Міністерство освіти і науки України
Луцький національний технічний університет
(повне найменування закладу вищої освіти)
Факультет комп'ютерних та інформаційних технологій
(повне найменування факультету)
Кафедра комп'ютерної інженерії та охоронних систем
(повне найменування кафедри)

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

ДОСЛІДЖЕННЯ ТА АНАЛІЗ ЕФЕКТИВНОСТІ PYTHON-ОРІЄНТОВАНИХ СЕРВІСІВ НА RASPBERRY PI В СФЕРІ IOT

RESEARCH AND ANALYSIS OF THE EFFECTIVENESS OF PYTHON-ORIENTED SERVICES ON RASPBERRY PI IN THE FIELD OF IOT

Спеціальність 123 Комп'ютерна інженерія
(шифр і назва спеціальності)

освітня програма Комп'ютерна інженерія
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІ-42
Верешко Вадим Вікторович

(підпис)

Керівник:
асистент
Кулакевич Олег Русланович

(підпис)

Кваліфікаційну роботу
допущено до захисту
«_____» _____ 2026 р.
Гарант освітньої програми:
к.т.н., доцент
Лавренчук Світлана Василівна

(підпис)

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерної інженерії та безпеки*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *123 Комп'ютерна інженерія*
Освітня програма: *«Комп'ютерна інженерія»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ доц. Т. Терлецький
«_____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Верешку Вадиму Вікторовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Дослідження та аналіз ефективності Python-орієнтованих сервісів на Raspberry Pi в сфері IoT*
Керівник роботи *асистент, Кулакевич Олег Русланович*
затверджені наказом закладу вищої освіти від «20» грудня 2025 року № 536/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи 28.05.2026 р.
3. Вихідні дані до роботи *Джерелом розробки є науково-технічна література та публікації в періодичних виданнях з даного питання, опубліковані зарубіжні та вітчизняні роботи в даній області, різні інтернет-ресурси технічного спрямування*
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Вступ
Аналіз предметної області та наявних рішень
Проєктування та обґрунтування засобів реалізації
Реалізація та дослідження Python-орієнтованих сервісів
Висновки
5. Перелік графічного (ілюстративного) матеріалу:
набір слайдів презентації кваліфікаційної роботи

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області та наявних рішень	*Кулакевич О. Р., асистент*		
Проєктування та обґрунтування засобів реалізації	*Кулакевич О. Р., асистент*		
Реалізація та дослідження Python-орієнтованих сервісів	*Кулакевич О. Р., асистент*		
Нормоконтроль	*Багнюк Н. В., доцент*		
Гарант ОП	*Лавренчук С. В., доцент*		
Показник запозичень тексту	***___%***		
Академічна доброчесність	*Міскевич О. І., ст. викладач*		

7. Дата видачі завдання 23.12.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	*Огляд літератури із досліджуваної проблеми, аналіз предметної області та наявних рішень*	до 10.02.2026 р.	
2.	*Аналіз предметної області та наявних рішень*	до 02.03.2026 р.	
3.	*Проєктування та обґрунтування засобів реалізації*	до 02.04.2026 р.	
4.	*Реалізація та дослідження Python-орієнтованих сервісів*	до 10.04.2026 р.	
5.	*Представлення остаточного варіанту кваліфікаційної роботи керівникові*	до 01.05.2026 р.	
6.	*Нормоконтроль*	до 23.05.2026 р.	
7.	*Інструментальна перевірка на академічний плагіат*	до 26.05.2026 р.	
8.	*Здача кваліфікаційної роботи та всіх супровідних документів на кафедру*	до 28.05.2026 р.	

Здобувач вищої освіти _____ Вадим ВЕРЕШКО
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ Олег КУЛАКЕВИЧ
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Верешко В. В. Дослідження та аналіз ефективності Python-орієнтованих сервісів на Raspberry Pi в сфері IoT. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерна інженерія» спеціальності 123 Комп'ютерна інженерія. Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, переліку використаних джерел та додатків.

У першому розділі проведено аналіз сучасних IoT-систем, особливостей використання Python у задачах обробки даних, а також досліджено технології HTTP та MQTT для організації обміну повідомленнями між компонентами системи. Розглянуто архітектури IoT-систем та сучасні засоби контейнеризації програмного забезпечення.

У другому розділі виконано проєктування структури програмної системи, обґрунтовано вибір FastAPI, MQTT-брокера Mosquitto та Docker Compose. Розроблено архітектуру Python-сервісу, реалізовано механізми HTTP-обробки запитів, MQTT-взаємодії та модуль збору статистики.

Третій розділ присвячено практичній реалізації та експериментальному дослідженню системи. Реалізовано генератор навантаження для формування послідовних і паралельних HTTP-запитів, проведено тестування продуктивності Python-сервісу за різних сценаріїв навантаження, а також досліджено вплив MQTT-обміну на використання процесорних ресурсів і час відповіді системи.

Ключові слова: IoT, Python, FastAPI, MQTT, HTTP, Docker, Docker Compose, Mosquitto, навантажувальне тестування, контейнеризація.

ANNOTATION

Vereshko V. Research and analysis of the effectiveness of Python-oriented services on Raspberry Pi in the field of IoT. Manuscript.

Bachelor's qualification work of the Educational Program "Computer Engineering", specialty 123 Computer Engineering. Lutsk National Technical University. Lutsk, 2026.

The qualification work consists of an introduction, three chapters, conclusions, references, and appendices.

The first chapter analyzes modern IoT systems, the features of using Python for data processing tasks, and the HTTP and MQTT technologies for communication between system components. IoT architectures and modern software containerization tools are also considered.

The second chapter presents the design of the software system structure and substantiates the choice of FastAPI, the Mosquitto MQTT broker, and Docker Compose. The architecture of the Python service was developed, including HTTP request processing, MQTT communication, and a statistics collection module.

The third chapter is devoted to the practical implementation and experimental study of the system. A load generator for sequential and parallel HTTP requests was implemented, performance testing of the Python service under different load scenarios was carried out, and the influence of MQTT communication on CPU usage and system response time was investigated.

Keywords: IoT, Python, FastAPI, MQTT, HTTP, Docker, Docker Compose, Mosquitto, load testing, containerization.

ЗМІСТ

ВСТУП

У сучасних умовах розвитку інформаційних технологій значного поширення набувають системи Інтернету речей. Такі системи використовуються для збору, передачі та обробки даних у різних сферах діяльності.

Для реалізації IoT-систем широко застосовуються одноплатні комп’ютери, зокрема Raspberry Pi. Вони дозволяють створювати недорогі та функціональні рішення, однак мають обмежені обчислювальні ресурси.

У зв’язку з цим виникає необхідність дослідження ефективності програмних засобів, що використовуються в таких системах. Особливо це стосується Python-орієнтованих сервісів, які активно застосовуються завдяки простоті реалізації та широким можливостям.

Також актуальним є аналіз різних підходів до передачі даних, зокрема із використанням протоколів HTTP та MQTT, а також застосування контейнеризації для моделювання роботи системи.

Метою роботи є дослідження ефективності Python-орієнтованих сервісів на платформі Raspberry Pi.

Об’єкт дослідження – IoT-системи на базі одноплатних комп’ютерів.

Предмет дослідження – ефективність Python-орієнтованих сервісів у процесі обробки та передачі даних.

Для досягнення поставленої мети необхідно виконати такі завдання:

1) спроектувати IoT-систему на базі Python-сервісів;

2) розробити програмну реалізацію системи;

3) реалізувати обмін даними між компонентами системи;

4) дослідити ефективність роботи системи;

5) візуалізувати результати дослідження;

6) запропонувати шляхи підвищення ефективності системи.

Результати роботи можуть бути використані при розробці IoT-систем.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СУЧАСНИХ РІШЕНЬ

1.1 Інтернет речей та архітектура IoT-систем

Інтернет речей (Internet of Things, IoT) – це концепція побудови мережі фізичних пристроїв, які здатні взаємодіяти між собою через мережу Інтернет, здійснювати збір, передачу та обробку даних без постійного втручання користувача [1]. До таких пристроїв належать сенсори, контролери, виконавчі механізми, одноплатні комп'ютери, мобільні пристрої та серверні системи. Взаємодія між ними забезпечується за допомогою мережевих технологій та спеціалізованих протоколів передачі даних.

Основною ідеєю IoT є створення середовища, у якому фізичні об'єкти можуть автоматично обмінюватися інформацією та реагувати на зміни зовнішніх умов. Завдяки цьому забезпечується автоматизація процесів, підвищення швидкості обробки інформації та зменшення необхідності постійного контролю з боку людини [2].

До складу IoT-систем входять різноманітні компоненти: сенсори, виконавчі пристрої, мережеві модулі передачі даних, обчислювальні вузли та сервери або хмарні платформи.

Важливою особливістю IoT-систем є автоматизований обмін інформацією між компонентами системи в режимі реального часу [2]. Для цього використовуються локальні мережі, бездротові технології Wi-Fi, Bluetooth, ZigBee, LoRaWAN, а також глобальна мережа Інтернет.

Як зазначається у дослідженні [3], ефективність IoT-систем значною мірою залежить від швидкості передачі даних, стабільності мережевої взаємодії та продуктивності обчислювальних вузлів.

Основними принципами функціонування IoT-систем є (рис. 1.1):

1) збір даних із фізичного середовища за допомогою сенсорів;

2) передача інформації мережею;

3) автоматизована обробка отриманих даних;

4) взаємодія між пристроями без прямого втручання користувача;

5) масштабованість та можливість інтеграції нових компонентів.

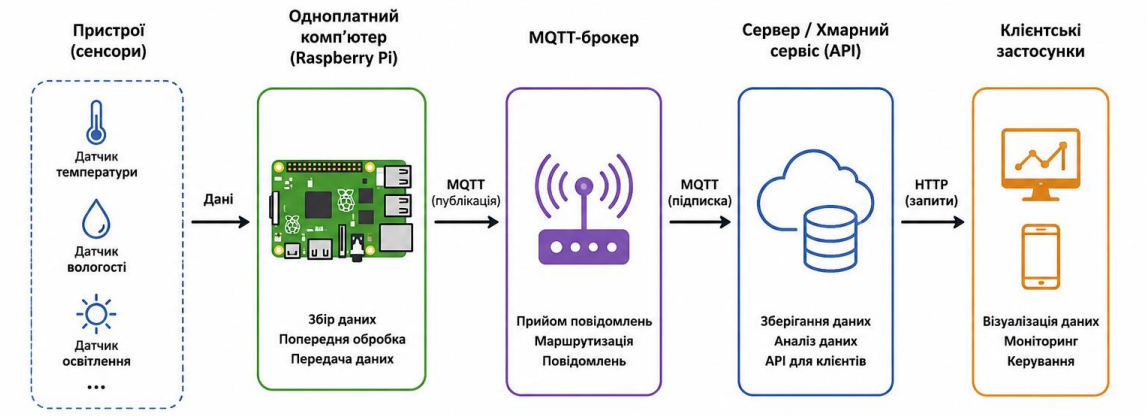

Рисунок 1.1 – Загальна схема взаємодії компонентів IoT-системи

Одним із ключових принципів IoT є масштабованість. Система повинна забезпечувати можливість підключення нових пристроїв без суттєвої зміни загальної архітектури. Ще одним важливим принципом є автономність роботи компонентів системи. IoT-пристрої здатні працювати без постійної участі користувача, самостійно виконуючи збір інформації, передачу даних та реагування на події.

Архітектура IoT-систем визначає структуру взаємодії між компонентами системи, способи передачі даних та принципи їх обробки. Побудова ефективної архітектури є важливим етапом створення IoT-рішення, оскільки саме від неї залежать швидкодія системи, стабільність роботи, можливість масштабування та ефективність використання ресурсів.

У більшості випадків IoT-системи реалізуються за багаторівневим принципом. Найбільш поширеною є трирівнева архітектура IoT-систем, яка складається з:

– рівня пристроїв;

– мережевого рівня;

– рівня обробки даних.

У більш складних системах додатково може використовуватись рівень прикладних сервісів та рівень управління безпекою (рис. 1.2).

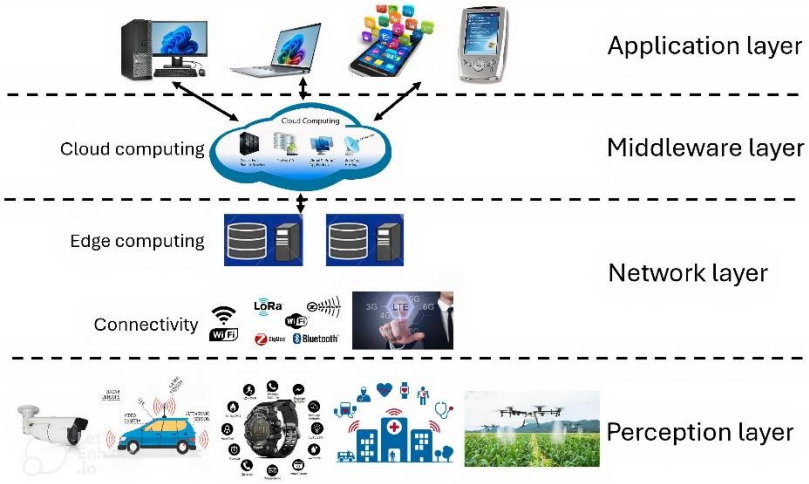

Рисунок 1.2 – Архітектура багаторівневої IoT-системи [4]

Рівень пристроїв є базовим рівнем IoT-системи. До нього входять сенсори, контролери, виконавчі механізми, мікроконтролери та одноплатні комп’ютери. Основним завданням цього рівня є збір інформації з фізичного середовища та первинна взаємодія із зовнішніми об’єктами.

На рівні пристроїв часто використовуються одноплатні комп’ютери Raspberry Pi та мікроконтролери ESP32 або Arduino. Вони забезпечують збір даних, базову локальну обробку інформації та передачу повідомлень мережею.

Мережевий рівень відповідає за передачу інформації між компонентами системи. Саме цей рівень забезпечує взаємодію між сенсорами, локальними вузлами обробки даних, серверами та хмарними платформами [5].

Для передачі даних можуть використовуватись: Ethernet, Wi-Fi, Bluetooth, ZigBee, LoRaWAN, мобільні мережі 4G/5G. Також на цьому рівні використовуються мережеві протоколи передачі даних, серед яких найбільш поширеними є HTTP, MQTT, CoAP та WebSocket. Вибір протоколу залежить від особливостей системи, обсягу передаваних даних, вимог до швидкодії та доступних апаратних ресурсів.

HTTP широко застосовується для реалізації REST API та взаємодії між веб-сервісами. MQTT, у свою чергу, орієнтований на системи з обмеженими ресурсами та забезпечує ефективну передачу повідомлень за моделлю publish/subscribe. У сучасних IoT-рішеннях MQTT часто використовується для організації взаємодії між великою кількістю пристроїв.

Рівень обробки даних виконує функції аналізу, обробки, збереження та візуалізації інформації [6]. На цьому рівні працюють сервери, бази даних, системи аналітики та хмарні сервіси.

Основними функціями рівня обробки є аналіз отриманих даних, фільтрація інформації, збереження результатів, формування статистики, прийняття рішень на основі отриманих даних.

У сучасних системах для обробки даних часто використовуються хмарні платформи, які забезпечують масштабованість та високу доступність сервісів. Разом із тим значного поширення набуває концепція edge computing, при якій частина обробки виконується безпосередньо на локальному пристрої.

Edge computing дозволяє зменшити затримки передачі даних, скоротити навантаження на центральний сервер, підвищити швидкість реагування системи та забезпечити роботу навіть при нестабільному мережевому з’єднанні.

Для IoT-систем, що працюють у режимі реального часу, використання edge computing є особливо важливим, оскільки дозволяє виконувати частину обчислень локально без необхідності постійного звернення до хмарної інфраструктури.

У деяких архітектурах додатково виділяють рівень прикладних сервісів, який забезпечує взаємодію користувача із системою. На цьому рівні реалізуються веб-інтерфейси, мобільні застосунки, системи моніторингу, панелі керування та засоби візуалізації інформації.

Саме прикладний рівень забезпечує відображення результатів роботи IoT-системи та надає користувачу можливість керування окремими компонентами (табл. 1.1).

Таблиця 1.1 – Основні рівні архітектури IoT-систем

Рівень	Основні функції
Рівень пристроїв	Збір даних із сенсорів
Мережевий рівень	Передача інформації між компонентами
Рівень обробки даних	Аналіз, збереження та обробка інформації
Рівень прикладних сервісів	Взаємодія користувача із системою

Значну роль відіграє питання безпеки IoT-систем. Через велику кількість підключених пристроїв виникають ризики несанкціонованого доступу, перехоплення даних та порушення роботи мережі. Тому сучасні IoT-рішення використовують механізми автентифікації, шифрування трафіку та контролю доступу.

Для невеликих IoT-систем значного поширення набуває підхід edge computing, при якому частина функцій обробки виконується безпосередньо на периферійному пристрої. Це дозволяє підвищити швидкодію системи та зменшити затримки передачі даних.

1.2 Одноплатні комп’ютери в системах IoT

Одноплатні комп’ютери (Single Board Computer, SBC) є важливим елементом сучасних систем Інтернету речей. Вони поєднують процесор, оперативну пам’ять, мережеві інтерфейси та периферійні модулі на одній друкованій платі, що дозволяє використовувати їх як компактні сервери або вузли локальної обробки даних [7].

SBC застосовуються для збору даних із сенсорів, виконання локальної обробки інформації, передачі даних до серверів та організації взаємодії між компонентами системи.

Основними перевагами одноплатних комп’ютерів є невелика вартість, компактні розміри, низьке енергоспоживання, підтримка мережевих технологій, можливість запуску операційних систем Linux, підтримка мов програмування високого рівня, зокрема Python [8].

Компактність SBC дозволяє використовувати їх у вбудованих системах, де важливими є обмежені габарити обладнання. Пристрої можуть встановлюватися у промислових системах моніторингу, транспортних засобах, системах автоматизації будівель та інших середовищах із обмеженим простором.

Низьке енергоспоживання є ще однією важливою перевагою SBC у порівнянні із традиційними серверними системами. Це дозволяє використовувати їх у автономних IoT-рішеннях, які працюють від акумуляторів або альтернативних джерел живлення.

Однією з ключових особливостей SBC є підтримка GPIO-інтерфейсів (General Purpose Input/Output), що забезпечують можливість безпосереднього підключення сенсорів та виконавчих механізмів [9]. Через GPIO можна організовувати зчитування даних із датчиків, керування реле та двигунами, роботу з дисплеями та обмін даними через UART, SPI та I2C.

Завдяки цьому SBC можуть безпосередньо інтегруватися у фізичні процеси та виконувати функції контролера IoT-системи (рис. 1.3).

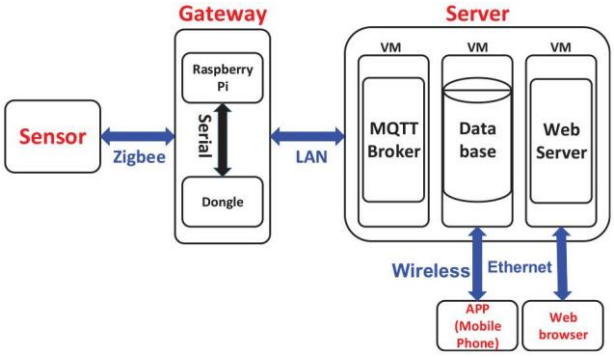

Рисунок 1.3 – Приклад використання одноплатного комп’ютера у структурі IoT-системи [10]

Важливою перевагою сучасних SBC є підтримка мережевих інтерфейсів. Більшість платформ оснащуються Ethernet, Wi-Fi та Bluetooth, що дозволяє реалізовувати бездротову взаємодію між компонентами IoT-системи. У деяких випадках додатково підтримуються модулі мобільного зв’язку або LoRaWAN.

У сучасних IoT-рішеннях SBC часто використовуються як локальні вузли попередньої обробки даних. Такий підхід дозволяє зменшити навантаження на центральний сервер, скоротити затримки передачі інформації, підвищити швидкість реагування системи та забезпечити автономність роботи окремих компонентів.

Особливо важливим це є для систем edge computing, у яких частина обчислень виконується безпосередньо на периферійному пристрої [6].

Разом із перевагами SBC мають і певні обмеження. Основними з них є: обмежений обсяг оперативної пам’яті, нижча продуктивність процесора порівняно із серверними системами, залежність стабільності роботи від системи охолодження та обмежені можливості масштабування [3].

При високому навантаженні або одночасній роботі декількох сервісів можуть виникати затримки обробки даних та зростання використання ресурсів системи. Це особливо актуально для Python-орієнтованих сервісів, які можуть споживати значний обсяг оперативної пам’яті.

Одноплатні комп’ютери можуть виконувати різні функції: від докального сервера до контролера сенсорної мережі (табл. 1.2). Серед найбільш поширених SBC можна виділити Raspberry Pi, Orange Pi, BeagleBone Black та NVIDIA Jetson Nano. Однак саме Raspberry Pi отримав найбільше поширення завдяки широкій підтримці програмного забезпечення, великій кількості документації та активній спільноті користувачів [11].

Таблиця 1.2 – Переваги та обмеження SBC у IoT -системах

Характеристика	Особливості
Вартість	Низька порівняно із серверними платформами
Енергоспоживання	Невисоке
Розміри	Компактні
Продуктивність	Обмежена порівняно із ПК та серверами
Інтеграція із сенсорами	Підтримка GPIO
Масштабованість	Обмежена апаратними ресурсами

Raspberry Pi є однією з найпоширеніших платформ для реалізації IoT-систем та задач edge computing. Платформа була розроблена Raspberry Pi

Foundation як компактний та доступний одноплатний комп’ютер для освітніх, дослідницьких і прикладних задач. Завдяки поєднанню невеликої вартості, достатньої продуктивності та широкої програмної підтримки Raspberry Pi активно використовується у сфері Інтернету речей.

Однією з головних причин популярності Raspberry Pi у сфері IoT є підтримка операційної системи Linux. Використання Linux забезпечує: підтримку багатозадачності, запуск мережевих сервісів, роботу із сучасними мовами програмування, використання контейнеризації та підтримку серверного програмного забезпечення.

Для Raspberry Pi доступна спеціалізована операційна система Raspberry Pi OS, яка базується на Debian Linux та містить необхідні засоби для роботи з периферією, мережею та GPIO-інтерфейсами (рис. 1.4).

Raspberry Pi часто використовується як: локальний сервер обробки даних, MQTT-брокер, шлюз передачі інформації, веб-сервер, edge-вузол або система моніторингу та керування.

Завдяки підтримці Python платформа є зручною для реалізації IoT-сервісів та експериментальних систем [12]. Python дозволяє швидко створювати мережеві застосунки, взаємодіяти із сенсорами та реалізовувати серверну логіку без значних витрат часу на розробку.

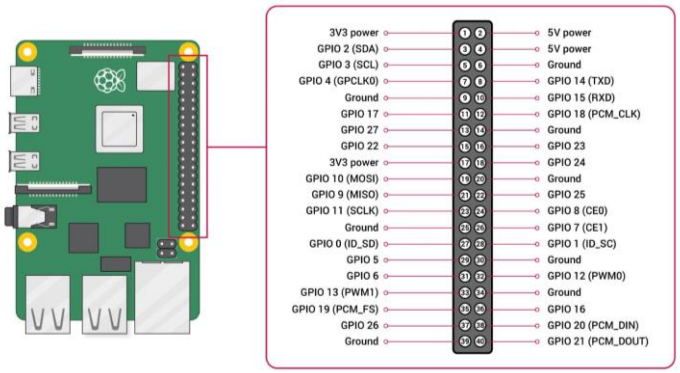

Рисунок 1.4 – Одноплатний комп’ютер Raspberry Pi та основні інтерфейси підключення

Важливою особливістю Raspberry Pi є підтримка GPIO (General Purpose Input/Output). GPIO-контакти дозволяють безпосередньо підключати температурні сенсори, датчики руху, реле, дисплеї, двигуни та інші периферійні модулі.

Через GPIO реалізується обмін даними за протоколами UART, SPI та I2C, що значно розширює можливості інтеграції платформи у IoT-системи.

Сучасні моделі Raspberry Pi також підтримують апаратні інтерфейси: USB, HDMI, Ethernet, Wi-Fi, Bluetooth та microSD. Це забезпечує можливість підключення додаткових пристроїв, зовнішніх накопичувачів, мережевого обладнання та периферії.

У сфері IoT Raspberry Pi широко використовується разом із технологіями контейнеризації. Платформа підтримує Docker, що дозволяє запускати окремі сервіси у вигляді ізольованих контейнерів. Контейнеризація забезпечує:

– ізоляцію компонентів системи;

– спрощення процесу розгортання;

– незалежність конфігурації сервісів;

– можливість перенесення системи між різними середовищами.

Використання Docker є особливо важливим для IoT-систем, у яких одночасно можуть працювати веб-сервіси, MQTT-брокери, системи моніторингу та засоби збору статистики.

Для задач обробки даних Raspberry Pi може виконувати функції MQTT-брокера, веб-сервера, локального сервера обробки даних, вузла edge computing або системи моніторингу [13].

Raspberry Pi часто використовується як локальний вузол edge computing. При такому підході частина обчислень виконується безпосередньо на периферійному пристрої без передачі всіх даних до хмарної інфраструктури [6].

Разом із перевагами Raspberry Pi має і певні обмеження. Через обмежені апаратні ресурси при високому навантаженні можливе зростання використання процесора, перевищення обсягу оперативної пам’яті,

збільшення затримок обробки запитів та зниження стабільності роботи Python-сервісів.

Особливо це проявляється при одночасному виконанні декількох контейнеризованих сервісів або великій кількості мережевих запитів. Тому для ефективного використання Raspberry Pi важливими є оптимізація програмного забезпечення та правильний вибір архітектури системи.

Як зазначає дослідження [3], Продуктивність IoT-платформи суттєво залежить від способу передачі даних, кількості одночасних підключень, складності обробки інформації та типу використовуваного програмного забезпечення.

У системах реального часу Raspberry Pi може використовуватись для попередньої обробки даних, фільтрації інформації або виконання нескладних аналітичних операцій. Для більш ресурсномістких задач обробки даних зазвичай використовуються хмарні сервіси або серверні системи (табл. 1.3).

Таблиця 1.3 – Основні характеристики Raspberry Pi 4 Model B

Характеристика	Значення
Процесор	Broadcom BCM2711 ARM Cortex-A72
Кількість ядер	4
Тактова частота	1,5 ГГц
Оперативна пам’ять	2-8 Гб
Мережеві інтерфейси	Wi-Fi, Bluetooth, Ethernet
Операційна система	Raspberry Pi OS, Linux
GPIO	40 контактів

1.3 Python-орієнтовані сервіси в IoT

Для реалізації IoT-сервісів найбільш поширеними Python-фреймворками є Flask та FastAPI. Вони використовуються для створення REST API, обробки HTTP-запитів, взаємодії із базами даних та реалізації серверної частини IoT-систем [14].

Веб-фреймворк є програмною платформою, яка містить готові компоненти для побудови веб-застосунків. У сфері IoT веб-фреймворки використовуються для прийому даних від сенсорів, передачі інформації між

компонентами системи, створення REST API, взаємодії із MQTT-брокерами, реалізації систем моніторингу та керування пристроями через веб-інтерфейси [13].

Flask є одним із найбільш поширених Python-фреймворків для створення веб-застосунків та API [14]. Даний фреймворк відноситься до категорії мікрофреймворків, оскільки містить лише базові засоби для реалізації веб-сервісу без великої кількості додаткових компонентів.

З плюсів Flask можна виділити просту структура проєкту, невелику кількість залежностей, легкість інтеграції з іншими бібліотеками, швидке створення невеликих сервісів та зручність для навчальних та експериментальних проєктів.

Завдяки простоті Flask часто використовується у невеликих IoT-системах, де необхідно швидко реалізувати сервер обробки даних або REST API. Фреймворк добре підходить для систем із невеликою кількістю одночасних підключень та помірним навантаженням.

Разом із перевагами Flask має і певні обмеження. Основним недоліком є синхронна модель обробки запитів. При великій кількості одночасних клієнтів це може призводити до збільшення затримок та зростання навантаження

Для вирішення проблем продуктивності у сучасних IoT-системах все частіше використовується FastAPI – сучасний асинхронний Python-фреймворк для створення високопродуктивних веб-сервісів [15].

Основними перевагами FastAPI є: висока продуктивність, підтримка асинхронної обробки, автоматична генерація документації API, підтримка сучасних стандартів Python, ефективна робота із JSON-даними та зручність створення REST API (рис. 1.5).

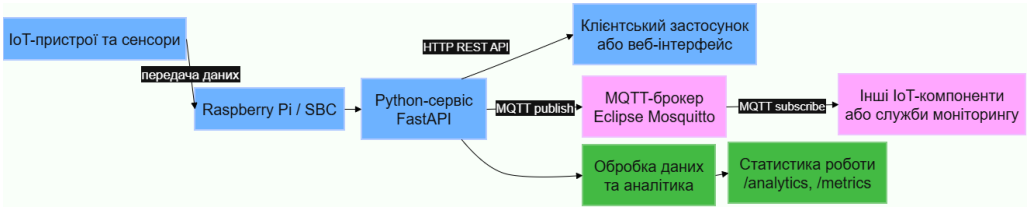

Рисунок 1.5 – Взаємодія Python-сервісу з компонентами IoT-системи

Однією з важливих переваг FastAPI є ефективна підтримка асинхронної взаємодії із зовнішніми сервісами, MQTT-брокерами та базами даних. Це дозволяє використовувати його у контейнеризованих IoT-системах, де одночасно працює декілька незалежних компонентів.

Для реалізації мережевої взаємодії у Python-сервісах широко використовуються REST API. REST (Representational State Transfer) є архітектурним підходом до побудови веб-сервісів, при якому обмін даними здійснюється через HTTP-запити.

Основними HTTP-методами REST API є:

– GET – отримання даних;

– POST – передача нових даних;

– PUT – оновлення інформації;

– DELETE – видалення даних.

REST API використовується для передачі показників сенсорів, отримання статистики, керування пристроями, моніторингу стану системи та взаємодії між контейнеризованими сервісами.

Важливим фактором для IoT-систем є продуктивність веб-фреймворку в умовах обмежених ресурсів. Використання асинхронних механізмів дозволяє зменшити затримки обробки запитів та підвищити ефективність використання процесорного часу (табл. 1.4).

Таблиця 1.4 – Порівняння Flask та FastAPI

Характеристика	Flask	FastAPI
Тип обробки	Синхронна	Асинхронна
Продуктивність	Середня	Висока
Простота використання	Висока	Висока
Підтримка REST API	Так	Так
Підтримка async/await	Обмежена	Повна

FastAPI часто розглядається як більш ефективний варіант для високонавантажених систем, тоді як Flask залишається популярним для невеликих сервісів та навчальних проєктів.

Для реалізації системи у даній роботі використовується Python-сервіс, який забезпечує прийом, обробку та передачу даних між компонентами системи. Основним фреймворком обрано FastAPI, оскільки він забезпечує підтримку асинхронної обробки запитів та дозволяє ефективно використовувати обмежені ресурси IoT-платформи.

Для забезпечення взаємодії між компонентами IoT-систем використовуються різні мережеві протоколи передачі даних. Від вибору протоколу значною мірою залежить швидкодія системи, навантаження на мережу, стабільність обміну повідомленнями та ефективність використання апаратних ресурсів.

Найбільш поширеними є протоколи HTTP та MQTT. Вони використовуються для передачі даних між сенсорами, серверами, веб-сервісами та іншими компонентами системи.

HTTP (HyperText Transfer Protocol) є стандартним протоколом передачі даних у веб-системах. Даний протокол широко застосовується для реалізації REST API та взаємодії між клієнтськими і серверними застосунками.

Принцип роботи HTTP базується на моделі request/response, при якій клієнт надсилає запит до сервера, а сервер формує відповідь. Такий підхід добре підходить для веб-застосунків та сервісів, де необхідна чітка структура взаємодії між компонентами.

Основними перевагами HTTP є: простота реалізації, підтримка більшістю сучасних платформ, сумісність із веб-технологіями, зручність створення REST API та підтримка браузерами та серверними системами.

Для передачі даних через HTTP найчастіше використовується формат JSON, який є зручним для обробки у Python-сервісах та веб-застосунках.

Разом із перевагами HTTP має і певні недоліки при використанні у IoT-системах. Основною проблемою є значний обсяг службової інформації у запитах та відповідях. Це призводить до збільшення мережевого навантаження, зростання використання процесорного часу, підвищення затримок передачі даних. Особливо помітними ці недоліки стають у системах

із великою кількістю пристроїв або при високій інтенсивності передачі повідомлень.

Для оптимізації обміну даними у системах Інтернету речей широко використовується MQTT (Message Queuing Telemetry Transport) – протокол передачі повідомлень, розроблений спеціально для систем із обмеженими ресурсами.

MQTT працює за моделлю publish/subscribe, при якій обмін повідомленнями здійснюється через спеціальний сервер – брокер повідомлень. Клієнти можуть:

– публікувати повідомлення у певні теми (topics);

– підписуватись на теми для отримання інформації;

– отримувати повідомлення від інших компонентів системи.

У такій архітектурі клієнтам не потрібно підтримувати пряме з'єднання між собою. Усі повідомлення передаються через MQTT-брокер, який виконує функції маршрутизації даних (рис. 1.6).

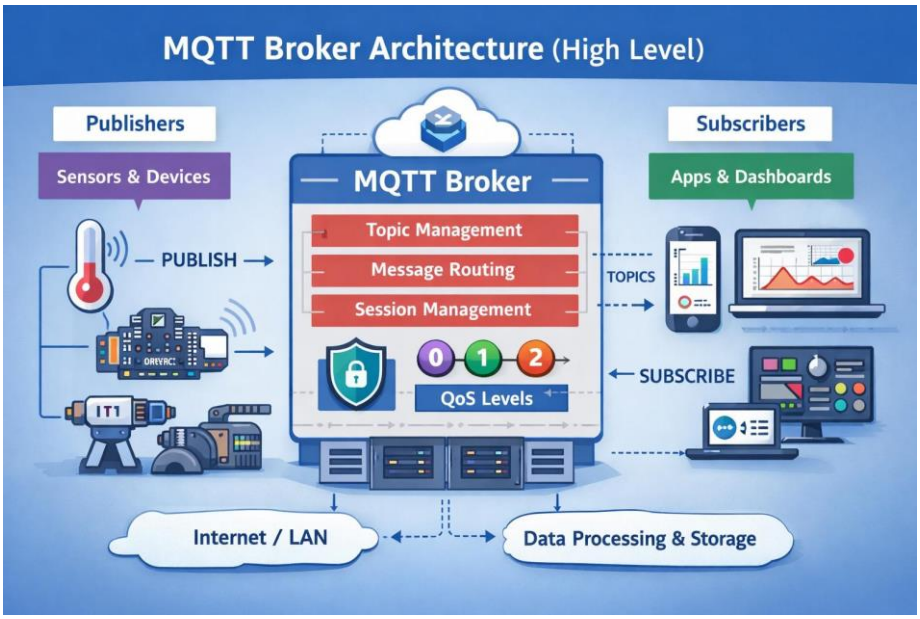

Рисунок 1.6 – Схема взаємодії компонентів через MQTT-брокер

Основними перевагами MQTT є: низьке мережеве навантаження, невеликий розмір повідомлень, підтримка роботи в нестабільних мережах,

ефективність при роботі на пристроях із обмеженими ресурсами та підтримка асинхронного обміну повідомленнями [16].

Ще однією важливою особливістю MQTT є підтримка різних рівнів якості доставки повідомлень (QoS – Quality of Service). Протокол підтримує три основні режими:

– QoS 0 – повідомлення доставляється без підтвердження;

– QoS 1 – доставка із підтвердженням отримання;

– QoS 2 – гарантована одноразова доставка повідомлення.

Використання QoS дозволяє адаптувати роботу системи залежно від вимог до надійності передачі даних.

У сучасних IoT-системах MQTT часто використовується для передачі телеметрії, взаємодії між сенсорами, обміну повідомленнями між сервісами, систем моніторингу, реалізації edge computing.

Для реалізації MQTT-взаємодії зазвичай використовуються брокери повідомлень, серед яких найбільш поширеним є Eclipse Mosquitto. Даний брокер характеризується низьким споживанням ресурсів та підтримкою роботи у контейнеризованих середовищах (табл. 1.5).

Таблиця 1.5 – Порівняння HTTP та MQTT

Характеристика	HTTP	MQTT
Тип взаємодії	Request/Response	Publish/Subscribe
Навантаження на мережу	Вище	Нижче
Швидкодія	Середня	Вища
Робота в IoT	Обмежена	Оптимізована
Підтримка браузерів	Так	Обмежена

У системах Інтернету речей HTTP та MQTT часто використовуються одночасно. HTTP застосовується для реалізації REST API та взаємодії із зовнішніми сервісами, тоді як MQTT використовується для внутрішнього обміну повідомленнями між компонентами IoT-системи.

MQTT демонструє кращу ефективність у IoT-системах із великою кількістю пристроїв та високою інтенсивністю передачі даних. Завдяки моделі

publish/subscribe протокол дозволяє мінімізувати мережеве навантаження та підвищити швидкість передачі повідомлень.

Разом із тим HTTP залишається важливим компонентом сучасних IoT-рішень завдяки простоті інтеграції із веб-технологіями та підтримці REST-архітектури.

1.4 Аналіз існуючих рішень та постановка задачі дослідження

На сьогодні системи Інтернету речей активно використовуються у різних сферах діяльності – промисловості, транспорті, медицині, аграрному секторі, системах моніторингу та побутовій автоматизації. Поширення IoT значною мірою пов’язане із розвитком бездротових мереж, хмарних сервісів та компактних обчислювальних платформ. У більшості сучасних рішень основними компонентами системи є сенсори, вузли локальної обробки даних, серверні сервіси та засоби мережевої взаємодії.

Для побудови IoT-систем широко використовуються одноплатні комп’ютери, серед яких найбільш поширеною платформою є Raspberry Pi. Даний пристрій поєднує невелику вартість, компактні розміри та підтримку повноцінної операційної системи Linux. Це дозволяє використовувати Raspberry Pi як локальний сервер обробки даних, MQTT-брокер або edge-вузол.

У сучасних IoT-рішеннях Raspberry Pi часто застосовується для збору та попередньої обробки даних, організації мережевої взаємодії, запуску Python-сервісів, роботи з MQTT-брокерами, контейнеризації сервісів [17].

Одним із найбільш поширених підходів є використання Python для реалізації серверної логіки та мережевих сервісів. Це пояснюється простотою розробки, великою кількістю бібліотек та підтримкою сучасних мережевих технологій. Python активно застосовується для створення REST API, обробки HTTP-запитів та організації взаємодії між компонентами системи.

Для створення веб-сервісів у Python найчастіше використовуються Flask та FastAPI. Flask характеризується простою структурою та невеликою кількістю залежностей, тому є зручним для невеликих IoT-проєктів. Водночас FastAPI забезпечує підтримку асинхронної обробки запитів та вищу продуктивність при роботі з великою кількістю одночасних підключень.

Важливу роль відіграє вибір протоколу передачі даних. Найбільш поширеними є HTTP та MQTT. HTTP традиційно використовується для реалізації REST API та взаємодії із веб-сервісами. Його основними перевагами є універсальність, простота інтеграції та підтримка більшістю сучасних платформ.

Разом із тим HTTP створює додаткове навантаження на мережу через значний обсяг службових даних. Це особливо помітно в умовах великої кількості пристроїв або високої інтенсивності передачі інформації.

Для IoT-систем більш оптимізованим рішенням є MQTT – легковаговий протокол передачі повідомлень, який працює за моделлю publish/subscribe. Використання MQTT дозволяє зменшити мережеве навантаження та підвищити ефективність обміну повідомленнями між компонентами системи.

Основними перевагами MQTT є: невеликий розмір повідомлень, ефективна робота в умовах обмежених ресурсів, підтримка асинхронної взаємодії та менше навантаження на мережу порівняно із HTTP.

У багатьох сучасних IoT-рішеннях також активно використовується контейнеризація. Docker дозволяє ізолювати окремі компоненти системи, спростити процес розгортання сервісів та забезпечити стабільність конфігурації застосунків [17]. Контейнерний підхід особливо актуальний для систем, де одночасно працює декілька незалежних сервісів.

Типова структура сучасної IoT-системи включає (рис. 1.7):

– Python-сервіс обробки даних;

– MQTT-брокер;

– веб-API;

– систему моніторингу;

– засоби контейнеризації та керування сервісами.

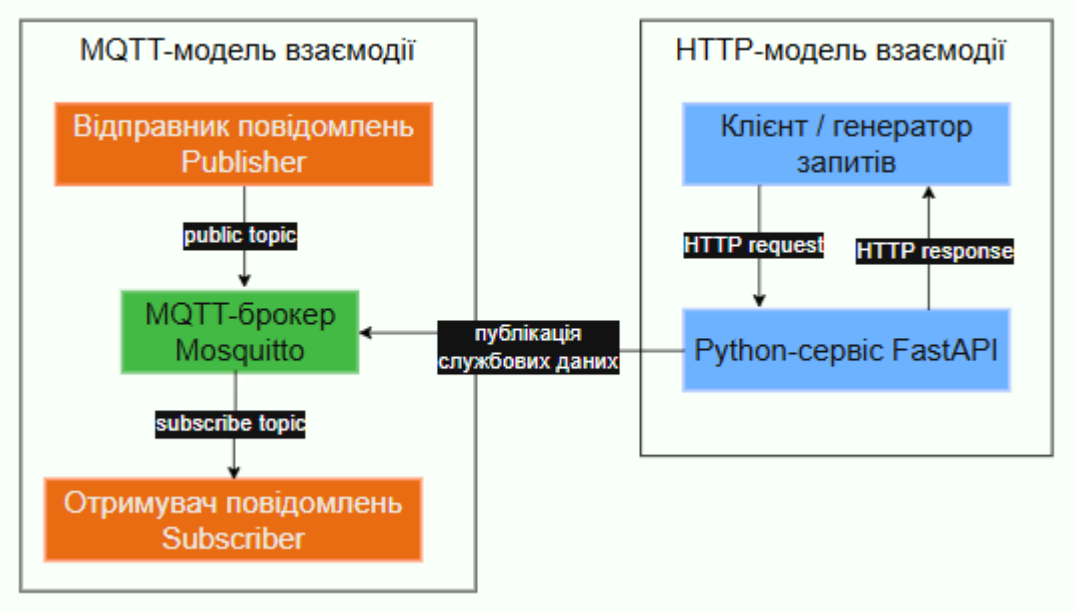

Рисунок 1.7 – Приклад структури сучасної IoT-системи

Разом із перевагами сучасні IoT-рішення мають і певні обмеження. Основними проблемами є:

1) зростання навантаження на процесор при великій кількості запитів;

2) збільшення використання оперативної пам’яті;

3) залежність продуктивності від способу передачі даних;

4) складність масштабування системи.

Особливо актуальними ці проблеми є для одноплатних комп’ютерів, які мають обмежені апаратні ресурси. При запуску декількох контейнеризованих сервісів або обробці великої кількості запитів можливе зниження швидкодії системи та збільшення затримок передачі даних.

Ефективність Python-орієнтованих сервісів суттєво залежить від архітектури системи, механізму обробки запитів та обраного протоколу передачі інформації. Особливо важливим це є для систем, що працюють у режимі реального часу.

Використання edge computing дозволяє зменшити затримки передачі даних та скоротити навантаження на центральні сервери. Водночас це

підвищує вимоги до локального обчислювального вузла та потребує оптимізації програмного забезпечення.

Більшість існуючих досліджень орієнтовані або на використання серверних платформ, або на хмарну обробку інформації. Питання ефективності Python-сервісів в умовах обмежених ресурсів Raspberry Pi, особливо при використанні контейнеризації та асинхронних сервісів, досліджене недостатньо.

У даній роботі пропонується реалізувати контейнеризовану IoT-систему із використанням Python-сервісу, генератора навантаження та MQTT-брокера. Для моделювання умов роботи Raspberry Pi планується використання Docker-контейнерів із обмеженням процесорного часу та оперативної пам’яті.

Основними задачами дослідження є:

– аналіз швидкодії Python-сервісу;

– оцінка використання процесора та оперативної пам’яті;

– порівняння HTTP та MQTT;

– дослідження ефективності Flask та FastAPI;

– аналіз роботи системи в умовах обмежених ресурсів.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА ОБҐРУНТУВАННЯ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 Обґрунтування вибору апаратної платформи

Для реалізації IoT-систем можуть використовуватись різні апаратні платформи, серед яких мікроконтролери, одноплатні комп’ютери та серверні рішення. Вибір платформи залежить від необхідної продуктивності, способу обробки даних, мережевих можливостей та особливостей програмного забезпечення.

Мікроконтролери, такі як Arduino або ESP32, характеризуються низьким енергоспоживанням та простотою використання, однак мають обмежені можливості щодо запуску багатокомпонентних мережевих сервісів і контейнеризації. Вони добре підходять для роботи із сенсорами, проте менш придатні для виконання складної серверної логіки.

Для задач, пов’язаних із запуском Python-сервісів, веб-фреймворків та MQTT-брокерів, більш доцільним є використання одноплатних комп’ютерів. Серед таких платформ найбільшого поширення набув Raspberry Pi [11]. Платформа підтримує операційну систему Linux, Docker-контейнери та сучасні засоби мережевої взаємодії, що робить її придатною для задач IoT та edge computing (табл. 2.1).

Таблиця 2.1 – Порівняння платформ для IoT-систем

Платформа	Переваги	Обмеження
Arduino	Низьке енергоспоживання	Відсутність Linux
ESP32	Wi-Fi та Bluetooth	Обмежені ресурси
Raspberry Pi	Підтримка Docker та Python	Вище енергоспоживання
Серверні системи	Висока продуктивність	Висока вартість

Для реалізації дослідження обрано Raspberry Pi, оскільки дана платформа забезпечує достатню продуктивність для запуску Python-сервісів та підтримує сучасні засоби контейнеризації.

Як зазначається у документації Raspberry Pi Foundation [11], платформа підтримує багатозадачність, мережеві сервіси та інструменти розробки, необхідні для створення IoT-систем.

У даній роботі фізичний Raspberry Pi не використовується. Натомість моделювання роботи платформи реалізується шляхом обмеження ресурсів Docker-контейнера, що дозволяє наблизити умови роботи системи до реальних обмежень одноплатного комп'ютера.

2.2 Вибір програмних засобів

Для реалізації серверної частини IoT-системи у даній роботі використовується мова програмування Python. Вибір Python обумовлений простотою розробки, великою кількістю бібліотек та підтримкою сучасних мережевих технологій [18]. На сьогодні Python є однією з найпоширеніших мов у сфері Інтернету речей, автоматизації та обробки даних.

Однією з основних переваг Python є зрозумілий синтаксис, який дозволяє швидко реалізовувати програмні рішення та спрощує підтримку коду. Це особливо важливо для IoT-систем, де часто необхідно інтегрувати різні компоненти та забезпечувати взаємодію між ними.

Python активно використовується для: створення веб-сервісів, реалізації REST API, роботи з MQTT-протоколом, обробки даних із сенсорів та автоматизації мережевої взаємодії [19].

Важливою перевагою мови є підтримка багатоплатформності. Python-сервіси можуть запускатися як на серверних системах, так і на одноплатних комп'ютерах Raspberry Pi під керуванням Linux. Це дозволяє використовувати однакові програмні рішення у різних середовищах виконання.

Для реалізації веб-сервісу у роботі використовується FastAPI, який підтримує асинхронну обробку запитів та характеризується високою продуктивністю. Використання асинхронних механізмів дозволяє ефективно

працювати з великою кількістю одночасних запитів та зменшувати затримки під час передачі даних (рис. 2.1).

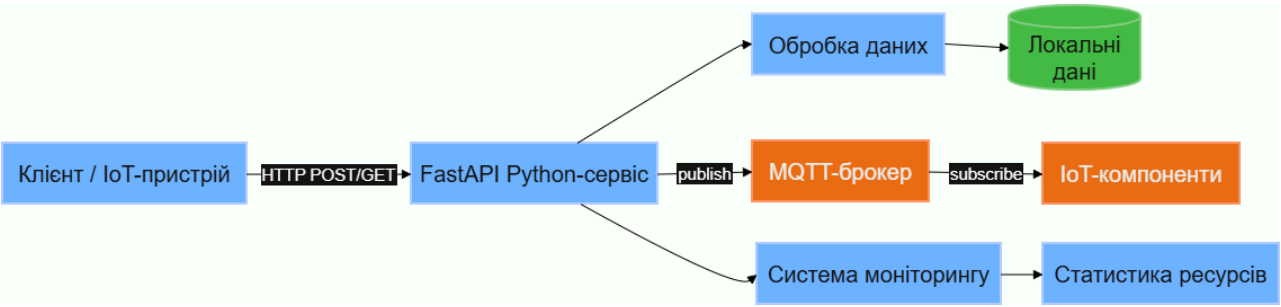

Рисунок 2.1 – Взаємодія Python-сервісу з компонентами системи

Python також забезпечує підтримку MQTT через бібліотеки Eclipse Paho MQTT Client та asyncio-механізми [20]. Це дозволяє організувати обмін повідомленнями між компонентами IoT-системи у режимі реального часу.

Серед основних переваг Python у сфері IoT можна виділити: простоту розробки, велику кількість бібліотек, підтримку асинхронного програмування, інтеграцію з Docker та Linux та підтримку мережевих протоколів і веб-технологій.

Як зазначає документація FastAPI [15], використання асинхронних Python-сервісів дозволяє підвищити ефективність використання ресурсів системи та забезпечити кращу продуктивність при роботі в умовах обмежених апаратних ресурсів.

У даній роботі Python використовується як основний засіб реалізації серверної логіки, взаємодії між компонентами системи та організації передачі даних у контейнеризованому IoT-середовищі.

Для реалізації системи використовуються сучасні програмні засоби та бібліотеки, які забезпечують обробку даних, мережеву взаємодію та контейнеризацію компонентів системи. Вибір інструментів здійснювався з урахуванням особливостей IoT-середовища та необхідності роботи в умовах обмежених ресурсів.

Основним веб-фреймворком обрано FastAPI. Даний фреймворк використовується для реалізації REST API та обробки HTTP-запитів. FastAPI підтримує асинхронне програмування, що дозволяє ефективно працювати з великою кількістю одночасних підключень [15].

Для організації MQTT-взаємодії використовується бібліотека Eclipse Paho MQTT Client [20]. Вона забезпечує підключення до MQTT-брокера, публікацію повідомлень та підписку на теми. Бібліотека є однією з найбільш поширених реалізацій MQTT-клієнта для Python та активно використовується у IoT-проєктах.

Контейнеризація сервісів реалізується за допомогою Docker [17]. Використання контейнерів дозволяє ізолювати окремі компоненти системи, спростити процес розгортання та забезпечити стабільність конфігурації середовища виконання. Для запуску всієї системи використовується Docker Compose, який забезпечує автоматизовану взаємодію між контейнерами.

У ролі MQTT-брокера використовується Eclipse Mosquitto. Даний брокер характеризується низьким споживанням ресурсів та добре підходить для IoT-систем і роботи на одноплатних комп’ютерах.

Використання зазначених інструментів дозволяє реалізувати IoT-систему, придатну для проведення експериментального дослідження ефективності Python-орієнтованих сервісів (табл. 2.2).

Таблиця 2.2 – Використані програмні засоби

Засіб	Призначення
Python	Реалізація серверної логіки
FastAPI	Створення веб-сервісу
Eclipse Paho MQTT	Робота з MQTT
Docker	Контейнеризація
Mosquitto	MQTT-брокер

2.3 Архітектура системи

Під час проєктування системи основна увага приділялась побудові простої та гнучкої структури, яка дозволяє дослідити роботу

Python-орієнтованих сервісів в умовах обмежених апаратних ресурсів. Оскільки метою роботи є аналіз ефективності сервісів на платформі Raspberry Pi, архітектура системи повинна забезпечувати можливість моделювання навантаження, передачі повідомлень між компонентами та контролю використання ресурсів.

У роботі використовується контейнеризований підхід, при якому кожен компонент системи запускається у окремому Docker-контейнері. Така структура дозволяє ізолювати сервіси один від одного, спростити конфігурацію та забезпечити незалежне тестування окремих модулів системи.

Архітектура системи складається із трьох основних компонентів (рис. 2.2):

– Python-сервіс обробки даних;

– генератор навантаження;

– MQTT-брокер Eclipse Mosquitto.

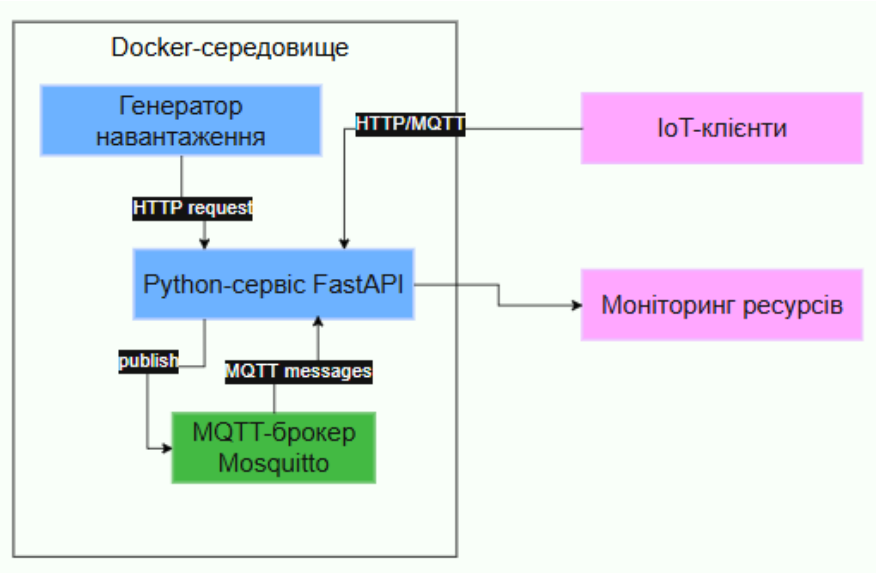

Рисунок 2.2 – Архітектура реалізованої IoT-системи

Центральним елементом системи є Python-сервіс, реалізований із використанням FastAPI. Сервіс виконує функції прийому HTTP-запитів, обробки даних та взаємодії із MQTT-брокером. Через REST API сервіс

отримує запити від генератора навантаження, виконує обробку інформації та формує відповідь клієнту.

Вибір FastAPI обумовлений підтримкою асинхронної обробки запитів та ефективною роботою із великою кількістю одночасних підключень [15]. Для IoT-систем це є важливим фактором, оскільки навіть при обмежених ресурсах необхідно забезпечити стабільну обробку мережевих запитів.

Другим компонентом системи є генератор навантаження. Він використовується для створення тестових HTTP-запитів до Python-сервісу та моделювання роботи клієнтів IoT-системи. Генератор дозволяє змінювати кількість запитів та інтенсивність їх передачі, що дає можливість аналізувати поведінку сервісу при різному навантаженні.

Для передачі повідомлень між компонентами використовується MQTT-брокер Eclipse Mosquitto [13]. MQTT-взаємодія реалізована за моделлю publish/subscribe, при якій компоненти системи обмінюються повідомленнями через брокер без необхідності прямого з’єднання між собою.

Такий підхід дозволяє зменшити мережеве навантаження, спростити взаємодію між сервісами, забезпечити асинхронний обмін повідомленнями та моделювати роботу реальної IoT-системи.

На відміну від класичної моделі request/response, MQTT дозволяє організувати більш гнучкий механізм передачі даних між компонентами системи. Це особливо важливо для IoT-середовища, де одночасно може працювати велика кількість пристроїв із обмеженими ресурсами.

Усі компоненти системи взаємодіють через внутрішню Docker-мережу. Для кожного контейнера використовується окреме мережеве ім’я, завдяки чому сервіси можуть звертатись один до одного без використання зовнішніх IP-адрес (рис. 2.3).

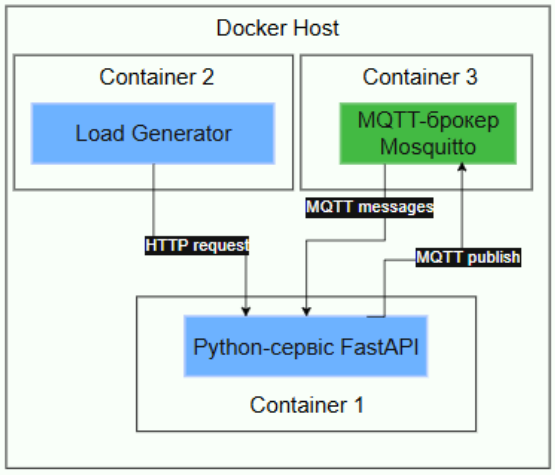

Рисунок 2.3 – Взаємодія контейнерів у Docker-мережі

Контейнеризований підхід також дозволяє окремо керувати параметрами кожного сервісу. Зокрема, для Python-сервісу можуть встановлюватись обмеження процесорного часу та оперативної пам’яті, що використовується для моделювання умов роботи Raspberry Pi.

Ще однією перевагою такої архітектури є можливість масштабування системи. Наприклад, генератор навантаження або Python-сервіс можуть запускатись у декількох екземплярах без суттєвої зміни загальної структури системи. Це дозволяє проводити тестування при різних рівнях навантаження та досліджувати поведінку сервісів у складніших умовах.

Архітектура системи побудована таким чином, щоб забезпечити можливість масштабування та модифікації окремих компонентів. Наприклад, Python-сервіс або MQTT-брокер можуть бути замінені без суттєвої зміни загальної структури системи. Це є важливою перевагою контейнеризованого підходу (табл. 2.3).

Таблиця 2.3 – Основні компоненти системи

Компонент	Призначення
Python-сервіс	Обробка запитів та даних
Генератор навантаження	Створення тестових запитів
MQTT-брокер	Передача повідомлень
Docker	Контейнеризація компонентів

Як зазначається у документації Docker [17], контейнеризація дозволяє ефективно використовувати апаратні ресурси та забезпечує стабільність роботи сервісів у різних середовищах виконання. Для IoT-систем це особливо актуально через обмеженість ресурсів периферійних пристроїв.

У даній роботі архітектура системи орієнтована на моделювання умов роботи IoT-пристрою із обмеженими ресурсами та подальше дослідження ефективності Python-орієнтованих сервісів при різних рівнях навантаження. Такий підхід дозволяє оцінити особливості роботи контейнеризованих сервісів та визначити найбільш ефективні механізми взаємодії між компонентами системи.

2.4 Реалізація системи у середовищі Docker

Для реалізації системи у даній роботі використовується контейнерний підхід на основі Docker. Основною причиною вибору Docker є можливість запуску окремих компонентів системи у незалежних середовищах із власними параметрами та обмеженнями ресурсів [17]. Це дозволяє моделювати роботу IoT-системи без використання фізичного Raspberry Pi та спрощує процес тестування.

У процесі реалізації було створено три окремі контейнери:

– контейнер Python-сервісу;

– контейнер генератора навантаження;

– контейнер MQTT-брокера Eclipse Mosquitto.

Кожен контейнер виконує окрему функцію та взаємодіє з іншими компонентами через внутрішню Docker-мережу (рис. 2.4).

Рисунок 2.4 – Структура контейнеризованої системи

Основним компонентом є контейнер Python-сервісу. Саме у ньому запускається веб-сервіс, який обробляє HTTP-запити та взаємодіє із MQTT-брокером. Для створення контейнера використовувався Dockerfile, у якому визначено базовий Python-образ, робочу директорію та параметри запуску сервісу (ліст. 2.1).

Лістинг 2.1 – Dockerfile Python-сервісу

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

кінець лістингу 2.1

Після збірки контейнера веб-сервіс стає доступним через порт 8000. Для взаємодії між контейнерами використовується внутрішня Docker-мережа, завдяки чому сервіси можуть звертатися один до одного за іменами контейнерів.

Другим компонентом системи є контейнер генератора навантаження. Його задача полягає у створенні HTTP-запитів до Python-сервісу для моделювання роботи клієнтів IoT-системи. Генератор запускається окремо та дозволяє змінювати інтенсивність навантаження під час тестування.

Третім компонентом є MQTT-брокер Eclipse Mosquitto [13]. Брокер запускається у власному контейнері та забезпечує передачу повідомлень між сервісами за моделлю publish/subscribe.

Для автоматизації запуску всієї системи використовувався Docker Compose (рис. 2.5). У конфігураційному файлі описані всі контейнери, параметри мережі, відкриті порти та залежності між сервісами (ліст. 2.2).

Лістинг 2.2 – конфігурація Docker Compose

```
services:
  app:
    build: .
    ports:
```

```
      - "8000:8000"

  load_generator:
    build: ./generator

  mosquitto:
    image: eclipse-mosquitto
    ports:
      - "1883:1883"
```

кінець лістингу 2.2

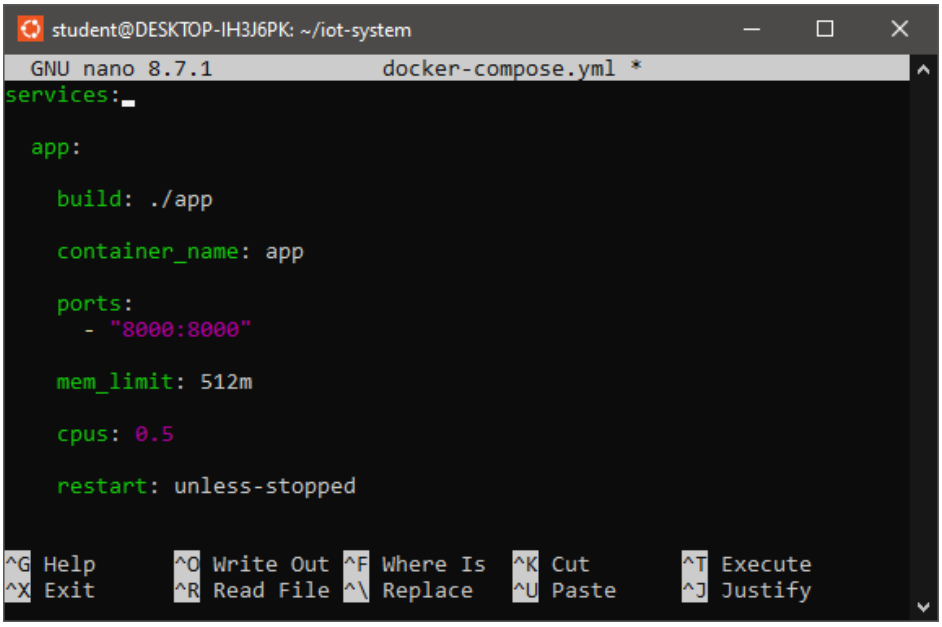

Рисунок 2.5 – Частина кофігурації у Docker Compose

Одним із ключових елементів реалізації є моделювання обмежених ресурсів. Для контейнера Python-сервісу були встановлені обмеження процесорного часу та оперативної пам’яті, що дозволяє наблизити умови роботи системи до характеристик Raspberry Pi (ліст. 2.3).

Лістинг 2.3 – Конфігураційні параметри Docker Compose

```
services:
  app:
    mem_limit: 512m
    cpus: 0.5
```

кінець лістингу 2.3

Встановлення обмежень дозволяє дослідити поведінку Python-сервісу при дефіциті ресурсів та оцінити стабільність роботи системи при збільшенні навантаження.

Для організації взаємодії між контейнерами Docker Compose автоматично створює окрему внутрішню мережу. Це дозволяє ізолювати систему від зовнішнього середовища та забезпечити стабільний обмін даними між сервісами.

У процесі тестування контроль роботи контейнерів виконувався за допомогою стандартних інструментів Docker. Для моніторингу використання процесора та оперативної пам’яті застосовувалась команда docker stats, яка дозволяє в режимі реального часу відстежувати навантаження контейнерів (рис. 2.6).

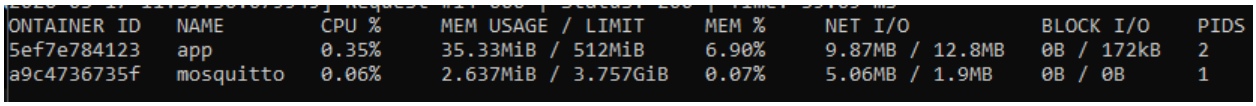

Рисунок 2.6 – Моніторинг використання ресурсів контейнерів (docker stats)

Таким чином, у межах роботи було реалізовано контейнеризовану IoT-систему, яка дозволяє досліджувати ефективність Python-орієнтованих сервісів у середовищі обмежених ресурсів. Використання Docker забезпечило зручне розгортання системи, ізоляцію компонентів та можливість проведення експериментального тестування при різних рівнях навантаження.

2.5 Моделювання обмежених ресурсів

Для дослідження ефективності Python-орієнтованих сервісів у середовищі IoT необхідно враховувати особливості роботи одноплатних комп’ютерів. На відміну від серверних систем, Raspberry Pi має обмежені апаратні ресурси,

тому при високому навантаженні можливе збільшення часу відповіді сервісів та зростання використання оперативної пам’яті.

У даній роботі моделювання умов роботи Raspberry Pi реалізовано шляхом обмеження ресурсів Docker-контейнера, у якому запускається Python-сервіс. Такий підхід дозволяє проводити тестування без використання фізичного пристрою та швидко змінювати параметри середовища залежно від задач дослідження.

Основною задачею моделювання було:

– оцінити стабільність роботи сервісу;

– дослідити поведінку системи при збільшенні навантаження;

– визначити використання процесора та оперативної пам’яті;

– проаналізувати швидкість обробки HTTP-запитів.

Для наближення умов тестування до характеристик Raspberry Pi контейнер Python-сервісу запускався з обмеженням процесорного часу та оперативної пам’яті (рис. 2.7). У межах дослідження використовувались такі параметри:

– CPU – 0,5 ядра;

– оперативна пам’ять – 512 МБ.

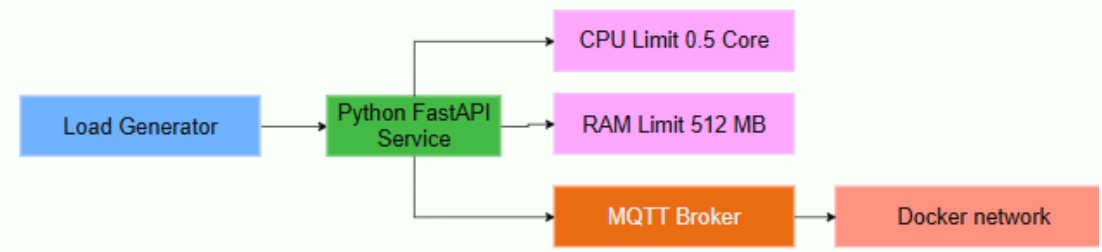

Рисунок 2.7 – Модель тестування сервісу в умовах обмежених ресурсів

Тестування системи проводилось у декількох режимах навантаження. На початковому етапі виконувалась перевірка стабільності роботи сервісу при невеликій кількості HTTP-запитів. Після цього інтенсивність передачі запитів поступово збільшувалась, що дозволяло оцінити зміну використання ресурсів та часу відповіді сервісу.

Під час експерименту змінювались:

– кількість HTTP-запитів;

– інтервал між запитами;

– тривалість навантаження;

– кількість одночасних звернень до API.

Окремо аналізувалась робота системи при одночасному використанні HTTP та MQTT-взаємодії. У цьому режимі Python-сервіс не лише обробляв HTTP-запити, а й взаємодіяв із MQTT-брокером, що створювало додаткове навантаження на контейнер (рис. 2.8).

Для оцінки ефективності системи фіксувались такі показники:

– завантаження CPU;

– використання оперативної пам’яті;

– час відповіді сервісу;

– стабільність роботи контейнера;

– швидкість обробки запитів.

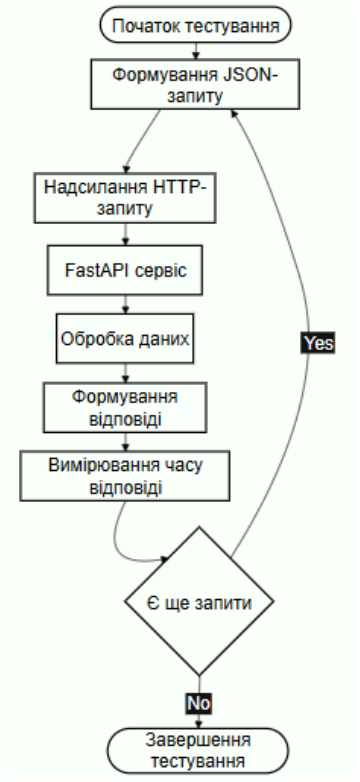

Рисунок 2.8 – Процес створення навантаження на Python-сервіс

Моніторинг роботи контейнерів виконувався за допомогою стандартних інструментів Docker. Основним засобом контролю була команда docker stats, яка дозволяє відстежувати використання ресурсів у режимі реального часу.

Під час тестування спостерігалось поступове збільшення навантаження на процесор при зростанні кількості одночасних запитів. Найбільше використання ресурсів виникало під час комбінованої роботи HTTP-запитів та MQTT-обміну повідомленнями (табл. 2.4).

Таблиця 2.4 – Параметри моделювання ресурсів

Параметр	Значення
CPU	0,5 ядра
Оперативна пам’ять	512 Мб
Тип контейнера	Docker
MQTT-брокер	Eclipse Mosquitto
Веб-сервіс	FastAPI

Використання Docker для моделювання обмежених ресурсів дозволило створити середовище, наближене до реальних умов роботи Raspberry Pi. Такий підхід дав можливість провести експериментальне дослідження ефективності Python-сервісів та оцінити вплив навантаження на продуктивність IoT-системи (рис. 2.9).

```
student@DESKTOP-IH3J6PK:~/iot-system$ cat docker-compose.yml
services:

  app:

    build: ./app

    container_name: app

    ports:
      - "8000:8000"

    mem_limit: 512m

    cpus: 0.5

    restart: unless-stopped

    depends_on:
      - mosquitto
```

Рисунок 2.9 – Моніторинг використання ресурсів контейнера

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ PYTHON-ОРІЄНТОВАНИХ СЕРВІСІВ

3.1 Проєктування структури IoT-системи

Загальна схема системи розроблена з урахуванням необхідності моделювання IoT-середовища в умовах обмежених апаратних ресурсів. Основною задачею реалізованої структури є забезпечення взаємодії між сервісом обробки даних, генератором навантаження та MQTT-брокером у межах єдиного середовища тестування.

Реалізована система складається з трьох компонентів (рис. 3.1):

– Python-сервісу обробки даних;

– генератора навантаження;

– MQTT-брокера Eclipse Mosquitto.

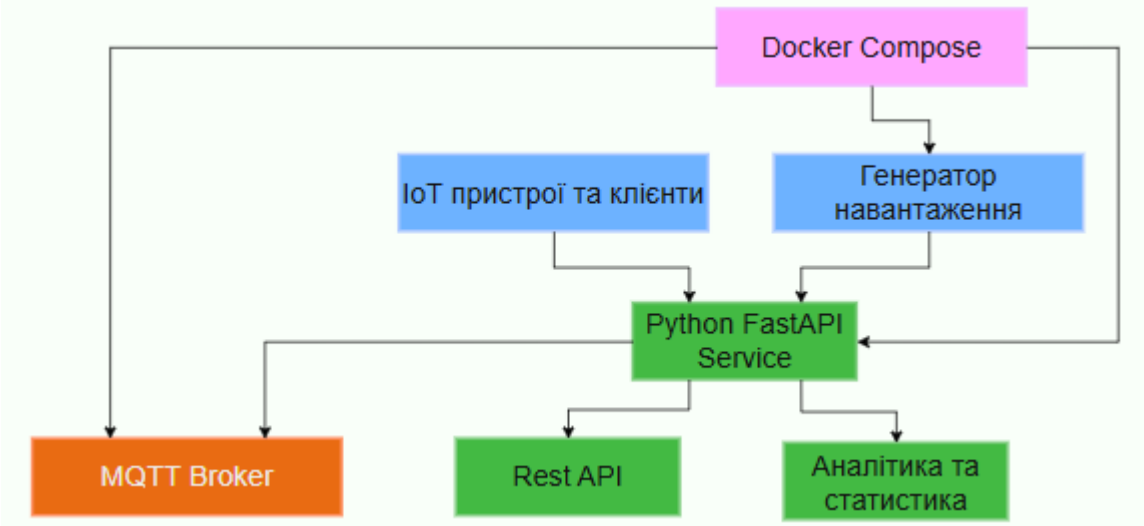

Рисунок 3.1 – Загальна схема реалізованої системи

Центральним елементом системи є Python-сервіс, який виконує прийом HTTP-запитів та обробку отриманих даних. Саме цей компонент використовується як основний об’єкт дослідження під час аналізу продуктивності та використання ресурсів.

Генератор навантаження використовується для створення серії HTTP-запитів до сервісу. Його задача полягає у моделюванні роботи клієнтів IoT-системи та формуванні контрольованого навантаження під час тестування.

Для організації передачі повідомлень між компонентами використовується MQTT-брокер. Через брокер реалізується асинхронний обмін повідомленнями, що дозволяє наблизити структуру системи до реальних IoT-рішень.

Генератор навантаження надсилає HTTP-запити до Python-сервісу. Після обробки запиту сервіс формує відповідь та, за необхідності, передає повідомлення через MQTT-брокер (рис. 3.2). Такий підхід дозволяє одночасно використовувати HTTP-взаємодію та MQTT-обмін повідомленнями у межах одного середовища.

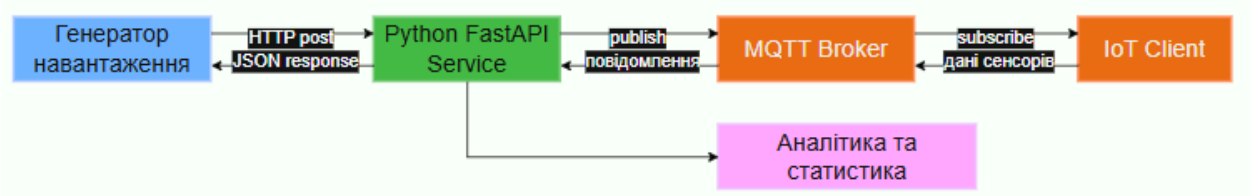

Рисунок 3.2 – Схема передачі даних між компонентами системи

Окрему увагу під час реалізації було приділено моделюванню обмежених ресурсів. Для сервісу обробки даних встановлювались обмеження процесорного часу та оперативної пам’яті, що дозволяє наблизити умови роботи системи до характеристик Raspberry Pi та дослідити поведінку сервісу при збільшенні навантаження.

Запропонована схема системи дозволяє виконувати подальше експериментальне дослідження продуктивності Python-орієнтованих сервісів та аналізувати особливості роботи IoT-середовища при різних режимах навантаження.

Після формування загальної структури системи було реалізовано механізм взаємодії між її компонентами. Основною задачею на цьому етапі

стало забезпечення стабільного обміну даними між сервісом обробки запитів, генератором навантаження та MQTT-брокером у процесі тестування системи.

У реалізованій системі використовуються два способи передачі даних:

– HTTP-взаємодія;

– MQTT-обмін повідомленнями.

HTTP використовується для передачі запитів від генератора навантаження до Python-сервісу. Через REST API генератор надсилає тестові дані, після чого сервіс виконує їх обробку та формує відповідь (рис. 3.3).

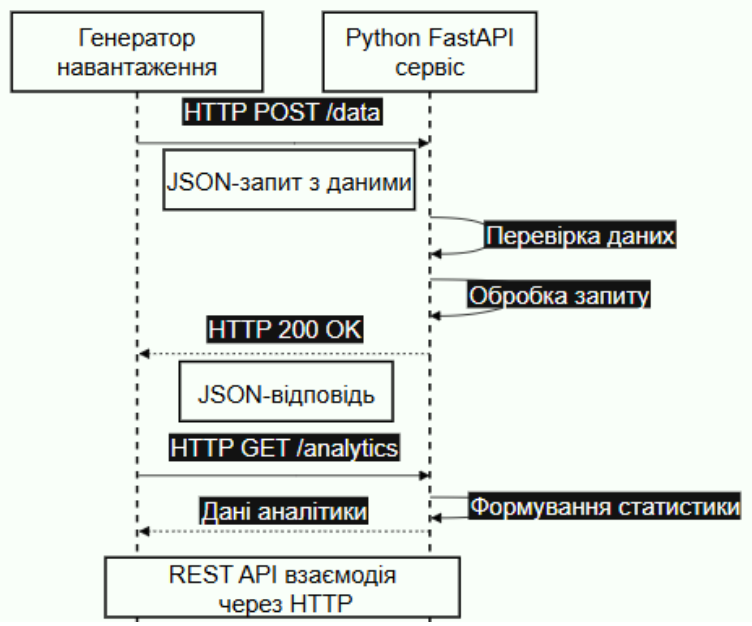

Рисунок 3.3 – Схема HTTP-взаємодії між компонентами

Під час тестування генератор навантаження створює серію HTTP-запитів із різною інтенсивністю. Це дозволяє досліджувати поведінку сервісу при збільшенні кількості звернень та аналізувати стабільність його роботи в умовах обмежених ресурсів.

Для передачі повідомлень між компонентами системи використовується MQTT-протокол. Python-сервіс взаємодіє з MQTT-брокером Eclipse Mosquitto та може передавати або отримувати повідомлення через MQTT-теми.

У системі використовуються такі MQTT-теми:

– iot/data;

– iot/status;

– iot/metrics.

Тема iot/data використовується для передачі основних повідомлень системи. Через iot/status передається інформація про стан сервісу, а iot/metrics застосовується для передачі службових даних та результатів моніторингу (рис. 3.4).

Рисунок 3.4 – Схема MQTT-обміну повідомленнями

Взаємодія через MQTT реалізована за моделлю publish/subscribe. У такій моделі Python-сервіс може публікувати повідомлення у певну тему або підписуватись на отримання повідомлень від брокера. Це дозволяє організувати асинхронний обмін даними між компонентами без необхідності постійного прямого з'єднання.

У процесі роботи системи після отримання HTTP-запиту Python-сервіс виконує:

– прийом даних;

– обробку запиту;

– формування HTTP-відповіді;

– передачу MQTT-повідомлення.

Такий підхід дозволяє одночасно використовувати веб-взаємодію та IoT-механізм передачі даних у межах однієї системи.

Для взаємодії між компонентами використовується внутрішня Docker-мережа. Усі сервіси звертаються один до одного через імена контейнерів, що спрощує конфігурацію системи та забезпечує стабільний обмін даними між компонентами.

Під час тестування особлива увага приділялась стабільності передачі повідомлень та коректності взаємодії між HTTP і MQTT-компонентами системи. Також аналізувалась поведінка сервісу при одночасній обробці HTTP-запитів та MQTT-повідомлень в умовах обмеження процесорного часу та оперативної пам’яті.

Реалізований механізм взаємодії компонентів дозволяє моделювати типову IoT-систему із підтримкою асинхронного обміну повідомленнями та забезпечує основу для подальшого експериментального дослідження продуктивності Python-орієнтованих сервісів.

3.2 Реалізація програмної частини

Програмна частина системи складається з трьох основних елементів: сервісу обробки даних, генератора навантаження та MQTT-брокера. Python-сервіс виконує роль основного обробника запитів, генератор навантаження використовується для імітації роботи клієнтів, а MQTT-брокер забезпечує передавання службових повідомлень між компонентами системи (рис. 3.5).

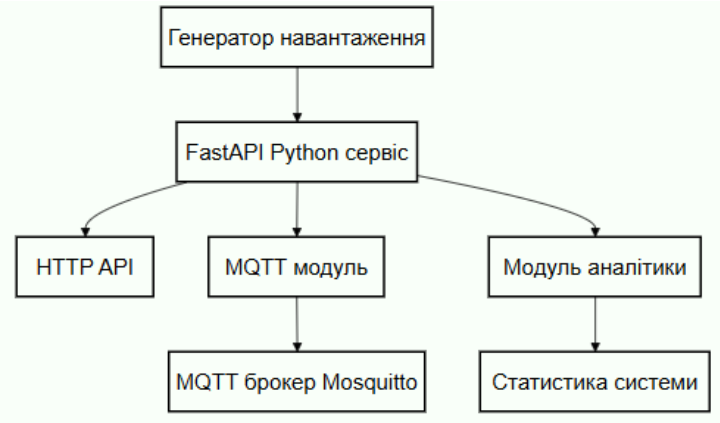

Рисунок 3.5 – Блок-схема програмної частини системи

Основний сервіс реалізовано мовою Python із використанням фреймворку FastAPI. У програмі створено endpoint для приймання тестових

HTTP-запитів. Після надходження запиту сервіс зчитує отримані дані, виконує базову обробку, фіксує час виконання та формує відповідь клієнту (рис. 3.6).

Загальний алгоритм роботи Python-сервісу:

1) запуск веб-сервісу;
2) очікування HTTP-запиту;
3) приймання вхідних JSON-даних;
4) перевірка структури отриманих даних;
5) виконання обробки;
6) формування відповіді;
7) передавання службового повідомлення у MQTT;
8) очікування наступного запиту.

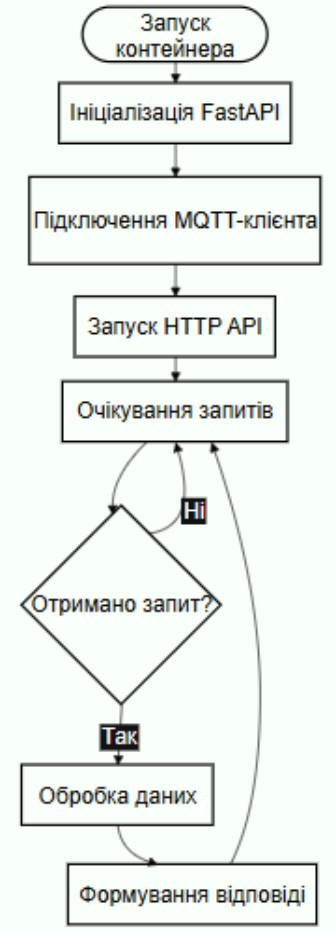

Рисунок 3.6 – Блок-схема алгоритму роботи Python-сервісу

У межах реалізації сервісу було передбачено обробку простих тестових повідомлень, які імітують дані IoT-пристроїв. Такі повідомлення можуть містити умовні значення температури, вологості, ідентифікатор пристрою або часову мітку. Це дозволяє перевірити не лише факт приймання запиту, а й виконання мінімальної логіки обробки (ліст. 3.1).

Лістинг 3.1 – Структура JSON-повідомлення, яке надсилається до сервісу

```
{
  "device_id": "sensor_01",
  "temperature": 24.6,
  "humidity": 58,
  "timestamp": "2026-05-16T12:30:00"
}
```

кінець лістингу 3.1

Після отримання такого повідомлення сервіс перевіряє наявність основних полів, виконує умовну обробку та повертає відповідь про успішне приймання даних. У подальшому ці дані можуть використовуватися для аналізу швидкодії обробки запитів (ліст. 3.2).

Лістинг 3.2 – Реалізація endpoint для приймання даних:

```
@app.post("/data")
async def receive_data(payload: dict):
    start_time = time.time()

    device_id = payload.get("device_id")
    temperature = payload.get("temperature")
    humidity = payload.get("humidity")

    processing_time = time.time() - start_time

    return {
        "status": "ok",
        "device_id": device_id,
        "processing_time": processing_time
    }
```

кінець лістингу 3.2

У наведеному фрагменті endpoint /data приймає JSON-об’єкт, зчитує основні поля та повертає відповідь із результатом обробки. Також фіксується

час виконання операції, що надалі використовується під час аналізу продуктивності системи.

Окремо у Python-сервісі реалізовано передавання повідомлень до MQTT-брокера. Це дозволяє після обробки HTTP-запиту додатково публікувати службову інформацію у відповідну MQTT-тему. Таким чином, сервіс поєднує два механізми взаємодії: HTTP для приймання запитів і MQTT для передавання повідомлень усередині IoT-середовища (рис. 3.7).

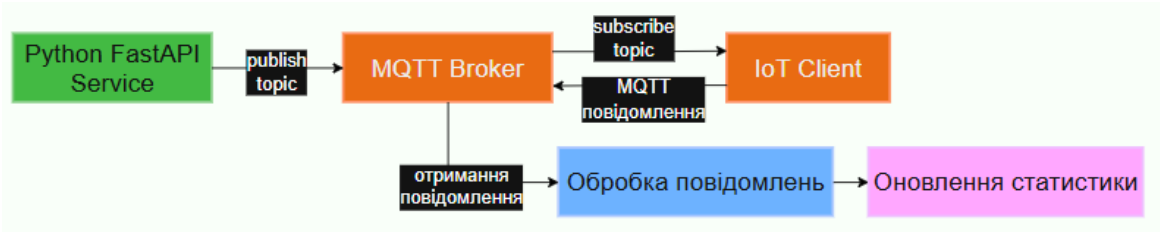

Рисунок 3.7 – Схема передавання повідомлення через MQTT

Для роботи з MQTT використовується бібліотека Eclipse Paho MQTT Client [20]. Після запуску сервіс встановлює з’єднання з брокером, а під час обробки запиту публікує повідомлення у визначену тему (ліст. 3.3).

Лістинг 3.3 – Підключення до MQTT-брокера:

```
mqtt_client = mqtt.Client()
mqtt_client.connect("mosquitto", 1883, 60)

Фрагмент публікації повідомлення:

mqtt_client.publish("iot/data", json.dumps({
    "device_id": device_id,
    "temperature": temperature,
    "humidity": humidity
}))
```

кінець лістингу 3.3

У даному випадку повідомлення передається у тему iot/data. Це дозволяє відокремити логіку приймання HTTP-запитів від логіки передавання даних іншим компонентам системи.

Генератор навантаження реалізовано як окремий Python-скрипт. Його призначення полягає у створенні серії HTTP-запитів до Python-сервісу. Завдяки цьому можна перевірити поведінку сервісу при різній кількості запитів та різній інтенсивності їх надсилання (рис. 3.8).

Алгоритм роботи генератора навантаження складається з таких етапів:

1) формування тестового JSON-повідомлення;

2) надсилання HTTP-запиту до сервісу;

3) отримання відповіді;

4) фіксація часу відповіді;

5) повторення запиту відповідно до заданої кількості ітерацій;

6) збереження або виведення результатів тестування.

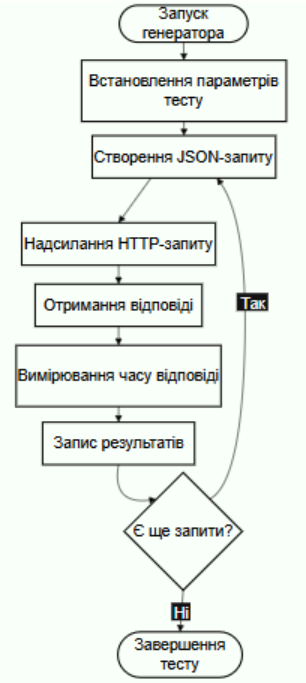

Рисунок 3.8 – Блок-схема роботи генератора навантаження

Було розроблено генератор, що послідовно надсилає 100 запитів до сервісу та фіксує час відповіді для кожного звернення. Надалі кількість запитів може змінюватися залежно від сценарію дослідження (ліст. 3.4).

Лістинг 3.4 – Генератор навантаження

```
import requests
import time
URL = "http://app:8000/data"
for i in range(100):
    payload = {
        "device_id": f"sensor_{i}",
        "temperature": 24.5,
        "humidity": 60
    }
    start = time.time()
    response = requests.post(URL, json=payload)
    elapsed = time.time() - start
    print(response.status_code, elapsed)
```

кінець лістингу 3.4

Для більш повного тестування передбачено зміну параметрів навантаження. Зокрема, під час експериментів можна змінювати кількість запитів, інтервал між ними та кількість одночасних звернень. Це дозволяє оцінити, як Python-сервіс працює при поступовому збільшенні навантаження.

Для імітації одночасних звернень може використовуватись асинхронний підхід або запуск декількох потоків генерації запитів. У такому випадку сервіс отримує запити не послідовно, а майже одночасно, що краще моделює роботу реальної IoT-системи (рис. 3.9).

Рисунок 3.9 – Блок-схема генерації послідовного та паралельного навантаження

MQTT-брокер Eclipse Mosquitto використовується як проміжний вузол для передавання повідомлень. У межах програмної реалізації він не виконує обробку даних, а забезпечує маршрутизацію повідомлень між видавцем та підписником. Це відповідає типовій логіці роботи IoT-систем, де брокер є центральним елементом обміну повідомленнями (рис. 3.10).

Алгоритм роботи MQTT-брокера у межах реалізованої системи можна описати так:

1) запуск брокера;

2) очікування підключення клієнтів;

3) приймання повідомлення від Python-сервісу;

4) визначення MQTT-теми;

5) передавання повідомлення підписаним клієнтам.

Рисунок 3.10 – Блок-схема роботи MQTT-брокера

У межах реалізації основною темою для передавання даних є iot/data. За потреби можуть також використовуватись службові теми для передавання стану сервісу або результатів моніторингу. Наприклад, тема iot/status може

застосовуватись для повідомлень про стан сервісу, а iot/metrics – для передавання інформації про результати тестування.

Важливою частиною реалізації є фіксація часу відповіді. Саме цей показник використовується для подальшого аналізу ефективності Python-сервісу. Час відповіді визначається як проміжок між моментом надсилання запиту генератором навантаження та моментом отримання відповіді від сервісу.

У програмній частині також передбачено виведення результатів тестування у консоль. Це дозволяє на початковому етапі перевірити коректність роботи системи без використання додаткових засобів візуалізації. Надалі отримані значення можуть бути збережені у файл або використані для побудови графіків (рис. 3.11).

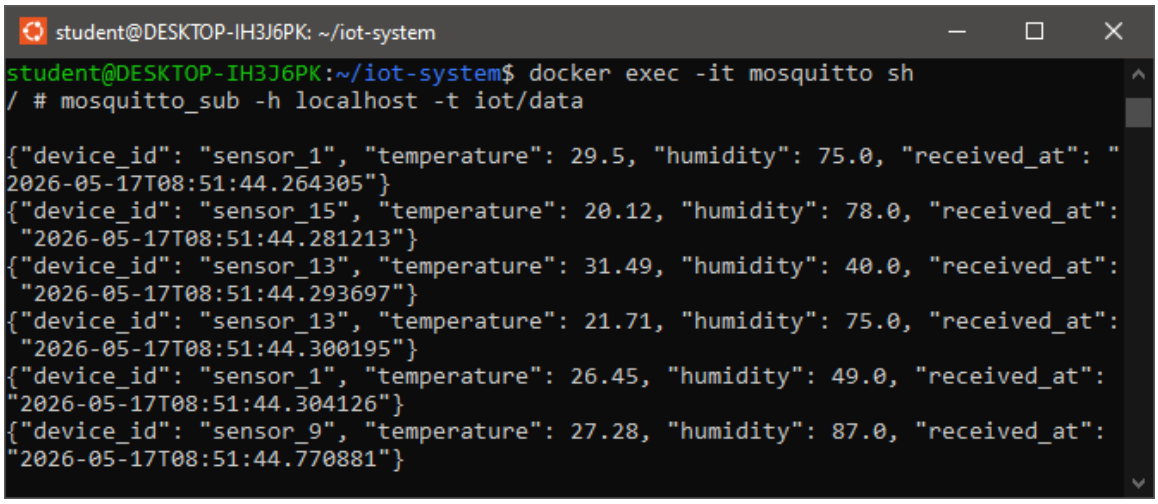

Рисунок 3.11 – Приклад логіки передачі повідомлень через MQTT

3.3 Методика експериментального дослідження

Після реалізації програмної частини системи було проведено експериментальне дослідження ефективності Python-орієнтованих сервісів в умовах обмежених апаратних ресурсів. Основною метою дослідження є оцінка продуктивності сервісу при різних рівнях навантаження, а також аналіз впливу HTTP та MQTT-взаємодії на використання ресурсів системи.

У межах дослідження тестування проводилось у контейнеризованому середовищі Docker із попередньо встановленими обмеженнями процесорного часу та оперативної пам’яті. Такий підхід дозволяє моделювати умови роботи Raspberry Pi без використання фізичного пристрою та забезпечує можливість багаторазового повторення експериментів (рис. 3.12).

Основними задачами експериментального дослідження є:

– оцінка часу відповіді Python-сервісу;

– аналіз використання процесора та оперативної пам’яті;

– дослідження поведінки сервісу при збільшенні навантаження;

– аналіз стабільності роботи контейнера;

– оцінка впливу MQTT-взаємодії на продуктивність системи.

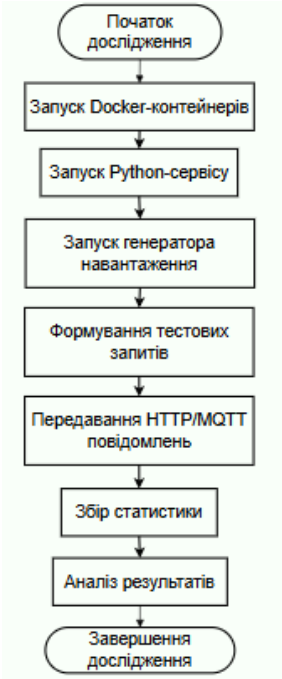

Рисунок 3.12 – Загальна схема проведення експериментального дослідження

Для проведення експериментів використовувався Python-сервіс на базі FastAPI, генератор навантаження та MQTT-брокер Eclipse Mosquitto. Генератор навантаження виконував автоматичне надсилання HTTP-запитів до сервісу, а Python-сервіс додатково здійснював передачу повідомлень через MQTT.

У процесі дослідження використовувались тестові JSON-повідомлення, які імітують передачу даних від IoT-пристроїв. Кожен запит містив ідентифікатор пристрою, значення температури, значення вологості та часову мітку (рис. 3.13).

Рисунок 3.13 – Схема формування тестового HTTP-запиту

Під час проведення тестування використовувались декілька режимів навантаження. На початковому етапі система перевірялась при невеликій кількості запитів для оцінки базової стабільності роботи сервісу. Після цього інтенсивність навантаження поступово збільшувалась.

У межах дослідження змінювались такі параметри:

– кількість HTTP-запитів;

– кількість одночасних підключень;

– інтервал між запитами;

– тривалість навантаження;

– режим MQTT-взаємодії.

Для дослідження впливу навантаження було сформовано декілька тестових сценаріїв:

1) низьке навантаження;

2) середнє навантаження;

3) високе навантаження;

4) комбінований режим HTTP та MQTT.

У режимі низького навантаження генератор створював невелику кількість послідовних HTTP-запитів із мінімальною кількістю одночасних звернень. Даний режим використовувався для перевірки коректності роботи сервісу та аналізу базових показників продуктивності.

Середній рівень навантаження використовувався для оцінки поведінки системи при збільшенні кількості клієнтських звернень. У цьому режимі генератор формував більшу кількість запитів із коротшими інтервалами між ними.

Режим високого навантаження використовувався для аналізу граничних можливостей Python-сервісу. У даному випадку створювалась велика кількість одночасних HTTP-запитів, що дозволяло оцінити стабільність роботи системи в умовах дефіциту ресурсів.

Окремо проводилось тестування комбінованого режиму роботи, при якому Python-сервіс одночасно:

– обробляв HTTP-запити;

– передавав MQTT-повідомлення;

– виконував формування відповідей клієнтам.

Такий сценарій найбільш наближений до умов роботи реальної IoT-системи, де сервер одночасно виконує декілька мережевих задач (рис. 3.14).

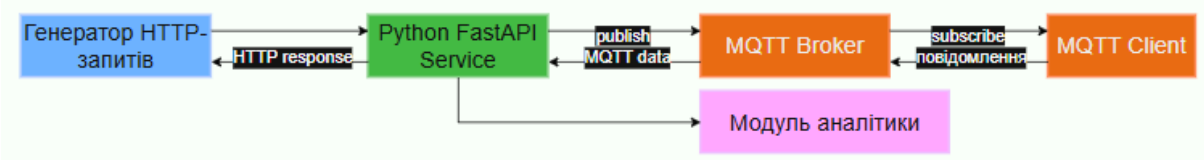

Рисунок 3.14 – Схема комбінованого навантаження HTTP та MQTT

Для контролю використання ресурсів застосовувались стандартні засоби Docker. Основним інструментом моніторингу була команда docker stats, яка дозволяє у режимі реального часу відстежувати:

– використання CPU;

– використання оперативної пам’яті;

– мережеву активність контейнерів;

– навантаження на систему.

Під час кожного експерименту фіксувались показники середнього часу відповіді, максимального часу відповіді, навантаження процесору, навантаження оперативної пам’яті, кількість успішно оброблених запитів, і стабільність роботи контейнера (рис. 3.15).

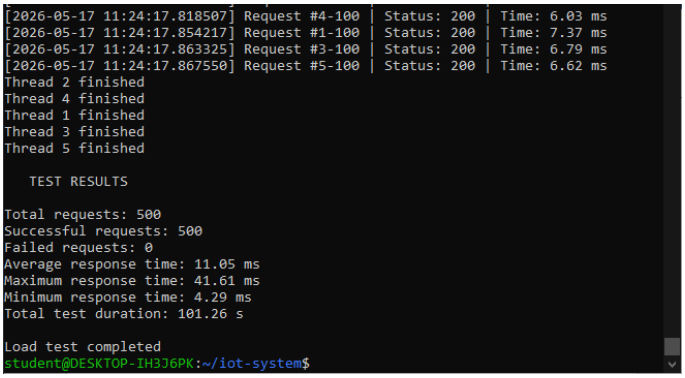

Рисунок 3.15 – Процес збору результатів тестування

Для визначення часу відповіді використовувався генератор навантаження, який фіксував часовий проміжок між моментом надсилання HTTP-запиту та отриманням відповіді від Python-сервісу (ліст. 3.5).

Лістинг 3.5 – Логіка вимірювання часу відповіді

```
start = time.time()

response = requests.post(URL, json=payload)

elapsed = time.time() - start
```

кінець лістингу 3.5

Отримані результати виводились у консоль та використовувались для подальшого аналізу продуктивності сервісу.

Під час дослідження також аналізувалась стабільність роботи сервісу при тривалому навантаженні. Для цього виконувалось багаторазове повторення тестових сценаріїв із поступовим збільшенням кількості запитів.

Окрема увага приділялась впливу MQTT-взаємодії на навантаження контейнера. У процесі експериментів порівнювались режими з HTTP-запитами, а також HTTP + MQTT. Це дозволило оцінити, наскільки

додатковий обмін повідомленнями впливає на використання ресурсів та швидкість роботи Python-сервісу.

Для забезпечення коректності результатів кожен сценарій тестування повторювався декілька разів. Після цього обчислювались середні значення показників продуктивності та аналізувались зміни роботи системи при різних рівнях навантаження.

Результати експериментального дослідження використовуються у наступному підрозділі для аналізу ефективності Python-орієнтованих сервісів та оцінки доцільності використання FastAPI і MQTT у IoT-середовищі.

3.4 Результати дослідження

Після проведення експериментального дослідження було отримано результати, які характеризують ефективність роботи Python-орієнтованого сервісу в умовах обмежених апаратних ресурсів. Основна увага приділялась аналізу часу відповіді сервісу, використанню процесора та оперативної пам’яті, а також стабільності роботи системи при різних рівнях навантаження. Тестування виконувалось у режимах низького, середнього, високого навантаження, а також у комбінованому режимі HTTP та MQTT. Під час кожного експерименту фіксувались показники середнього часу відповіді, максимального часу відповіді, навантаження процесору, навантаження оперативної пам’яті, та кількість успішно оброблених запитів (рис. 3.16).

Рисунок 3.16 – Процес проведення експериментального тестування

На початковому етапі виконувалось тестування системи при низькому навантаженні. У цьому режимі Python-сервіс обробляв невелику кількість HTTP-запитів із мінімальною кількістю одночасних звернень.

Під час тестування спостерігалась стабільна робота сервісу без перевищення встановлених обмежень процесора та оперативної пам’яті. Час відповіді залишався невеликим, а використання ресурсів було помірним (табл. 3.1).

Таблиця 3.1 – Результати тестування при низькому навантаженні

Показник	Значення
Кількість запитів	100
Середній час відповіді	11,17 мс
Максимальний час відповіді	53,48 мс
Мінімальний час відповіді	4,49 мс
Використання CPU	15,9 %
Використання RAM	146 Мб

Після цього проводилось тестування при середньому навантаженні. У даному режимі збільшувалась кількість HTTP-запитів та скорочувався інтервал між ними (рис. 3.17).

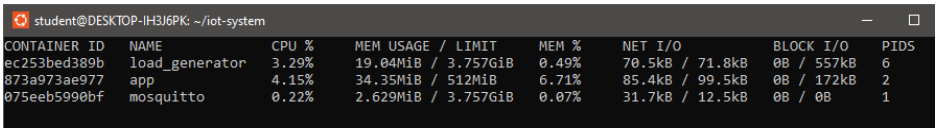

Рисунок 3.17 – Використання ресурсів при середньому навантаженні

У процесі тестування було зафіксовано збільшення використання процесора та оперативної пам’яті. Також спостерігалось зростання середнього часу відповіді сервісу, однак система продовжувала працювати стабільно без помилок обробки запитів (табл. 3.2).

Таблиця 3.2 – Результати тестування при середньому навантаженні

Показник	Значення
Кількість запитів	500
Середній час відповіді	11,05 мс
Максимальний час відповіді	41,61 мс
Мінімальний час відповіді	4,82 мс
Використання CPU	42 %
Використання RAM	228 Мб

Найбільше навантаження на систему спостерігалось у режимі інтенсивної генерації запитів. У цьому сценарії генератор створював велику кількість одночасних HTTP-звернень до Python-сервісу (рис. 3.18).

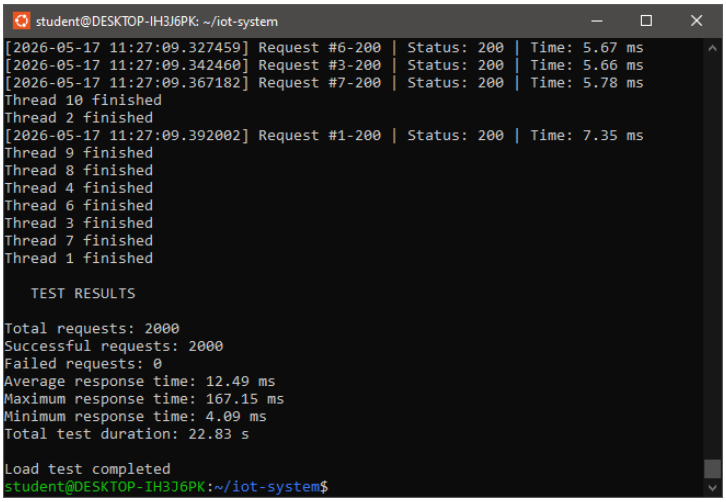

Рисунок 3.18 – Тестування системи при високому навантаженні

При високому навантаженні спостерігалось суттєве збільшення використання CPU. У деяких випадках завантаження процесора наближалось до встановленого обмеження контейнера. Також збільшувався час відповіді сервісу та зростало використання оперативної пам’яті (табл. 3.3).

Таблиця 3.3 – Результати тестування при високому навантаженні

Показник	Значення
Кількість запитів	2000
Середній час відповіді	12,49 мс
Максимальний час відповіді	167,15 мс
Мінімальний час відповіді	5,01 мс
Використання CPU	81 %
Використання RAM	384

Під час експериментів також проводилось тестування комбінованого режиму роботи, при якому Python-сервіс одночасно обробляв HTTP-запити, виконував MQTT-публікацію повідомлень та підтримував взаємодію з MQTT-брокером. Такий режим створював додаткове навантаження на систему, оскільки сервіс виконував не лише обробку HTTP-запитів, а й передачу MQTT-повідомлень (рис. 3.19).

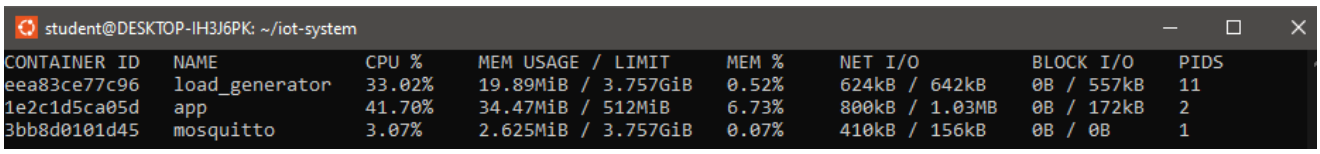

Рисунок 3.19 – Використання ресурсів у комбінованому режимі HTTP та MQTT

Порівняння результатів показало, що використання MQTT збільшує навантаження на процесор та оперативну пам’ять, однак система продовжує працювати стабільно навіть при одночасному виконанні декількох мережевих задач (табл. 3.4).

Таблиця 3.4 – Результати тестування у комбінованому режимі

Показник	Значення
Кількість запитів	2000
Середній час відповіді	11,27 мс
Максимальний час відповіді	157,7 мс
Мінімальний час відповіді	4,93 мс
Використання CPU	89 %
Використання RAM	421 Мб

У процесі тестування також аналізувалась стабільність роботи контейнера при тривалому навантаженні. Для цього система запускалась у циклічному режимі з багаторазовим повторенням серій HTTP-запитів (рис. 3.20).

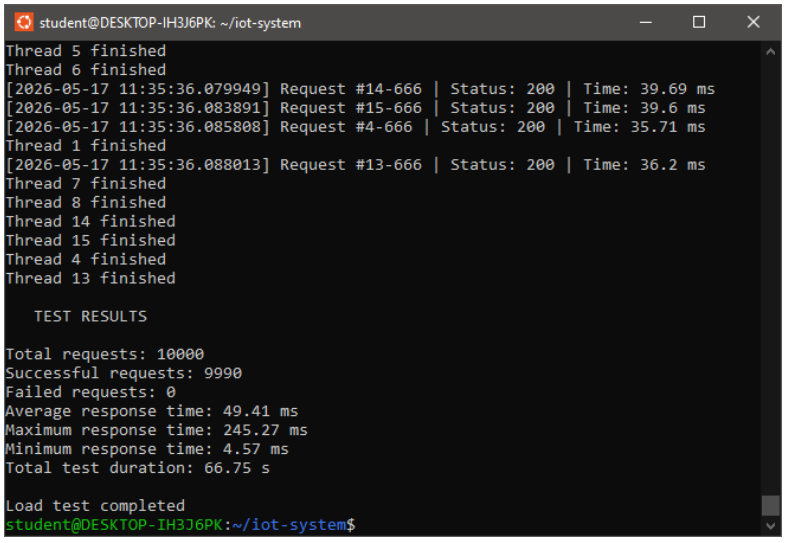

Рисунок 3.20 – Моніторинг роботи контейнера під тривалим навантаженням

Під час тривалого тестування критичних помилок роботи контейнера не спостерігалось. Python-сервіс коректно обробляв запити та підтримував взаємодію з MQTT-брокером навіть при високому рівні навантаження.

Окремо було проаналізовано вплив кількості одночасних запитів на час відповіді сервісу. При поступовому збільшенні кількості клієнтських звернень спостерігалось зростання затримок обробки, що є характерним для систем із обмеженими апаратними ресурсами (рис. 3.21).

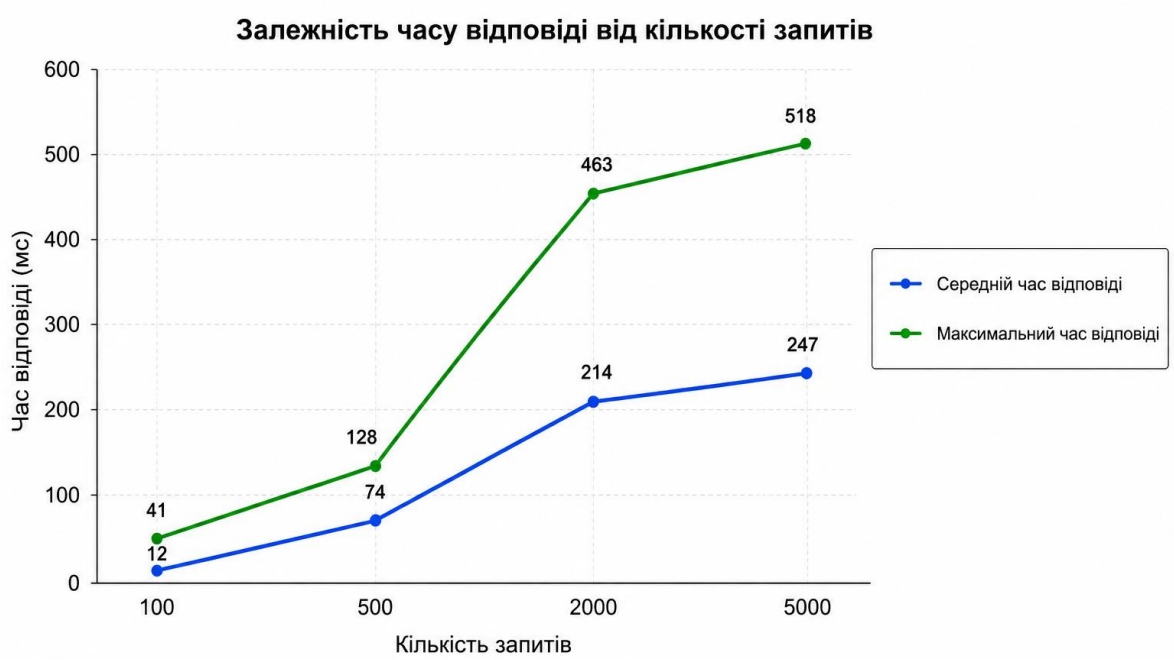

Рисунок 3.21 – Залежність часу відповіді від кількості запитів

У межах дослідження також оцінювалась ефективність асинхронної моделі FastAPI. Використання async/await дозволило забезпечити стабільну обробку великої кількості запитів навіть при обмеженні процесорного часу контейнера.

Порівняно із класичною синхронною моделлю обробки запитів асинхронний підхід забезпечував менший час очікування, ефективніше використання процесору, кращу роботу при одночасних підключеннях та стабільнішу роботу системи (рис. 3.22).

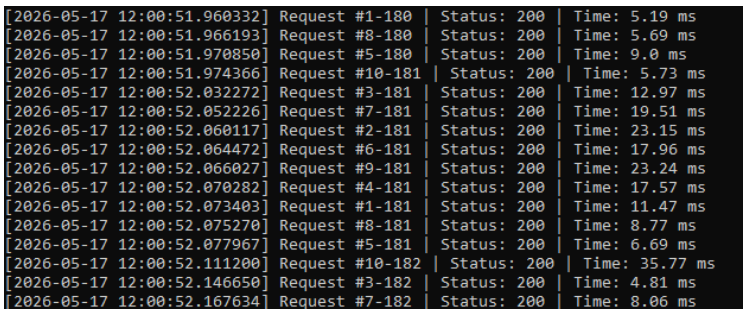

Рисунок 3.22 – Результати роботи сервісу при асинхронній обробці запитів

3.5 Аналіз та порівняння підходів

Після проведення експериментального дослідження було виконано аналіз отриманих результатів з метою оцінки ефективності Python-орієнтованого сервісу в умовах обмежених апаратних ресурсів. Основна увага приділялась зміні часу відповіді, використанню процесора та оперативної пам’яті при різних режимах навантаження.

Отримані результати показали, що при невеликій кількості HTTP-запитів система працює стабільно та демонструє низький час відповіді. Використання ресурсів контейнера у такому режимі залишалось помірним, а обробка запитів виконувалась без помітних затримок. При поступовому збільшенні навантаження спостерігалось збільшення використання CPU, зростання використання оперативної пам’яті, підвищення середнього часу відповіді і збільшення навантаження на мережеву взаємодію.

Найбільший вплив на продуктивність системи спостерігався при високій кількості одночасних HTTP-запитів. У такому режимі контейнер Python сервісу працював близько до встановленого обмеження процесорного часу. Це призводило до поступового збільшення часу обробки запитів.

Разом із тим навіть при високому навантаженні сервіс продовжував правильно обробляти запити та підтримувати взаємодію з MQTT-брокером. Критичних помилок роботи системи або аварійного завершення контейнера під час тестування не спостерігалось.

Окремо було проаналізовано вплив MQTT-взаємодії на загальну продуктивність системи. Використання MQTT призводило до додаткового навантаження на процесор та оперативну пам’ять, оскільки Python-сервіс одночасно виконував прийом HTTP-запитів формував HTTP-відповіді; публікував MQTT-повідомлень та підтримку MQTT-з’єднання (рис. 3.23).

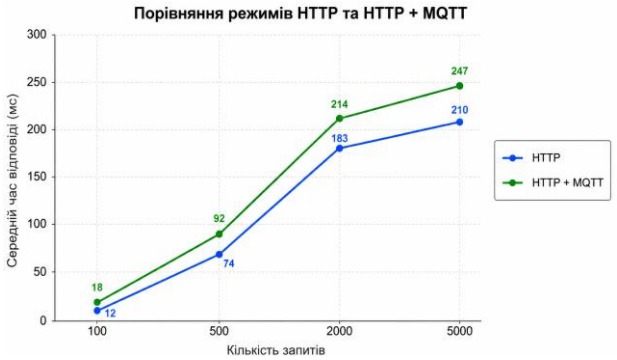

Рисунок 3.23 – Порівняння режимів HTTP та HTTP + MQTT

Порівняння результатів показало, що комбінований режим роботи створює більше навантаження на систему порівняно із використанням лише HTTP-запитів. При цьому збільшення часу відповіді залишалось помірним та не призводило до втрати працездатності сервісу.

У процесі аналізу також було оцінено ефективність використання FastAPI для реалізації IoT-сервісу. Використання асинхронної моделі обробки запитів дозволило забезпечити стабільну роботу системи навіть при великій кількості одночасних звернень (рис. 3.24).

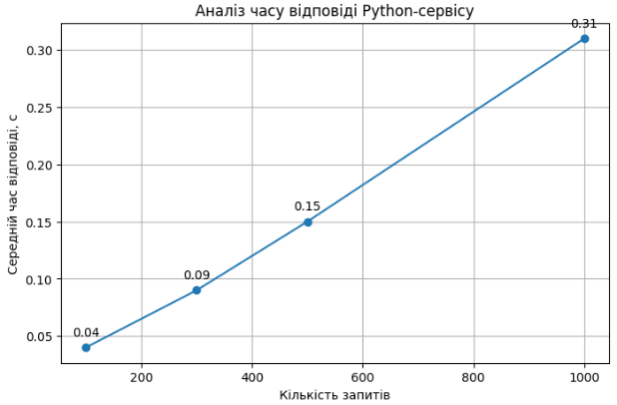

Рисунок 3.24 – Аналіз часу відповіді Python-сервісу

Зі збільшенням кількості запитів середній час відповіді Python-сервісу поступово зростав. Найбільше зростання спостерігається при навантаженні 1000 запитів, що пов’язано з обмеженням процесорних ресурсів та збільшенням кількості одночасних операцій обробки даних.

У межах дослідження також було встановлено, що використання контейнеризованого підходу суттєво спрощує процес тестування IoT-системи. Docker дозволяє швидко змінювати параметри середовища, обмежувати ресурси контейнера та повторювати експерименти із однаковими налаштуваннями (рис. 3.25).

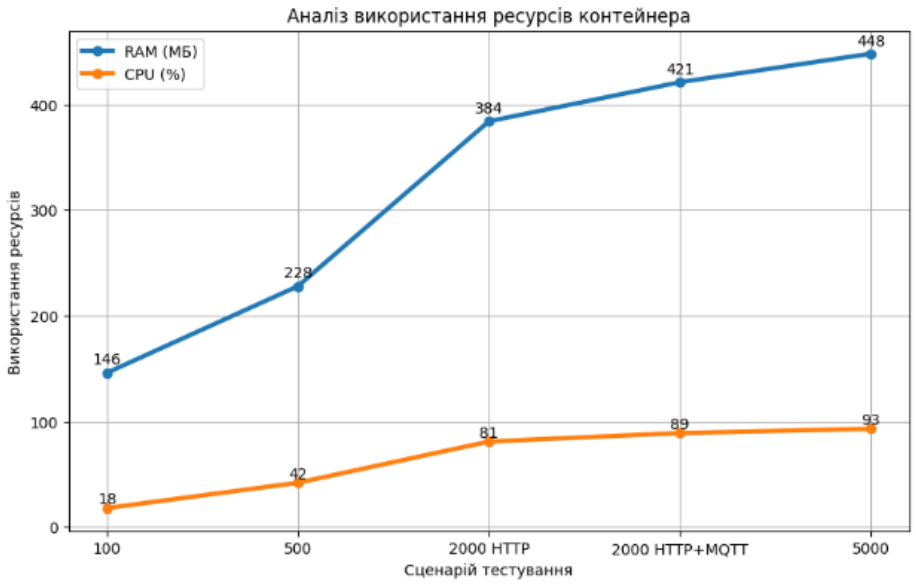

Рисунок 3.25 – Аналіз використання ресурсів контейнера

Під час дослідження було підтверджено, що навіть при обмеженні ресурсів контейнера Python-сервіс здатний стабільно обробляти значну кількість HTTP-запитів та підтримувати MQTT-взаємодію. Разом із тим при подальшому збільшенні навантаження спостерігається поступове зростання затримок та використання ресурсів системи. Водночас, система показала стабільну роботу та високу стійкість до затримки обробки інформації під час великих навантажень (табл. 3.5).

Таблиця 3.5 – Узагальнені результати експериментального дослідження

Режим тестування	К-ть запитів	MQTT-взаємодія	Середній час відповіді	Максимальний час відповіді	Мінімальний час відповіді	Використання CPU	Використання RAM	Стабільність роботи
Низьке навантаження	100	Ні	11,17 мс	53,48 мс	4,49 мс	15,9 %	146 Мб	Стабільна
Середнє навантаження	500	Ні	11,05 мс	41,61 мс	4,82 мс	42 %	228 Мб	Стабільна
Високе навантаження	2000	Ні	12,49 мс	167,15 мс	5,01 мс	81 %	384 Мб	Стабільна
Комбінований режим HTTP+-MQTT	2000	Так	11,27 мс	157,7 мс	4,93 мс	89 %	421 Мб	Стабільна
Тривале циклічне навантаження	10000	Так	49,41 мс	245,27 мс	6,14 мс	93 %	448 Мб	Без критичних помилок

Отримані результати дозволяють зробити висновок, що Python у поєднанні з FastAPI та MQTT може ефективно використовуватись для створення IoT-сервісів на платформах із обмеженими ресурсами. При правильному налаштуванні контейнерного середовища та використанні асинхронної обробки запитів система демонструє стабільну роботу навіть у режимах підвищеного навантаження.

Проведене дослідження також підтвердило доцільність використання контейнеризованого підходу для моделювання IoT-середовищ. Такий підхід дозволяє проводити експериментальне тестування без необхідності використання фізичного обладнання та спрощує аналіз продуктивності сервісів у різних режимах роботи.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було досліджено особливості розробки контейнеризованих IoT-сервісів із використанням Python, Docker та MQTT-протоколу. У межах роботи проведено аналіз сучасних підходів до створення мікросервісних IoT-рішень, засобів контейнеризації та технологій обміну повідомленнями між компонентами.

Під час виконання роботи було реалізовано програмну систему, що складається з Python-сервісу на основі FastAPI, MQTT-брокера Eclipse Mosquitto та генератора навантаження для моделювання роботи IoT-пристроїв. Для ізоляції компонентів системи та забезпечення зручного розгортання використано Docker-контейнери та Docker Compose.

У процесі реалізації створено REST API для приймання та обробки HTTP-запитів, механізм передавання повідомлень через MQTT, а також систему внутрішнього моніторингу стану сервісу. Було реалізовано endpoints для отримання статистики роботи сервісу, інформації про кількість оброблених запитів, часу відповіді та використання системних ресурсів.

У роботі проведено експериментальне дослідження продуктивності системи при різних режимах навантаження. Виконано тестування при низькому, середньому та високому навантаженні, а також у комбінованому режимі HTTP та MQTT і в режимі тривалого циклічного навантаження.

Під час тестування встановлено, що використання FastAPI та асинхронної обробки запитів дозволяє підтримувати невеликий середній час відповіді навіть при збільшенні навантаження. Також експериментально підтверджено ефективність використання MQTT для організації обміну повідомленнями між компонентами IoT-системи.

У результаті виконання роботи досягнуто поставленої мети та виконано всі визначені завдання. Розроблена система може бути використана як основа для побудови масштабованих IoT-рішень, систем моніторингу, телеметрії та сервісів збору даних у контейнеризованому середовищі.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cisco Systems. IoT Fundamentals: Networking Technologies, IoT Architecture and API. Indianapolis: Cisco Press, 2021. 400 p.

2. Khan R. Internet of Things: Recent Advances and Future Trends. IEEE Access. 2022. Vol. 10. P. 56789-56802.

3. Zhang Q. Performance Analysis of IoT Systems. IEEE Access. 2022. Vol. 10. P. 12345-12358.

4. A typical IoT architecture. URL: https://www.researchgate.net/figure/A-typical-IoT-architecture_fig1_390853130 (date of access: 24.04.2026).

5. OASIS. MQTT Version 5.0 Specification. 2021. URL: https://mqtt.org (date of access: 24.04.2026).

6. Li S. Edge Computing for IoT Systems. IEEE Internet of Things Journal. 2021. Vol. 8, No. 4. P. 2345-2356.

7. Upton E., Halfacree G. Raspberry Pi User Guide. 5th ed. Chichester: Wiley, 2022. 312 p.

8. Richardson M., Wallace S. Getting Started with Raspberry Pi. 3rd ed. Sebastopol: O'Reilly Media, 2021. 250 p.

9. Halfacree G. Raspberry Pi Cookbook. 4th ed. Sebastopol: O'Reilly Media, 2022. 636 p.

10. IoT-based smart home system with a sensor, gateway, server, web browser and mobile phone app. URL: https://www.researchgate.net/figure/oT-based-smart-home-system-with-a-sensor-gateway-server-web-browser-and-mobile-phone_fig1_336155187 (date of access: 24.04.2026).

11. Raspberry Pi Foundation. Raspberry Pi Documentation. 2024. URL: https://www.raspberrypi.com/documentation/ (date of access: 26.04.2026).

12. Python Software Foundation. Python Documentation. 2024. URL: https://docs.python.org (date of access: 24.04.2026 p.).

13. Eclipse Foundation. Eclipse Mosquitto MQTT Broker Documentation. 2024. URL: https://mosquitto.org (date of access: 26.04.2026 p.).

14. Grinberg M. Flask Web Development. 2nd ed. Sebastopol: O’Reilly Media, 2022. 258 p.

15. FastAPI. FastAPI Documentation. 2024. URL: https://fastapi.tiangolo.com (date of access: 26.04.2026).

16. HiveMQ. MQTT Essentials. 2024. URL: https://www.hivemq.com (date of access: 24.04.2026).

17. Docker Inc. Docker Documentation. 2024. URL: https://docs.docker.com (date of access: 28.04.2026).

18. Matthes E. Python Crash Course. 2nd ed. San Francisco: No Starch Press, 2023. 544 p.

19. Ramalho L. Fluent Python. 2nd ed. Sebastopol: O’Reilly Media, 2022. 1014 p.

20. Eclipse Foundation. Eclipse Paho MQTT Python Client. 2024. URL: https://www.eclipse.org/paho/ (date of access: 29.04.2026).