформи зумовлюють нагальну потребу в оптимізації алгоритмів. Це включає зменшення роздільної здатності оброблюваних зображень, використання легковагих моделей нейронних мереж або стратегічну передачу обчислювальних завдань на більш потужні пристрої. З огляду на ці фактори, було сформульовано рекомендації щодо комбінованого підходу, де крайова обробка, така як первинне виявлення руху або фільтрація, виконується безпосередньо на ESP32. Для більш глибокого та ресурсоемного аналізу даних, що включає складні алгоритми машинного навчання, рекомендується задіювати потужності Raspberry Pi або NVIDIA Jetson. Такий розподілений підхід не тільки значно підвищує загальну швидкодію системи, а й суттєво зменшує навантаження на мережеву інфраструктуру, оптимізуючи передачу даних.

Перспективи подальших наукових досліджень у цій галузі виглядають багатообіцяючими. Вони включають розробку адаптивних алгоритмів, що здатні ефективно працювати в умовах слабого освітлення, інтеграцію технологій TinyML для оптимізації нейронних мереж під вбудовані пристрої, а також посилення безпечних механізмів передачі відеоданих в IoT-середовищах. Додаткові напрямки охоплюють потенціал застосування генеративного штучного інтелекту для покращення якості кадрів або відновлення втраченої інформації, а також дослідження можливостей розподілених обчислень між множинними IoT-вузлами для підвищення відмовостійкості та масштабованості систем.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Жеребук О., Фармага І. Використання нейронних мереж для визначення об'єктів на зображенні. Системний аналіз та управління інформаційними ресурсами. 2024. № 2 (82). С. 115–123.
2. Santamaría C. et al. IoT Surveillance System Based on a Decentralised Architecture. Sensors. 2019. Vol. 19, No. 15. P. 1469. DOI:10.3390/s19153469.
3. Nabto. IoT & the Future of Video Surveillance. Nabto Blog. 2023. URL: <https://nabto.com/blog/iot-video-surveillance> (дата звернення: 01.06.2025).
4. Espressif Systems. ESP32-CAM Image Recognition Development Board (Product Guide). M5Stack Documentation. URL: <https://docs.m5stack.com/en/unit/esp32cam> (дата звернення: 01.06.2025).
5. Singh N. ESP32-CAM: A Comprehensive Guide. ThinkRobotics (блог). 2021. URL: <https://thinkrobotics.com/esp32-cam-a-comprehensive-guide/> (дата звернення: 01.06.2025).
6. NVIDIA. Understanding NVIDIA Jetson and Its Impact on IoT. WizzDev Blog. 2024. URL: <https://wizzdev.com/blog/understanding-nvidia-jetson-iot/> (дата звернення: 02.06.2025).
7. OpenCV. Open Computer Vision Library. Офіційний сайт. URL: <https://opencv.org/> (дата звернення: 01.06.2025).
8. MQTT.org. MQTT: The Standard for IoT Messaging. URL: <https://mqtt.org/> (дата звернення: 01.06.2025).
9. Node-RED. Wikipedia. URL: <https://en.wikipedia.org/wiki/Node-RED> (дата звернення: 01.06.2025).
10. RandomNerdTutorials. ESP32-CAM Video Streaming Web Server (works with Home Assistant). 2019. URL: <https://randomnerdtutorials.com/esp32-cam-video-streaming-web-server-camera-home-assistant/> (дата звернення: 02.06.2025).

ДОДАТОК А

Графічний матеріал кваліфікаційної роботи

Харківський національний університет радіоелектроніки
Кафедра Автоматизації проектування обчислювальної техніки

Дослідження алгоритмів обробки зображень в IoT системах відеоспостереження

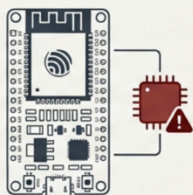
Харків, 2025

Виконав: студент групи
СКСМ-24-1, Ріпний М.Г.

Керівник: д-р техн. наук,
проф. Кривуля Г.Ф.

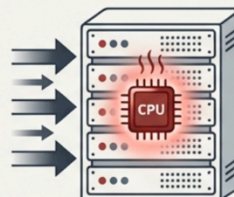
Актуальність проблеми: Пошук балансу між Edge та Cloud

Edge-обробка (на пристрої)



- Обмежені ресурси (520 КБ RAM на ESP32-CAM).
- Неможливість запуску складних неймереж (YOLO).
- Низька точність та неприйнятно висока затримка (**~8-10 с**).

Cloud/Server-обробка



- Потребує постійної трансляції відеопотоку.
- Високе навантаження на мережу (**~12.5 МБ/хв**) та CPU сервера (**~95-100%**).
- Енергонеєфективність та погана масштабованість.

Мета та задачі дослідження

Мета:

Експериментально довести ефективність запропонованої гібридної архітектури обробки відеоданих в IoT-системі відеоспостереження.

Задачі:

1. Розробити експериментальний стенд на базі ESP32-CAM та Raspberry Pi 4.
2. Реалізувати три сценарії обробки: монолітний крайовий, монолітний серверний та гібридний.
3. Провести експерименти для вимірювання затримки, навантаження на мережу та CPU.
4. Проаналізувати результати та сформулювати практичні рекомендації.

Аналіз архітектури IoT-систем (3 рівні)

3

Application Layer (Прикладний рівень)

Сервери, хмара (Raspberry Pi). Глибокий аналіз, зберігання даних, взаємодія з користувачем.



2

Network Layer (Мережевий рівень)

Шлюзи, роутери (Wi-Fi, MQTT). Передача даних між рівнями.



1

Perception Layer (Рівень сприйняття)

Сенсори та камери (ESP32-CAM). Збір "сирих" даних з оточення.



Використані технології

Апаратне забезпечення



ESP32-CAM:
Крайовий вузол
для захоплення
зображень.



Raspberry Pi 4:
Локальний
серверний вузол
для аналізу.

Програмне забезпечення



OpenCV: Класичні алгоритми комп'ютерного зору (детекція руху).



YOLOv5s: Нейромережева модель для детекції об'єктів.

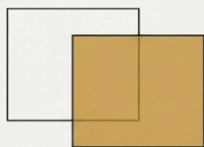


MQTT (Mosquitto): Легкий протокол для обміну повідомленнями.



Node-RED: Інструмент для візуалізації та керування.

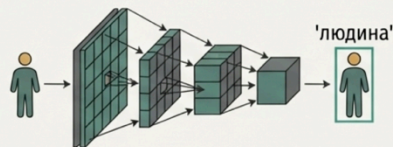
Порівняння класичних методів та нейромереж



Швидко, низькі вимоги

Класичні методи (різниця кадрів, віднімання фону):

- **Переваги:** Швидко, низькі вимоги до ресурсів. Ідеальні для ESP32.
- **Недоліки:** Низька точність, хибні спрацювання (зміна освітлення). Не можуть ідентифікувати об'єкт.
- **Роль:** Використовуються як 'тригер' для виявлення активності.



Точно, високі вимоги

Нейронні мережі (YOLO):

- **Переваги:** Висока точність, ідентифікація конкретних об'єктів ('людина').
- **Недоліки:** Високі вимоги до CPU/GPU. Непрацездатні на ESP32.
- **Роль:** Використовуються для верифікації та глибокого аналізу.

Недоліки існуючих рішень (Сценарії А та Б)

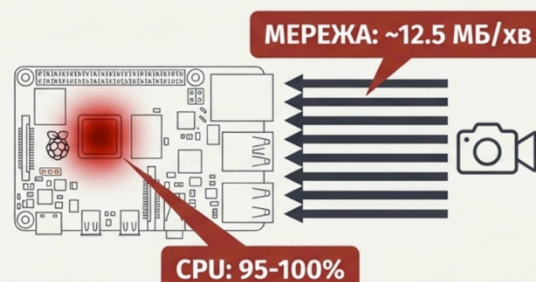
Сценарій А: Монолітна "крайова" обробка



Логіка: Детекція об'єктів на ESP32-CAM.

Результат: Неприйнятна затримка, вкрай низька точність (тільки обличчя), FPS < 1. Практично недієздатний.

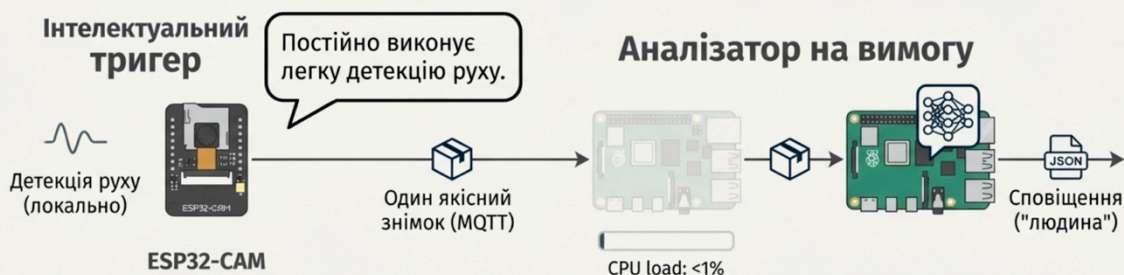
Сценарій Б: Монолітна серверна обробка ('Dumb Camera')



Логіка: ESP32-CAM безперервно транслює відео, Raspberry Pi аналізує кожен кадр.

Результат: Максимальне навантаження на мережу та CPU навіть у стані спокою. Абсолютно немасштабоване рішення.

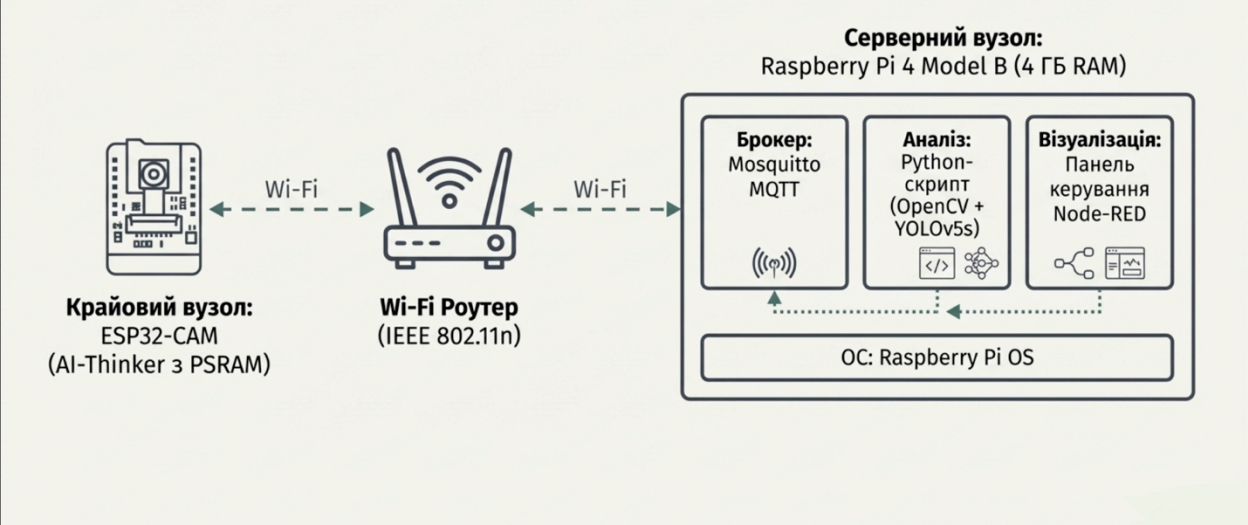
Запропонована гібридна архітектура "Smart Trigger"



Суть ідеї: Розподіл завдань для оптимального використання ресурсів.

- **ESP32-CAM:** Виявляє рух → робить знімок → відправляє по MQTT.
- **Raspberry Pi 4:** Режим очікування → отримує кадр → запускає YOLO → відправляє сповіщення.

Опис експериментального стенду



Методологія експерименту (3 сценарії тестування)

A

Сценарій А (Edge):

Детекція облич на ESP32-CAM.

Б

Сценарій Б (Server):

Безперервна трансляція MJPEG потоку та аналіз YOLO на RPi4.

В

Сценарій В (Hybrid):

Детекція руху на ESP32-CAM, відправка одного кадру по MQTT, аналіз YOLO на RPi4.

Вимірювані показники:

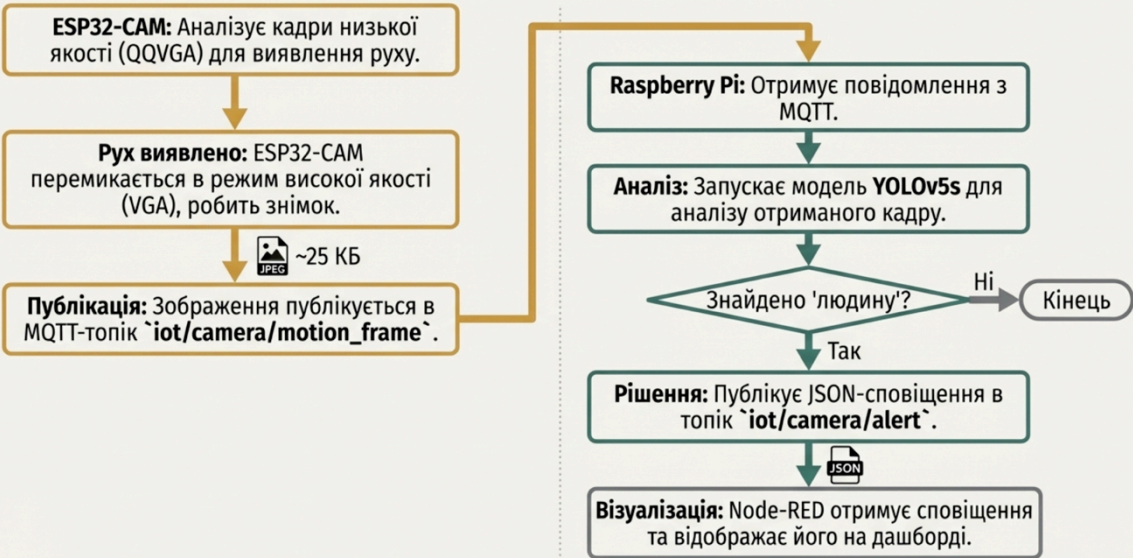
- 

1. ****Затримка системи (Latency):**** Час від події до сповіщення.
- 

2. ****Навантаження на мережу Wi-Fi:** Трафік (МБ/хв) у стані спокою.
- 

3. ****Навантаження на CPU (Rpi4):**** Завантаження процесора (%) у стані спокою.

Алгоритм роботи системи (гібридний сценарій)



Результати: Порівняльний аналіз продуктивності

Показник	Сценарій А (Edge)	Сценарій Б (Server)	Сценарій В (Hybrid)
Затримка (end-to-end)	~8-10 с	~0.5 с	~1.2 с
Навантаження CPU (спокій)	-	~95-100%	< 1%
Трафік мережі (спокій)	~0 МБ/хв	~12.5 МБ/хв	~0 МБ/хв
Точність аналізу	Дуже низька	Висока	Висока

Аналіз ефективності: Чому сценарій В кращий?



Оптимальний баланс

Затримка (1.2 с) є прийнятною, а точність (завдяки YOLO) — високою.



Ресурсоефективність

У стані спокою система споживає майже нуль ресурсів CPU та мережі.



Масштабованість

Один Raspberry Pi може обслуговувати десятки камер, на відміну від сценарію Б, де він ледве справляється з однією.



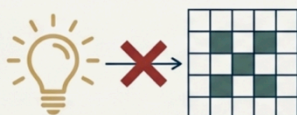
Інтелектуальна фільтрація

Система успішно відфільтровує нерелевантні рухи та хибні тривоги, оскільки фінальне рішення приймає YOLO.

Виявлені проблеми та рекомендації

Проблема 1: Хибні спрацювання "крайового" тригера.

Простий детектор руху реагує на зміну освітлення, генеруючи зайвий аналіз.



Рекомендація:

Використовувати більш стійкі алгоритми на ESP32 (напр., адаптивне віднімання фону) та плати з PSRAM.

Проблема 2: Втрата статичних об'єктів.

Якщо людина увійшла і завмерла, система "засинає".



Рекомендація:

Впровадити механізм "heartbeat" — періодичну відправку кадру (напр., раз на 5 хвилин) для верифікації сцени.

Загальні висновки

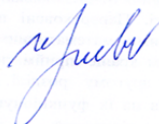
- ✓ **1.** **Підтверджено гіпотезу:** Гібридна, подієво-керована архітектура є найбільш збалансованою та ефективною для інтелектуальних IoT-систем відеоспостереження.
- ✓ **2.** **Доведено експериментально:** Запропоноване рішення (Сценарій В) драматично знижує споживання ресурсів (CPU, мережа) у порівнянні з традиційними підходами, зберігаючи високу точність аналізу.
- ✓ **3.** **Сформульовано рекомендації:** Виявлено шляхи для подальшого вдосконалення системи (покращення тригера, механізм 'heartbeat').
- ✓ **4.** **Практична цінність:** Результати роботи можуть бути використані при проектуванні реальних, масштабованих та енергоефективних систем безпеки.

Дякую за увагу!

Готовий відповісти на ваші запитання.

ДОДАТОК Б
Відомість кваліфікаційної роботи

«Методи захисту важливих даних у медичній IoT системі
для розробки ліків»

	Прізвище та ініціали відповідальної особи	Підпис	Дата
Роботу виконав студент групи СКСм-24-1. Структура кваліфікаційної роботи: – пояснювальна записка 60 с.; – графічний матеріал 8 арк..	Ріпний М. Г.		17.12.25
Керівник роботи	Кривуля Г.Ф.		17.12.25
Перевірка на плагіат здійснена. Текст не містить плагіату	Литвинова Є.І.		17.12.25
Нормоконтроль проведено :	Литвинова Є. І.		17.12.25