Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

Програмно-технічний засіб автоматизації підготовки, тестування та розгортання образів Raspberry Pi із використанням Docker та CI/CD.
Назва теми

Рівень вищої освіти *перший (бакалаврський)*

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія та програмування»*
Назва

Шифр КвРКІ 22051.22.03.17 ПЗ

Виконав здобувач IV курсу, група КІ2-22-3 _____ Сергій ЛІТНАРОВИЧ
Підпис Ініціали, прізвище

Керівник _____ Сергій ЛИСЕНКО
Науковий ступінь, учене звання Підпис Ініціали, прізвище

Нормоконтролер _____ Сергій ЛИСЕНКО
Науковий ступінь, учене звання Підпис Ініціали, прізвище

До захисту допускаю:
завідувач кафедри КІІС _____ Ольга ПАВЛОВА
«01» червня 2026 р. підпис Ініціали, прізвище
дата

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

Рівень вищої освіти ПЕРШИЙ (БАКАЛАВРСЬКИЙ)

Галузь знань 12 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

Спеціальність 123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Освітня програма «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ ТА ПРОГРАМУВАННЯ»

ЗАТВЕРДЖУЮ
Завідувачка кафедри КІІС
_____ Ольга ПАВЛОВА

" 10 " 01 2026 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Літнаровичу Сергію Сергійовичу
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Програмно-технічний засіб автоматизації підготовки, тестування та розгортання образів Raspberry Pi із використанням Docker та CI/CD
Керівник проекту (роботи) Лисенко Сергій Миколайович, д.т.н., проф.
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 20.01.2026 р. № 7

2. Термін подання здобувачем роботи на кафедру 01.06.2026 р.

3. Вихідні дані до роботи Завдання на кваліфікаційну роботу

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

Дослідження предметної області та постановка задачі автоматизації підготовки, тестування та розгортання образів Raspberry Pi.

Проєктування програмно-технічного засобу з використанням технологій контейнеризації Docker та конвеєрів CI/CD.

Програмно-апаратна реалізація та тестування розробленого програмно-технічного засобу у віртуалізованому та хмарному середовищах.

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Загальна структурна схема ПТЗ

Схема алгоритму функціонування

Схема апаратних підключень

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання «_10_»_____01_____2026 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	10.01.2026	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.02.2026	виконано
3	Робота над розділом 1 – дослідження предметної області та постановка задачі	01.03.2026	виконано
4	Робота над розділом 2 – проєктування програмно-технічного засобу з використанням технологій контейнеризації Docker та конвеєрів CI/CD	01.04.2026	виконано
5	Робота над розділом 3 – програмно-апаратна реалізація та тестування розробленого програмно-технічного засобу у віртуалізованому та хмарному середовищах	29.04.2026	виконано
6	Оформлення пояснювальної записки згідно вимог	25.05.2026	виконано
7	Попередній захист ВКР	26.05.2026	виконано
8	Захист ВКР на засіданні ЕК	Червень 2026 року	

Здобувач _____ Сергій ЛІТНАРОВИЧ
Підпис Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи _____ Сергій ЛИСЕНКО
Підпис Ім'я, ПРІЗВИЩЕ

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 22051.22.03.17 ПЗ	Пояснювальна записка	61		
			Графічні матеріали			
2		КвРКІ 22051.22.03.17 Е8	Загальна структурна схема ПТЗ	1		
3		КвРКІ 22051.22.03.17 Е8	Схема алгоритму функціонування	1		
4		КвРКІ 22051.22.03.17 Е8	Схема апаратних підключень	1		

Зм	Арк	№ докум	Підпис	Дата	КвРКІ 22051.22.03.17 ВП		
Розробив		Літнарович		01.06	Відомість проекту	Літера	Аркуш	Аркушів
Перевір.		Лисенко				У	1	61
Н. контр.		Лисенко				ХНУ, КІ2-22-3		
Затв.		Павлова		01.06				

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Програмно-технічний засіб автоматизації підготовки, тестування та розгортання образів Raspberry Pi із використанням Docker та CI/CD».

Автор роботи: Сергій ЛІТНАРОВИЧ.

Керівник роботи: Сергій ЛИСЕНКО.

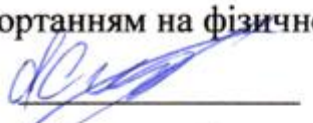
Пояснювальна записка: 61 с., 12 рис., 4 табл., 4 дод., 50 джерел.

Графічна частина: 3 креслення.

АВТОМАТИЗАЦІЯ, БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ ТА РОЗГОРТАННЯ, ЕМУЛЯЦІЯ, КОНТЕЙНЕРИЗАЦІЯ, МОНІТОРИНГ, МУЛЬТИ-АРХІТЕКТУРНА ЗБІРКА, CI/CD, DOCKER, RASPBERRY PI

Кваліфікаційна робота бакалавра присвячена розробці програмно-технічного засобу для автоматизації підготовки, перевірки та доставки ПЗ для вбудованих систем. Актуальність теми зумовлена потребою переходу від ручного налаштування пристроїв Raspberry Pi до декларативного розгортання інфраструктури, що усуває проблему «дрейфу конфігурацій» та залежність від фізичного обладнання під час розробки.

Метою роботи є створення CI/CD конвеєра для автоматичної збірки, тестування та публікації ПЗ цільової архітектури у хмарному середовищі. Для досягнення мети проаналізовано підходи до контейнеризації, розроблено архітектуру системи на базі GitHub Actions та застосовано апаратну крос-емуляцію. У результаті реалізовано мульти-архітектурну збірку Docker-образів із інтегрованим стеком моніторингу (Prometheus, Grafana) та налаштовано повний цикл безперервної інтеграції. Працездатність рішення підтверджено успішним автоматизованим тестуванням у віртуалізованому середовищі та швидким розгортанням на фізичному мікрокомп'ютері Raspberry Pi.



Підпис здобувача

30.05.2026

Дата

ЗМІСТ

ВСТУП

Актуальність дослідження. Розвиток Інтернету речей та розподілених обчислювальних систем зумовив широке використання одноплатних комп'ютерів на базі ARM-архітектури, зокрема платформ Raspberry Pi на основі систем-на-кристалі Broadcom. Специфіка завантаження таких мікрокомп'ютерів суттєво відрізняється від класичної x86-архітектури, оскільки вимагає точних маніпуляцій з таблицями розділів файлових систем (ext4, FAT32) та специфічним завантажувачем графічного процесора. Ручне розгортання образів на фізичних носіях (SD-картах або модулях eMMC) в умовах реального виробництва чи розробки призводить до виникнення помилок конфігурації, відсутності відтворюваності середовища та суттєвого уповільнення виробничого циклу. Відповідно, актуальність теми зумовлена об’єктивною потребою переходу від імперативного ручного налаштування вбудованих пристроїв до декларативного управління інфраструктурою за допомогою сучасних практик автоматизації. Дослідження безпосередньо стосується впровадження сучасних практик DevOps та контейнеризації в галузі Інтернету речей та вбудованих обчислювальних систем.

Метою кваліфікаційної роботи є розроблення програмно-технічного засобу для автоматичної збірки, тестування та розгортання програмного забезпечення під архітектуру Raspberry Pi (ARM) у хмарному середовищі, що дозволяє усунути необхідність використання фізичної плати на етапах розробки та валідації коду.

Для досягнення поставленої мети передбачено виконання таких завдань: проведення аналізу архітектурних особливостей завантаження SoC Broadcom; обґрунтування вибору технологічного стека для крос-платформної збірки; проєктування інфраструктури на базі технологій контейнеризації; реалізація конвеєра безперервної інтеграції та доставки для автоматичного тестування працездатності зібраного образу перед його розгортанням.

Об’єктом дослідження є процес безперервної інтеграції та доставки програмного забезпечення для вбудованих систем.

Предметом дослідження є методи та інструменти контейнеризації, мульти-архітектурної збірки, апаратної крос-емуляції (QEMU, binfmt_misc) та хмарної автоматизації GitHub Actions.

Новизна одержаних результатів полягає в удосконаленні процесу формування та верифікації образів операційних систем для ARM-архітектури шляхом інтеграції механізмів контейнеризації та трансляції інструкцій процесора. Це забезпечує можливість виконання модифікації та тестування образу без розгортання повноцінних віртуальних машин, що суттєво мінімізує витрати обчислювальних ресурсів.

Практичне значення результатів полягає у створенні готового програмно-технічного інструментарію для інженерів, застосування якого скорочує час підготовки IoT-пристроїв, забезпечує високу відтворюваність середовища та виключає вплив людського фактору на стабільність розгортання систем.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної області, її структурні та функціональні особливості

Фундаментальною основою сучасних периферійних обчислювальних вузлів є високоінтегровані системи на кристалі. Аналіз предметної області доцільно розпочати з дослідження архітектурних особливостей лінійки мікрокомп'ютерів Raspberry Pi які виступають базовою апаратною платформою у даній роботі. На відміну від класичних серверних рішень x86_64, апаратна база цих пристроїв орієнтована на архітектуру ARM, що вимагає специфічних інженерних підходів до формування операційного середовища та конфігурування завантажувачів [1, 2].

Важливим аспектом системного аналізу архітектури ARMv8-A є дослідження моделі рівнів винятків, оскільки саме вона визначає логіку ізоляції Docker-контейнерів на апаратному рівні. В архітектурі реалізовано чотири ієрархічні рівні привілеїв: від EL0 до EL3 (де EL3 це рівень монітора безпеки). Програмно-технічний засіб, що розробляється, функціонує переважно на рівнях EL0 та EL1. Використання контейнеризації дозволяє запускати ПЗ моніторингу у просторі користувача EL0, тоді як ядро операційної системи, що забезпечує роботу Docker-демона, функціонує на рівні EL1. Така архітектурна декомпозиція гарантує, що автоматизоване розгортання прикладних образів не зможе дестабілізувати роботу ключових системних служб або змінити стан гіпервізора EL2, що є визначальним для безпеки IoT-інфраструктур. Розуміння механізму перемикання контексту між цими рівнями дозволяє оптимізувати системні виклики у Python-скриптах моніторингу, мінімізуючи накладні витрати на обробку переривань при зчитуванні даних з апаратних лічильників продуктивності [3, 8, 15].

Архітектура ARMv8-A, що лежить в основі моделей Raspberry Pi 3, 4 та 5, є першою в лінійці ARM, яка запровадила 64-бітний стан виконання інструкцій

AArch64. Це дозволило значно розширити адресний простір оперативної пам'яті та підвищити ефективність математичних обчислень за рахунок використання 31 регістра загального призначення шириною 64 біти. Для вбудованих систем, що виконують завдання моніторингу та первинної обробки даних, перехід до 64-бітної архітектури став визначальним фактором, оскільки це дозволяє використовувати сучасні методи контейнеризації [3, 4].

Основою апаратної реалізації Raspberry Pi є системи на кристалі від компанії Broadcom. У моделі Raspberry Pi 3 використовується чип BCM2837, тоді як у Raspberry Pi 4 встановлено потужніший BCM2711. Архітектурно ці SoC складаються з центрального багатоядерного процесора ARM Cortex та графічного співпроцесора VideoCore. Специфічною особливістю Broadcom SoC є те, що графічне ядро VideoCore виступає головним контролером системи під час ініціалізації [5, 6]. Порівняльні характеристики апаратних платформ, що враховувалися при розробці програмно-технічного засобу, наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика цільових апаратних платформ Raspberry Pi

Характеристика	Raspberry Pi 3 Model B/B+	Raspberry Pi 4 Model B	Raspberry Pi 5
Система на кристалі (SoC)	Broadcom BCM2837B0	Broadcom BCM2711	Broadcom BCM2712
Архітектура CPU	4 ядра Cortex-A53	4 ядра Cortex-A72	4 ядра Cortex-A76
Розрядність ОС	32 / 64 біти	32 / 64 біти	64 біти
Тип та обсяг RAM	1 ГБ LPDDR2	1 - 8 ГБ LPDDR4	4 - 8 ГБ LPDDR4X
Мережеві інтерфейси	Fast Ethernet, WiFi 2.4	Gigabit Ethernet, WiFi 2.4/5	Gigabit, WiFi 2.4/5, PCIe
Основний носій	MicroSD (UHS-I)	MicroSD, USB 3.0	MicroSD, USB 3.0, NVMe

Для детальнішого розуміння внутрішньої організації цільових систем та специфіки взаємодії обчислювальних ядер із периферійними контролерами, доцільно розглянути структурну організацію компонентів SoC. На рисунку 1.1 представлено узагальнену архітектуру системи на кристалі Broadcom, що демонструє тісний зв'язок між ядрами ARM Cortex та графічним співпроцесором VideoCore, який відіграє ключову роль у процесах ініціалізації заліза.

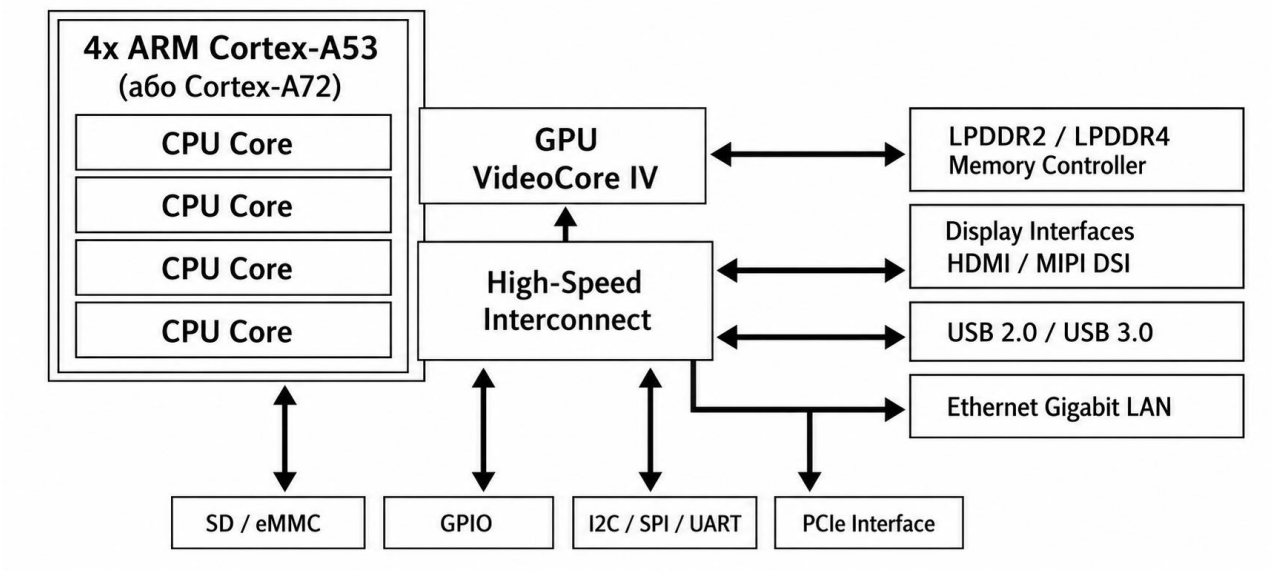


Рисунок 1.1 – Узагальнена структурна схема системи на кристалі Broadcom [23]

Для забезпечення високого рівня автоматизації підготовки образів необхідно враховувати, що процес початкового завантаження Raspberry Pi суттєво відрізняється від традиційних обчислювальних систем, що базуються на стандартах BIOS або UEFI. Специфіка процедури ініціалізації апаратного забезпечення створює низку технологічних перешкод, які зумовлені архітектурною роллю графічного співпроцесора VideoCore.

На первинній стадії завантаження, що розпочинається одразу після подачі живлення, центральні ядра архітектури ARM перебувають у стані апаратного скидання. У цей момент управління передається ядру відеопроцесора, яке зчитує початковий мікрокод із вбудованої пам'яті BootROM. Цей мікрокод є статичним

компонентом системи на кристалі, що реалізує базову логіку пошуку завантажувачів на зовнішніх носіях інформації [1, 6].

Продовження ініціалізації пов'язане з пошуком файлу другого рівня на першому розділі накопичувача, що зазвичай представлений файлом bootcode.bin. Основним завданням цього компонента є ініціалізація контролера оперативної пам'яті та підготовка системних шин до завантаження основної прошивки. Після успішного налаштування пам'яті GPU виконує завантаження файлу start.elf, який фактично є операційним середовищем відеопроцесора. Саме на цьому етапі відбувається обов'язкове зчитування дескриптора апаратної конфігурації config.txt [5, 7]. Цей файл дозволяє інженеру реалізувати декларативний підхід до управління залізом, встановлюючи параметри тактової частоти, активуючи периферійні інтерфейси та визначаючи межі адресного простору між обчислювальними ядрами.

Завершальна стадія завантаження наступає лише після повної підготовки апаратної інфраструктури, коли графічний процесор знімає сигнал скидання з ядер ARM. Це дозволяє розпочати виконання двійкового коду ядра Linux, представленого файлом kernel8.img. Розуміння цієї послідовності є основоположним при проєктуванні програмно-технічного засобу, оскільки будь-яке порушення структури файлової системи або цілісності бінарних компонентів на етапі автоматизованої підготовки призведе до неможливості ініціалізації цільового пристрою.

Така специфічна організація файлової системи створює суттєві технологічні бар'єри при масовому тиражуванні та модифікації образів ОС [8]. Образ диска для вбудованої системи є побітовим зліпком усього блокового пристрою. У середовищах розробки на базі Windows або macOS прямий доступ до розділу ext4 є ускладненим, що перешкоджає оперативному редагуванню конфігураційних файлів без фізичного розгортання образу на носій. Крім того, виникає проблема управління ідентифікаторами блокових пристроїв. Під час роботи на Raspberry Pi SD-карта визначається системою як /dev/mmcblk0, проте

при монтуванні цього ж образу на робочій станції розробника пристрій може ідентифікуватися як /dev/sdb [6]. Жорстке прив'язування конфігурацій до статичних імен пристроїв призводить до збоїв завантаження. Вирішенням цієї проблеми є використання універсальних унікальних ідентифікаторів розділів PARTUUID.

В умовах промислової експлуатації гострою проблемою стає явище «дрейфу конфігурацій» [9]. Ручне налаштування кожного окремого пристрою, встановлення залежностей, маніпуляції з конфігураційними файлами, налаштування сервісів неминуче призводить до розбіжностей у стані програмного середовища на різних вузлах. Це викликає ситуації, коли програмне оновлення, успішно валідоване на тестовому стенді, призводить до серйозних збоїв у виробничому середовищі. Традиційний метод вирішення цієї проблеми шляхом побітового клонування відлагоджених SD-карт є неефективним та небезпечним, оскільки разом із корисними даними копіюється системне сміття та унікальні криптографічні ключі SSH, що є грубим порушенням політик інформаційної безпеки [10, 11].

Додатковим фактором, що ускладнює експлуатацію вбудованих систем, є апаратний знос флеш-пам'яті [2]. На відміну від твердотільних накопичувачів, SD-карти не оснащені складними контролерами для апаратного вирівнювання зносу [6]. Активне журналювання файлової системи в стандартних конфігураціях Linux швидко деградує носій. Створення відмовостійкої системи вимагає внесення низки системних змін (монтування директорій в оперативну пам'ять, вимкнення файлу підкачки), виконання яких вручну для кожного релізу є трудомістким процесом із високим ризиком внесення помилок.

Аналіз апаратних та структурних особливостей платформи Raspberry Pi дозволяє зробити висновок, що традиційні підходи до адміністрування серверних ОС є неефективними для вбудованих систем [12]. Існує об'єктивна необхідність у переході до практик автоматизації розгортання [13, 14]. Застосування принципів безперервної інтеграції та доставки, зокрема

забезпечення версійності, збірки з чистого стану та перевірки у віртуалізованому середовищі дозволить нівелювати вплив апаратних обмежень та забезпечити високу стабільність кінцевих програмно-технічних рішень [8, 15].

Специфіка розроблення для одноплатних систем Raspberry Pi вимагає врахування парадигми сумісного проєктування апаратного та програмного забезпечення. Оскільки ресурси SoC Broadcom є фіксованими, оптимізація системного ПЗ має відбуватися на рівні взаємодії з контролерами прямого доступу до пам’яті. У контексті автоматизації розгортання це означає, що засіб повинен готувати образ таким чином, щоб мінімізувати кількість перемикань контексту процесора при зборі телеметрії. Аналіз апаратної архітектури ARM Cortex-A72 (у RPi 4) та Cortex-A53 (у RPi 3) виявив суттєві відмінності у роботі конвеєрів інструкцій та передбачувачів переходів, що безпосередньо впливає на швидкість виконання контейнеризованих додатків. Таким чином, спроєктований ПТЗ має враховувати не лише розрядність архітектури, а й специфічні набори розширень, такі як NEON для векторних обчислень та апаратне прискорення криптографічних операцій, що інтегруються у Docker-образ через специфічні флаги компіляції. Це дозволяє досягти максимальної продуктивності моніторингового стеку при мінімальному тепловиділенні кристала SoC [3, 5, 15, 31, 48].

1.2 Аналіз наявного програмно-технічного забезпечення предметної області

Для розв’язання завдань, пов’язаних із підготовкою системного програмного забезпечення для вбудованих рішень на базі архітектури ARM, застосовується розгалужений інструментальний апарат, який доцільно класифікувати за рівнями абстракції та ступенем автоматизації процесів. Вибір оптимальної стратегії розробки зумовлюється обґрунтованими вимогами до відтворюваності середовища, інтенсивності оновлень та специфіки

функціонування системи [11, 16, 17]. В інженерній практиці зазвичай виокремлюють три парадигмальні підходи: формування еталонного «золотого образу», експлуатація систем автоматизованої генерації дистрибутивів та використання засобів скриптової автоматизації у поєднанні з технологіями віртуалізації.

Найбільш поширеним, проте найменш відповідним парадигмі сучасних DevOps-практик є метод ручного створення еталонного образу. Даний підхід базується на ініціалізації стандартного дистрибутива на фізичному носії з подальшим інтерактивним конфігуруванням системи через термінальний доступ [1]. Процес налаштування охоплює інсталяцію програмних компонентів за допомогою пакетного менеджера, редагування конфігураційних дескрипторів, а також розгортання прикладного ПЗ у реальному часі. Кінцевим етапом є вилучення носія та генерація файлу-образу шляхом побітового копіювання за допомогою утиліти dd. Такий метод супроводжується низкою критичних недоліків, оскільки результуючий образ стає репозиторієм надлишкових артефактів: залишків кешу пакетних менеджерів, журналів командної оболонки та тимчасових об'єктів. Найбільш вагомим з точки зору безпеки є копіювання статичних SSH-ключів хоста й унікальних ідентифікаторів machine-id, що при масштабуванні на групу пристроїв призводить до конфліктів ідентифікаторів та створює вразливості в інфраструктурі захисту інформації [10, 11, 18]. Крім того, обсяг сформованого образу лінійно відповідає повній місткості вихідної SD-карти, що потребує залучення додаткового інструментарію для реструктуризації таблиць розділів та обнулення вільного простору. Проте найбільш деструктивним фактором залишається відсутність методологічної відтворюваності, що неминуче спричиняє похибки у фінальних збірках при деградації вихідного носія [9, 19].

Інша методологічна група базується на використанні комплексних систем генерації дистрибутивів, таких як Yocto Project або Buildroot. Ці інструментальні засоби дозволяють формувати операційне середовище з чистого стану,

здійснюючи нативну компіляцію ядра та системних бібліотек безпосередньо з вихідних кодів. Декларативний характер опису конфігурації забезпечує створення мінімалістичного образу, що пройшов глибоку оптимізацію під конкретну апаратну специфікацію [4, 20]. Попри технологічну досконаліть, інтеграція таких систем у виробничі цикли часто виявляється економічно та операційно нераціональною через високий поріг входження та значні часові витрати на повний цикл збирання образу, який може тривати кілька годин. Орієнтованість цих систем на формування статичних прошивок створює додаткові труднощі при роботі з екосистемою Raspberry Pi, яка вимагає залучення високорівневих інструментаріїв та стандартних пакетних баз [8, 12, 21].

Поряд із цим існують методи автоматизації на основі скриптових сценаріїв та механізмів chroot-ізоляції, репрезентативним прикладом яких є інструментарій pi-gen. Технологічний алгоритм тут спирається на розгортання базової файлової системи через утиліти debootstrap, її подальше монтування у середовище хост-машини та виконання процедур налаштування в ізольованому оточенні. Можливість виконання ARM-інструкцій на хостах x86 забезпечується за рахунок динамічної трансляції у режимі QEMU user-mode emulation [8, 9, 22, 29, 30]. Хоча даний підхід дозволяє оперувати стандартними репозиторіями, варіативність версій утиліт на робочих станціях часто призводить до непередбачуваних збоїв. Крім того, маніпуляції з loop-back пристроями вимагають привілеїв суперкористувача, що створює перешкоди для повної контейнеризації процесу.

У цьому контексті розглядаються також інструменти концепції «інфраструктура як код», зокрема HashiCorp Packer. Він дозволяє формалізувати створення образу через дескриптори JSON або HCL, інтегруючись із системами керування конфігураціями (наприклад, Ansible). Проте Packer виступає переважно високорівневим оркестратором, не вирішуючи повною мірою

проблему ізоляції низькорівневих механізмів взаємодії з ядром від середовища хоста [13, 15, 37, 49].

Окремої уваги заслуговує технологічний стек контейнеризації та апаратної емуляції. Центральне місце у ньому посідає Docker - це система ізоляції процесів, яка на відміну від повноцінної апаратної віртуалізації спирається на спільне ядро хостової ОС [13, 23, 24]. Функціонування Docker базується на просторах імен, що забезпечують логічну декомпозицію системних ресурсів, та контрольних групах, які відповідають за лімітування обчислювальних потужностей. Для цілей збирання образів така архітектура є вирішальною, оскільки дозволяє безпечно виконувати маніпуляції з файловою системою без ризику дестабілізації середовища розробника [14, 25, 26, 41].

Стандартна контейнеризація забезпечує лише ізоляцію середовища виконання, не розв'язуючи проблему несумісності процесорних архітектур. Типова ситуація полягає в тому, що CI/CD-сервери функціонують на базі x86_64, тоді як цільові платформи типу Raspberry Pi орієнтовані на ARM. Вирішення цієї задачі забезпечується інструментарієм QEMU у режимі user-mode emulation. На відміну від повносистемної емуляції, цей режим фокусується виключно на динамічній трансляції системних викликів, що значно знижує обчислювальні витрати [7, 8, 45]. Під час звернення до виконуваного файлу ARM ядро ідентифікує його заголовок та переспрямовує виконання відповідному емулятору, що робить можливим прозоре виконання команд у chroot-оточенні образу Raspberry Pi [22, 27, 28].

Важливим аспектом аналізу існуючих засобів є розгляд відмінностей між апаратною віртуалізацією та контейнеризацією на рівні операційної системи. Традиційні гіпервізори Type-1 та Type-2, такі як KVM або VMware, створюють повну абстракцію апаратного забезпечення, що вимагає запуску окремого екземпляра ядра ОС для кожного віртуального вузла. У контексті Raspberry Pi та архітектури ARM такий підхід є неефективним через значні накладні витрати на емуляцію системних шин та контролерів, що може забирати до 30-40%

обчислювальної потужності пристрою. Натомість технологія Docker, що досліджується у даній роботі, використовує спільне ядро хостової системи, що дозволяє досягти продуктивності, близької до нативної. Проведене порівняння архітектурних підходів доводить, що для мікрокомп'ютерів з обмеженим обсягом кеш-пам'яті другого рівня та вузькою шиною пам'яті, саме контейнеризація є найбільш раціональним технічним рішенням, оскільки вона дозволяє запускати десятки ізольованих сервісів моніторингу без деградації часу відгуку системи [10, 11, 18, 36, 38].

Для системного аналізу наявних технологічних рішень та обґрунтування вибору подальшого напряму розробки проведено порівняння існуючих підходів за ключовими інженерними критеріями: рівень ізоляції середовища, підтримка крос-емуляції інструкцій та декларативність конфігурації (таблиця 1.2).

Таблиця 1.2 – Порівняльна характеристика засобів підготовки образів ОС для вбудованих систем

Характеристика / Інструмент	Ручне налаштування	Yocto Project / Buildroot	pi-gen (chroot)
Рівень ізоляції середовища збірки	Низький (відсутній)	Високий	Середній
Підтримка крос-апаратної емуляції	Не підтримується	Нативна крос-компіляція	Трансляція QEMU (залежить від хоста)
Декларативність конфігурації	Відсутня (імперативно)	Висока (рецепти)	Середня (скрипти)
Інтеграція з хмарними CI/CD	Неможлива	Ускладнена (через час збірки)	Ускладнена (через root-привілеї)

Кінець таблиці 1.2

Схильність до «дрейфу конфігурацій»	Критично висока	Відсутня	Низька

Результати порівняльного аналізу, наведені у табл. 1.2, демонструють, що жоден із існуючих методів не поєднує у собі одночасно високу швидкість ітерацій та повну ізоляцію інструментарію від хостової системи. Це підтверджує доцільність розроблення гібридного підходу, який буде спроєктований у наступному розділі.

Дослідження існуючих підходів також виявило необхідність врахування концепції управління життєвим циклом ПЗ після первинного розгортання, так званих операцій другого дня. В умовах розподілених мереж Raspberry Pi ручне оновлення кожного вузла є економічно недоцільним. Використання Docker-образів як одиниць дистрибуції дозволяє реалізувати стратегію цілісних оновлень, де заміна версії ПЗ відбувається шляхом перезапуску контейнера з новим шаром даних. Це забезпечує можливість миттєвого відкату до попереднього стабільного стану у разі виявлення помилок під час моніторингу. Такий підхід трансформує архітектуру вбудованої системи в гнучке середовище, здатне до безперервної еволюції без переривання ключових бізнес-процесів, що є фундаментальною перевагою розробленого програмно-технічного засобу порівняно з монолітними образами ОС [10, 12, 22, 27, 42].

1.3 Визначення вимог для ПТЗ

На підставі результатів проведеного аналізу апаратної специфіки платформ Raspberry Pi та детального огляду наявних інструментальних методологій формування образів операційних систем було сформульовано обґрунтовану необхідність у розробці спеціалізованого програмно-технічного засобу (ПТЗ) автоматизації. Центральною метою проєкту є побудова цілісного

та автономного конвеєра, що забезпечує декларативне управління повним життєвим циклом системних артефактів від стадії первинного формування структури розділів до фінальної верифікації та публікації у хмарних реєстрах.

Виходячи з проблематики, описаної у підрозділах 1.1 та 1.2, до розроблюваного ПТЗ висуваються такі функціональні вимоги. Засіб повинен забезпечувати автоматизовану збірку, тестування та доставку контейнерних образів для архітектури ARM. Оскільки базове розгортання операційної системи на мікрокомп'ютерах є одноразовою апаратною процедурою, головний фокус автоматизації у роботі спрямовано на управління життєвим циклом прикладного програмного забезпечення [9, 14].

Обов'язковою є підтримка механізму контейнерної ізоляції з прозорою крос-архітектурною трансляцією інструкцій через рівень QEMU user-mode emulation. Це дозволяє виконувати компіляцію залежностей та налаштування оточення без залучення фізичного обладнання на стадії збирання, що усуває апаратну залежність розробника [12, 22].

Цільовим програмним навантаженням, яке упаковується в автоматизований образ, визначено стек системного моніторингу. Такий вибір зумовлений тим, що сервіси моніторингу вимагають прямої взаємодії з апаратними інтерфейсами ядра (статистика CPU, RAM, температурні датчики).

Це робить їх ідеальним еталонним додатком для верифікації працездатності CI/CD конвеєра та перевірки відсутності помилок архітектурної несумісності перед фінальним розгортанням на платі [15, 23].

До нефункціональних вимог належить забезпечення повної методологічної відтворюваності результатів: за ідентичних вхідних параметрів і конфігураційних файлів система повинна генерувати побітово однакові артефакти незалежно від архітектури хоста розробника. Вимога переносимості передбачає функціонування конвеєра у будь-якому середовищі, де доступна технологія Docker, без необхідності інсталяції додаткових системних залежностей на хостовій ОС розробника [13, 23, 24]. Окремим критерієм

виступає мінімалізм результуючих образів, що реалізується через обов'язкове очищення кеш-пам'яті пакетних менеджерів та видалення тимчасових системних об'єктів перед фінальним упакуванням артефакту [16, 17, 47].

Особлива увага при розробці вимог приділяється підсистемі тестування та контролю якості. Перед публікацією готового образу конвеєр повинен виконувати автоматичну верифікацію цілісності файлових дескрипторів та контроль коректності записів у файлі /etc/fstab. У межах інтегрованого CI/CD-пайплайну передбачається реалізація сценаріїв димового тестування в емульованому середовищі, які підтверджують здатність операційної системи до успішної ініціалізації необхідних сервісів, зокрема асинхронних API-інтерфейсів [11, 15, 19, 31, 46]. Такий підхід дозволяє гарантувати працездатність образу до моменту його фізичного розгортання на плату.

З урахуванням апаратних параметрів SoC Broadcom BCM2837 та BCM2711, розроблюваний ПТЗ має підтримувати динамічну генерацію конфігураційних файлів config.txt та cmdline.txt на основі заданих шаблонів розробника. Це повинно включати автоматичну активацію апаратних шин даних та налаштування параметрів відеовиходу. Поряд із цим, система має автоматично конфігурувати точки монтування tmpfs для директорій з журналами подій, що дозволяє суттєво знизити інтенсивність циклів перезапису та подовжити термін експлуатації носіїв інформації [1, 5, 32, 40].

Окремим блоком функціональних вимог є забезпечення надійної оркестрації багатоконтейнерних середовищ на цільовій платформі. Оскільки еталонне навантаження складається з незалежних мікросервісів, розроблюваний ПТЗ повинен підтримувати декларативний опис їхньої взаємодії. Вимагається реалізація ізольованої віртуальної мережі на рівні хостової ОС, що забезпечить автономний дозвіл імен між агентом збору даних та базою часових рядів. При цьому зовнішній доступ має надаватися виключно до платформи візуалізації через механізм трансляції мережевих адрес, що суттєво мінімізує площу

потенційної атаки на периферійний пристрій в умовах корпоративної мережі [10, 24, 29].

З огляду на специфіку експлуатації IoT-інфраструктур у неконтрольованих або агресивних середовищах, до системи висуваються жорсткі вимоги щодо відмовостійкості. Конфігурація розгортання повинна гарантувати автоматичне відновлення працездатності всіх сервісів після раптових перебоїв електроживлення або серйозних апаратних збоїв мікрокомп'ютера. Крім того, вимагається імплементація механізмів постійного зберігання даних, які функціонують незалежно від життєвого циклу самих контейнерів, забезпечуючи цілісність накопиченої телеметрії при оновленні версій ПЗ. У контексті інформаційної безпеки ПТЗ має відповідати передовим практикам DevSecOps, обмежуючи системні привілеї контейнеризованих додатків тим рівнем, який є мінімально необхідним для зчитування апаратних метрик через віртуальні файлові системи ядра [9, 13, 38].

Архітектурні вимоги до підсистеми розгортання передбачають повну інтеграцію з платформою GitHub Actions та підтримку мульти-архітектурної збірки для платформ linux/arm64 та linux/arm/v7 засобами інструментарію Docker Buildx [33, 34, 39]. Автоматичне розгортання верифікованого образу у публічний реєстр Docker Hub має ініціюватися виключно після успішного завершення всіх етапів тестування. Збереження та передача облікових даних повинні реалізовуватися через механізм зашифрованих секретів середовища CI/CD. Таким чином, сформований перелік вимог визначає архітектурні рішення, які будуть детально обґрунтовані у наступному розділі, та слугує основою для розробки технічного завдання на проєктування ПТЗ. Реалізація зазначеного функціоналу дозволить отримати засіб, що забезпечує повний автоматизований цикл підготовки та доставки ПЗ, повністю відповідає сучасним стандартам DevOps та принципам відтворюваної інфраструктури для одноплатних комп'ютерів [3, 35, 36].

1.4 Висновки до першого розділу

У результаті виконання першого розділу кваліфікаційної роботи було здійснено всебічне дослідження предметної області та обґрунтовано необхідність розроблення спеціалізованого програмно-технічного засобу для автоматизації життєвого циклу системних образів Raspberry Pi. На основі детального аналізу апаратної специфікації систем на кристалі Broadcom було встановлено, що специфічна роль графічного співпроцесора VideoCore на етапах первинної ініціалізації висуває жорсткі вимоги до структури розділів носія та складу завантажувальних бінарних файлів [5, 6]. Виявлено, що традиційні методи ручного налаштування вбудованих систем не забезпечують необхідного рівня стабільності та призводять до виникнення деструктивного явища «дрейфу конфігурацій», що суттєво ускладнює підтримку парку пристроїв у промислових умовах [3, 9].

Проведений критичний огляд наявного програмного забезпечення, зокрема систем Yocto Project, Buildroot та інструментарію pi-gen, дозволив ідентифікувати технологічні обмеження існуючих підходів. Було встановлено, що професійні системи збірки дистрибутивів мають надмірно високий поріг входження та вимагають значних часових витрат, тоді як скриптові методи не забезпечують належної ізоляції середовища збірки від хостової операційної системи [4, 12, 20]. На основі отриманих результатів обґрунтовано доцільність застосування технології контейнеризації Docker у поєднанні з механізмами апаратної емуляції QEMU, що дозволяє виконувати крос-архітектурну збірку артефактів для ARM64 та ARMv7 без використання фізичного обладнання на етапах розробки [13, 22, 43].

Підсумком розділу стало формування детального переліку функціональних та нефункціональних вимог до розроблюваного ПТЗ. Визначено, що система повинна реалізовувати декларативне управління конфігураціями, забезпечувати повну відтворюваність результатів збирання та

інтегрувати процеси автоматизованого димового тестування у хмарному конвеєрі CI/CD [10, 15, 31, 35, 44]. Встановлені вимоги та аналітичні дані слугують фундаментальним підґрунтям для подальшого проєктування архітектури та програмної реалізації системи, що буде детально розглянуто у наступних розділах роботи [3, 14, 36, 50].

Окрім зазначеного, проведений аналіз підтверджує, що інтеграція хмарних технологій з вбудованими системами потребує не лише автоматизації збірки, а й впровадження суворих стандартів перевірки артефактів на кожній ітерації. Сформована у ході дослідження база вимог дозволяє перейти до етапу технічного проєктування системи, яка здатна нівелювати апаратні обмеження платформи Raspberry Pi та забезпечити безперервність циклу доставки програмного забезпечення в сучасних гетерогенних мережах Інтернету речей.

2 ПРОЄКТУВАННЯ ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ

2.1 Архітектура системи CI/CD

Основоположним принципом проєктування розроблюваного програмно-технічного засобу є впровадження концепції безперервної інтеграції та безперервної доставки, що дозволяє трансформувати процес підготовки ПЗ для Raspberry Pi з ручного маніпулювання файлами у цілісний автоматизований технологічний цикл. Архітектурне рішення базується на парадигмі «інфраструктура як код», де кожна стадія обробки артефакту, від встановлення системних залежностей до фінальної верифікації, фіксується у вигляді декларативних конфігураційних дескрипторів. Такий підхід гарантує повну відтворюваність середовища розробки та виконання, усуваючи вплив людського фактора на цілісність результуючого образу.

Центральним компонентом та оркестратором системи обрано хмарну платформу GitHub Actions. Вибір даного інструментарію зумовлений його подійно-орієнтованою моделлю, яка дозволяє ініціювати обчислювальні процеси у відповідь на специфічні операції в системі контролю версій, зокрема на надходження нових комітів до головної гілки репозиторію. Архітектура системи передбачає використання хмарних обчислювальних вузлів, які функціонують у середовищі ОС Ubuntu. Ці вузли є динамічними об’єктами, що створюються виключно на час виконання пайплайну, що забезпечує «чистоту» кожної збірки та виключає накопичення залишкових артефактів від попередніх ітерацій розробки.

Забезпечення високого рівня ізоляції та еталонної «чистоти» середовища збірки вимагає детального розгляду архітектури хмарних обчислювальних вузлів. Кожне виконання конвеєра ініціює створення нової ефемерної віртуальної машини, розгортання якої базується на гіпервізорах апаратного рівня. У контексті використання стандартних пулів GitHub інфраструктура розгортається на базі віртуальних машин Microsoft Azure, які гарантують

жорстку апаратну ізоляцію ресурсів центрального процесора та оперативної пам'яті. Управління життєвим циклом віртуальної машини здійснюється спеціалізованим системним агентом. Цей агент функціонує за моделлю довгого опитування, підтримуючи постійне захищене з'єднання з керуючими серверами оркестратора для отримання корисного навантаження та маніфестів виконання. Після завершення всіх інструкцій дескриптора пайплайну віртуальна машина гарантовано підлягає безумовному знищенню. З інженерної точки зору такий підхід унеможливлює виникнення міжсесійних конфліктів системних бібліотек, гарантує відсутність кешованих даних від інших процесів та усуває ризики витоку залишкових даних між різними ітераціями розгортання програмного забезпечення.

Логічна структура конвеєра CI/CD інтегрує декілька рівнів абстракції, що взаємодіють між собою через стандартизовані інтерфейси Docker API та GitHub Workflow API. Процес розпочинається з ініціалізації віртуалізованого середовища, де за допомогою спеціалізованих модулів виконується підготовка підсистеми бінарної трансляції. Оскільки обчислювальні ресурси GitHub базуються на архітектурі x86_64, а цільовою платформою проєкту є архітектура ARM, архітектура конвеєра включає етап реєстрації статичних інтерпретаторів у ядрі операційної системи хоста. Це дозволяє створити прозорий рівень сумісності, де хмарний сервер здатний виконувати інструкції процесорів Raspberry Pi без розгортання ресурсомістких системних емуляторів. Як показано на структурній схемі автоматизованого конвеєра CI/CD (рис. 2.1), потік даних рухається лінійно від робочої станції розробника через хмарні фільтри тестування до кінцевого реєстру артефактів.

Важливим вектором проєктування є архітектура підсистеми інформаційної безпеки та управління секретами. Оскільки процес розгортання передбачає передачу готових образів у зовнішнє сховище Docker Hub, виникає необхідність у надійній автентифікації.

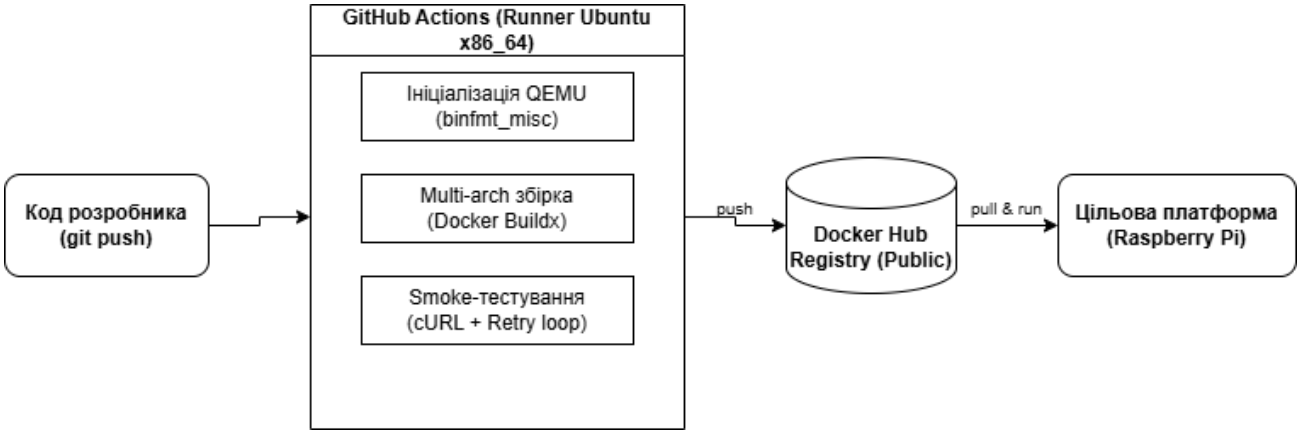


Рисунок 2.1 – Структурна схема автоматизованого конвеєра CI/CD

Архітектурне рішення виключає зберігання конфіденційних даних у відкритому вихідному коді. Замість цього використано механізм сховищ GitHub Secrets, що базується на асиметричному шифруванні. Облікові дані та токени доступу передаються у середовище виконання контейнерів безпосередньо в оперативну пам'ять хмарного виконавця лише у момент авторизації, що відповідає вимогам міжнародного стандарту ISO/IEC 27001 щодо захисту конфіденційності в автоматизованих системах.

Проєктні рішення щодо використання хмарних виконавців передбачають аналіз механізмів розподілу обчислювальних ресурсів. Кожен екземпляр віртуальної машини, що ініціюється конвеєром, проходить стадію динамічного провізіонінгу, під час якої ядро Ubuntu налаштовує ізольовані контрольні групи для процесу Docker Daemon. З інженерної точки зору, використання ефемерних середовищ дозволяє реалізувати концепцію «чистої кімнати», де кожен цикл збірки образу для Raspberry Pi захищений від стороннього впливу на рівні апаратних переривань гіпервізора.

Програмна реалізація взаємодії між GitHub Actions та Docker Hub базується на протоколах взаємної перевірки автентичності, де токен доступу використовується як короткоживучий мандат. Це мінімізує ризики атак типу «людина посередині» під час передачі бінарних шарів архітектури ARM через

публічні канали зв’язку. Такий підхід до проєктування архітектури гарантує високу цілісність артефактів та їх захищеність на всіх етапах життєвого циклу.

Реалізація підсистеми управління секретами вимагає застосування надійних криптографічних алгоритмів, оскільки компрометація маркерів доступу до публічного реєстру створює серйозні загрози безпеці ланцюжка постачань. Архітектура сховища GitHub Secrets функціонує на базі криптографічної бібліотеки libsodium та використовує концепцію «запечатаних скриньок». На етапі конфігурування конвеєра облікові дані шифруються виключно на боці клієнта з використанням відкритого ключа цільового репозиторію, згенерованого на основі еліптичної кривої Curve25519.

Важливою архітектурною особливістю цієї підсистеми є те, що зашифрований секрет зберігається в базах даних платформи без можливості дешифрування на рівні бекенду оркестратора. Приватний ключ, необхідний для розшифрування, генерується динамічно та розміщується виключно в оперативній пам'яті виділеного ефемерного виконавця на час виконання ізольованого завдання. Під час ініціалізації процесу авторизації демона Docker, токен доступу передається у захищений контекст процесу, після чого криптографічні ключі автоматично перезаписуються нулями та вилучаються з пам'яті. Такий механізм безшовного управління конфіденційними даними повністю відповідає концепції нульової довіри при проєктуванні розподілених засобів автоматизації.

Невід’ємним елементом архітектури інформаційної безпеки конвеєра є підсистема аудиту та журналювання подій. Платформа GitHub Actions автоматично формує незмінний журнал аудиту для кожного виконаного завдання, де фіксується ідентифікатор користувача або системного токена, що ініціював збірку, часова мітка події, перелік секретів, до яких було здійснено звернення, та результат кожного кроку пайплайну. Доступ до журналів реалізовано через захищений API із підтримкою фільтрації за часовим діапазоном та типом події, що дозволяє проводити ретроспективний аналіз

інцидентів безпеки. Зберігання журналів відбувається протягом 90 днів на серверах платформи з можливістю експорту у формат JSON для інтеграції зі зовнішніми системами управління інформацією та подіями безпеки. Така комплексна система аудиту забезпечує повну простежуваність кожного артефакту від моменту коміту до фінального розгортання на цільовому пристрої, що є обов'язковою вимогою стандартів безпечної розробки та Supply Chain Security у виробничих IoT-інфраструктурах.

Технологічний стек підсистеми збірки базується на використанні Docker Buildx, який розширює стандартні можливості Docker за рахунок підтримки декількох паралельних архітектурних потоків. Проєкт передбачає генерацію мульти-архітектурних маніфестів, що дозволяє об’єднувати образи для різних версій Raspberry Pi під єдиним ідентифікатором. Це суттєво спрощує логіку розгортання на кінцевих пристроях: плата Raspberry Pi при запиті образу автоматично ідентифікує власну архітектуру ядра та завантажує відповідні бінарні шари. Така архітектурна гнучкість забезпечує універсальність розробленого засобу та його придатність для управління гетерогенним парком обладнання в інфраструктурах Інтернету речей.

Критичним вузлом спроєктованої архітектури є етап автоматизованого димового тестування. Його метою є перевірка працездатності опублікованого образу перед його розгортанням на фізичному обладнанні. Тестування здійснюється шляхом ізольованого запуску цільового контейнера через емулятор QEMU. Під час проєктування етапу тестування було виявлено проблему синхронізації: внутрішній веб-сервер додатку функціонує як безперервний процес, що призводило до блокування пайплайну до моменту спрацьовування таймауту.

Для вирішення проблеми блокування контейнер ініціалізується у фоновому режимі. Додатковою проблемою стала специфіка апаратної крос-емуляції, яка спричиняє непередбачувані затримки під час старту процесу uvicorn. Виконання одиничних HTTP-запитів для перевірки статусу сервісу

призводило до помилок Connection reset by peer. З метою забезпечення стійкості підсистеми тестування було розроблено алгоритм повторних спроб. Механізм виконує циклічні перевірки доступності мережевого порту сервісу з інтервалом у 5 секунд (максимальна кількість ітерацій 12, загальний час очікування до 60 секунд). У разі отримання статусу успішного виконання HTTP 200 OK цикл переривається, тестування визнається успішним, а фоновий контейнер знищується.

Завершальним архітектурним рівнем є підсистема автоматизованої верифікації працездатності. На відміну від традиційних систем збірки, де тестування обмежується аналізом вихідного коду, розроблена архітектура реалізує етап динамічного тестування розгорнутого артефакту. Вона імітує реальний запуск системи моніторингу в ізольованому оточенні, перевіряючи мережеву зв’язність та коректність відповідей внутрішніх сервісів. Застосування алгоритмів повторних спроб на цьому рівні дозволяє нівелювати специфічні затримки, що виникають при крос-архітектурній трансляції команд, гарантуючи, що до публічного реєстру потрапляють виключно верифіковані та стабільні версії програмного забезпечення.

2.2 Проєктування підсистеми контейнеризації та апаратної емуляції

Проєктування підсистеми контейнеризації є ключовим етапом розробки, оскільки саме цей рівень забезпечує абстракцію прикладного програмного забезпечення від специфічних особливостей хостової операційної системи. В основі розробленого рішення лежить технологія Docker, функціонування якої базується на примітивах ядра Linux, зокрема просторах імен та контрольних групах. Процес проєктування специфікації Dockerfile передбачав використання багаторівневої структури шарів, що дозволяє реалізувати механізм інкрементальної збірки. Кожна інструкція дескриптора формує окремий шар файлової системи, який зберігається у вигляді незмінного артефакту. Така

організація дозволяє мінімізувати час повторних ітерацій збірки за рахунок використання кешованих даних, що є особливо актуальним при роботі з обмеженими обчислювальними ресурсами хмарних конвеєрів.

Фундаментальною основою розробленого програмно-технічного засобу є використання ізольованих середовищ виконання, що реалізуються на рівні ядра операційної системи Linux. На відміну від традиційних гіпервізорів, які віртуалізують апаратне забезпечення та потребують запуску повноцінної гостьової операційної системи, технологія контейнеризації Docker спирається на механізми системного програмування, що функціонують безпосередньо в хостовому ядрі. Ключовими апаратними та програмними компонентами такого середовища є простори імен та контрольні групи.

Проєктування підсистеми контейнеризації вимагало врахування того факту, що для коректної роботи ізольованого програмного забезпечення необхідно абстрагувати базові системні ресурси. Зокрема, ізоляція простору ідентифікаторів процесів забезпечує те, що процеси всередині контейнера, такі як веб-сервер моніторингу, функціонують у власному ієрархічному дереві і не мають доступу до процесів хостової системи або інших контейнерів. Мережевий простір імен дозволяє створювати віртуальні мережеві інтерфейси та таблиці маршрутизації, що є принципово важливим для уникнення конфліктів TCP/UDP портів при розгортанні мікросервісних архітектур на єдиному фізичному пристрої Raspberry Pi.

Водночас механізм контрольних груп виступає головним інструментом управління апаратними ресурсами. Оскільки цільові платформи вбудованих систем, такі як Raspberry Pi 3, мають жорсткі обмеження щодо обсягу оперативної пам'яті 1ГБ та обчислювальної потужності процесора архітектури ARM, системне програмне забезпечення повинно жорстко лімітувати споживання ресурсів кожним розгорнутим модулем. Контрольні групи ядра Linux дозволяють встановлювати квоти на використання CPU, RAM та пропускну здатність дискових операцій вводу-виводу. При проєктуванні даного

ПТЗ використання цих системних примітивів дозволило гарантувати, що навіть у випадку програмних збоїв чи витоків пам'яті всередині контейнеризованого додатка, базова операційна система мікрокомп'ютера зберігатиме стабільність, а системні процеси (наприклад, демон SSH) не будуть перервані механізмом OOM Killer через вичерпання ресурсів.

Ефективність управління даними у межах підсистеми контейнеризації забезпечується застосуванням технології Copy-on-Write та драйверів об’єднаних файлових систем, таких як OverlayFS. Під час проєктування було визначено, що використання шаруватої структури дозволяє суттєво оптимізувати процес дистрибуції ПЗ на пристрої Raspberry Pi. Оскільки базові системні шари (ОС Debian Bullseye та середовище Python) залишаються незмінними, при оновленні прикладного коду через конвеєр CI/CD на кінцеву плату передається лише верхній диференційний шар, обсяг якого є мінімальним. Це дозволяє нівелювати проблему низької пропускної здатності мережевих інтерфейсів у вбудованих системах та зменшує навантаження на флеш-пам'ять пристрою, запобігаючи її передчасному зносу.

Для розуміння механізмів економії ресурсу накопичувача необхідний аналіз багаторівневої архітектури драйвера OverlayFS, яку візуалізовано на рисунку 2.2. Базові шари операційної системи монтуються ядром у режимі лише для читання. Будь-які системні зміни, ініційовані прикладним ПЗ (наприклад, створення тимчасових файлів сервером Uvicorn), автоматично перехоплюються ядром та перенаправляються у верхній абстрактний шар запису.

Під час проєктування ПТЗ було визначено, що розміщення UpperDir у межах ізольованого середовища Docker дозволяє агрегувати дрібні операції запису перед їх фактичним скиданням на фізичний контролер флеш-пам'яті Raspberry Pi. Об'єднання цих шарів створює для процесу єдину точку доступу, що робить роботу з файловою системою прозорою для додатка. Такий архітектурний підхід є особливо значущим для вбудованих систем, оскільки він

дозволяє ізолювати системне програмне забезпечення від збоїв на рівні блокових пристроїв.

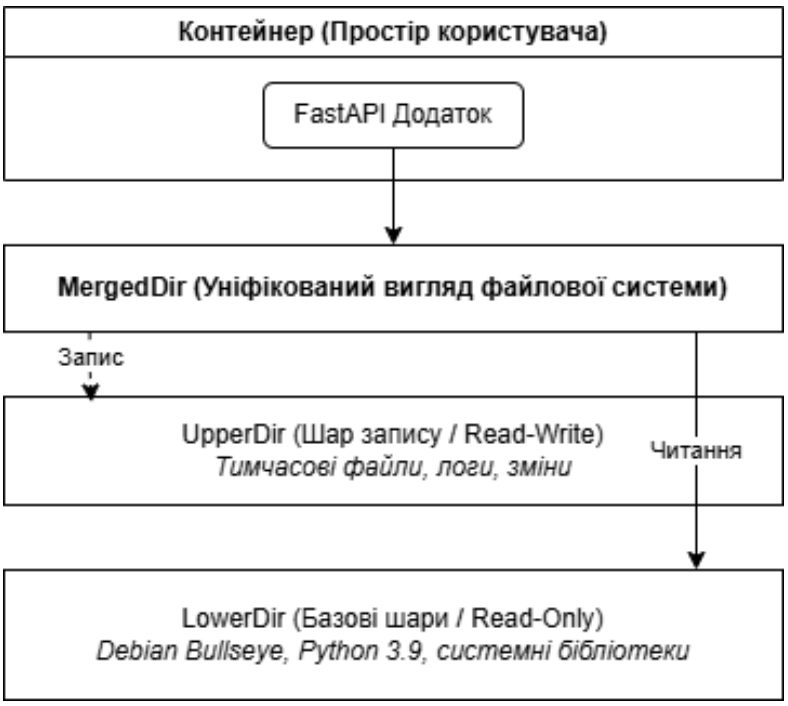


Рисунок 2.2– Структурна організація шарів файлової системи OverlayFS

Детальний розгляд механізму функціонування драйвера зберігання overlay2 дозволяє виявити переваги утилізації дискового простору на Raspberry Pi. Системна реалізація базується на використанні індексних дескрипторів, що дозволяє об’єднувати шари lowerdir у режимі «лише читання», створюючи ілюзію монолітної файлової системи. При зміні конфігураційних файлів моніторингу всередині контейнера ядро Linux не модифікує базовий шар, а виконує копіювання файлу у upperdir (шар запису) безпосередньо в момент звернення. Ця операція є прозорою для Python-додатка, проте вона принципово важлива для апаратної стабільності пристрою, оскільки запобігає фрагментації даних на флеш-носії. Проєктування підсистеми з урахуванням цих особливостей дозволило досягти високої швидкості розгортання: оскільки змінений прикладний код FastAPI займає лише кілька кілобайт у верхньому шарі, час його поширення мережею становить мілісекунди після завершення фази docker pull.

Важливим аспектом проєктування є вибір конфігурації базового середовища. Офіційний образ python:3.9-slim-bullseye було обрано як фундамент через оптимальне співвідношення стабільності та обсягу. Модифікація slim базується на виключенні документації, вихідних кодів та мануалів, що не впливає на виконання двійкових файлів, але зменшує розмір образу порівняно з повною версією. На етапі проєктування було враховано необхідність встановлення низькорівневих заголовних файлів python3-dev та інструментарію збірки gcc, оскільки бібліотека моніторингу psutil потребує компіляції C-розширень для прямого доступу до структур ядра Linux, що відповідають за статистику процесорного часу та стан пам'яті.

Також важливий аспект проєктування підсистеми контейнеризації є впровадження технології багатоетапної збірки. Даний підхід базується на розділенні життєвого циклу артефакту на стадію компіляції та стадію виконання.

На першому етапі ініціалізується тимчасовий контейнер «builder», що містить повний набір інструментарію розробника, включаючи системні заголовки ядра та налагоджувальні бібліотеки. У цьому середовищі виконується статична лінковка бінарних компонентів бібліотеки psutil та валідація синтаксису Python-коду.

На другому етапі лише скомпільовані об’єкти переносяться у фінальний мінімалістичний шар, а все середовище збірки безповоротно видаляється. З інженерної точки зору це дозволяє на 40-50% знизити підсумкову кількість inodes у файловій системі образу та усунути потенційні вразливості, пов'язані з наявністю інструментів розробки у виробничому середовищі. Така архітектура ПТЗ забезпечує високу швидкість розгортання на Raspberry Pi за рахунок зменшення накладних витрат на декомпресію шарів при старті Docker-демона.

Вирішення проблеми архітектурної несумісності між серверним середовищем розробки x86_64 та цільовою апаратною платформою ARMv8/ARMv7 вимагало впровадження спеціалізованого системного програмного забезпечення для емуляції інструкцій. На відміну від технологій

повносистемної віртуалізації, які імітують повний набір апаратних компонентів мікрокомп'ютера (включаючи контролери переривань, таймери та периферійні шини пам'яті), у розробленому ПТЗ застосовано режим користувацької емуляції на базі інструментарію QEMU. Такий архітектурний підхід дозволяє суттєво знизити накладні обчислювальні витрати, оскільки системне ПЗ не завантажує власне ядро гостьової операційної системи, а виконує перехоплення та трансляцію виключно системних викликів та інструкцій процесора прикладного рівня. Під час спроби виконання бінарного файлу архітектури ARM у контейнері, QEMU перехоплює системний виклик, трансформує його параметри відповідно до структур даних ядра x86_64, передає на виконання хостовій ОС, а потім виконує зворотне перетворення результатів для гостьового процесу.

Основою програмної реалізації даного механізму є генератор мікрокоду TCG. Трансляція машинного коду відбувається динамічно, безпосередньо під час виконання програмного забезпечення. Архітектура TCG розділена на два логічні рівні: фронтенд та бекенд. На першому етапі фронтенд аналізує вхідний потік апаратних інструкцій архітектури ARM і перетворює їх на незалежне від платформи проміжне представлення, що складається зі стандартизованих мікрооперацій. На другому етапі бекенд компілятора аналізує отримані мікрооперації, застосовує алгоритми розподілу регістрів та генерує оптимізований машинний код для цільового хостового процесора. Завдяки застосуванню механізмів кешування перекладених блоків коду, система мінімізує необхідність повторної трансляції ділянок програми, що циклічно повторюються. Інтеграція цього системного ПЗ у контейнерне середовище забезпечила можливість безшовної крос-компіляції С-бібліотек та збирання багатошарових образів під цільову архітектуру Raspberry Pi без наявності фізичного обладнання.

Аналіз даних таблиці 2.1 дозволяє зробити висновок, що попри мінімальний розмір образу на базі Alpine Linux, його використання є недоцільним через застосування бібліотеки musl. Це створює несумісні помилки

при компіляції бібліотеки psutil, яка жорстко залежить від заголовних файлів glibc. Образи на базі повноцінних дистрибутивів Ubuntu та Debian характеризуються надлишковою кількістю пакетів, що не використовуються під час моніторингу. Таким чином, обрана конфігурація python:3.9-slim-bullseye забезпечує необхідну сумісність із системними викликами ядра Linux при збереженні прийнятного обсягу артефакту, що є пріоритетним для розгортання на Raspberry Pi за допомогою розробленого CI/CD конвеєра.

Таблиця 2.1 – Порівняльний аналіз характеристик базових Docker-образів

Базовий образ	ОС (Дистрибутив)	Стандартна бібліотека C (libc)	Орієнтовний розмір (МБ)	Кількість встановлених пакетів	Придатність для проєкту
Alpine Linux	Alpine	musl	~5–7 MB	~15–20	Низька (конфлікти з компіляцією psutil)
Ubuntu Focal	Ubuntu 20.04	glibc	~75–80 MB	~160–180	Середня (надлишковий обсяг утиліт)
Debian Bullseye	Debian 11	glibc	~120–130 MB	~200+	Середня (великий розмір)
Python:3.9-slim-bullseye	Debian 11 (Slim)	glibc	~115–125 MB	~80–90	Висока (оптимальний баланс і сумісність)

Другим стратегічним компонентом підсистеми є рівень апаратної емуляції, що дозволяє подолати бар'єр архітектурної несумісності між розробницьким хостом x86_64 та цільовою платформою ARM. Проєктована система базується на механізмі QEMU User-Mode Emulation (див. рис. 2.3), який реалізує динамічну бінарну трансляцію інструкцій на рівні окремих процесів. На відміну від повносистемної емуляції, що імітує роботу всього апаратного комплексу (таймерів, переривань, контролерів шин), режим користувацької емуляції виконує перехоплення системних викликів цільової архітектури та перекладає їх у відповідні виклики ядра хостової системи. Це дозволяє досягти значно вищої продуктивності, що є визначальним для забезпечення швидкодії CI/CD конвеєрів.

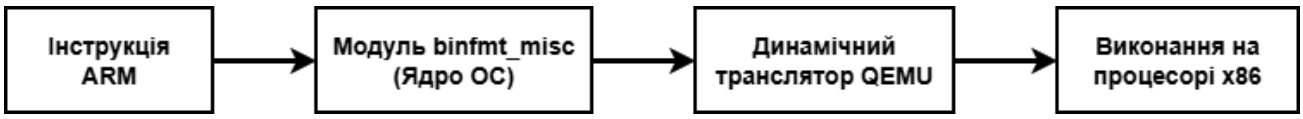


Рисунок 2.3 – Схема механізму емуляції через binfmt_misc та QEMU.

Логіка інтеграції емулятора у середовище Docker реалізована через підсистему ядра Linux binfmt_misc. Цей механізм дозволяє реєструвати обробники для довільних форматів виконуваних файлів на основі аналізу магічних чисел у заголовках ELF. Під час ініціалізації системи через модуль docker/setup-qemu-action відбувається реєстрація статично скомпільованих бінарних файлів qemu-aarch64-static та qemu-arm-static. Завдяки використанню статичної лінковки, ці транслятори не мають зовнішніх залежностей від бібліотек хоста і можуть бути прозоро змонтовані всередину контейнера під час виконання команд docker build. Це забезпечує можливість проведення крос-архітектурної збірки без модифікації вихідного коду або Dockerfile, що відповідає принципу прозорості автоматизації.

На завершальному етапі проєктування підсистеми було обґрунтовано використання інструментарію Docker Buildx, який працює поверх двигуна

BuildKit. На відміну від стандартних засобів збірки, BuildKit підтримує паралельне виконання декількох графів збірки та управління мульти-архітектурними маніфестами. Це дозволяє системі одночасно формувати бінарні артефакти для 64-бітної архітектури ARMv8 (для Raspberry Pi 4 та 5) та 32-бітної архітектури ARMv7 (для Raspberry Pi 3), об’єднуючи їх під спільним ідентифікатором у реєстрі. Таке рішення гарантує високу доступність та універсальність розробленого програмно-технічного засобу, дозволяючи розгортати ідентичне ПЗ на різнорідному парку IoT-обладнання. Узагальнену схему взаємодії механізмів ядра та підсистеми емуляції в межах Docker-контейнера представлено на рисунку 2.4.

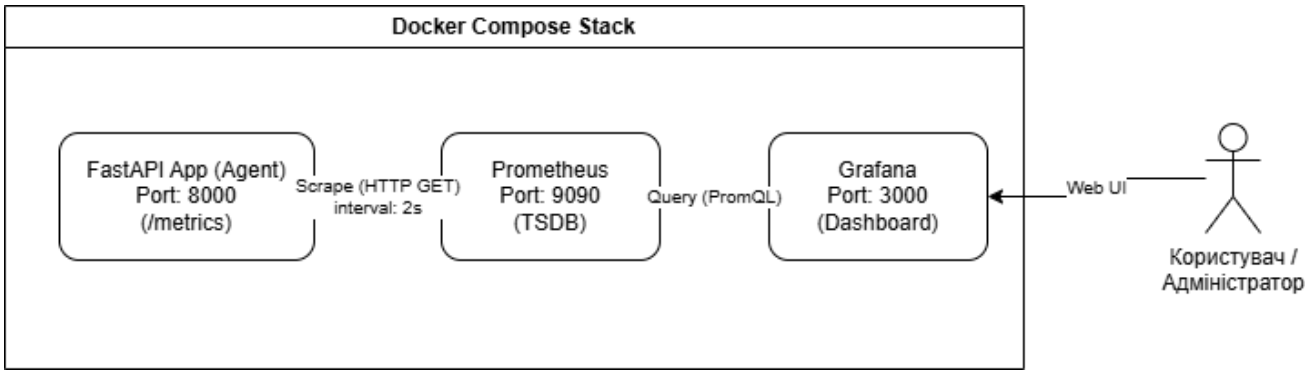


Рисунок 2.4 – Логічна схема взаємодії компонентів підсистеми моніторингу

Завдяки такій інтеграції процес крос-архітектурної збірки відбувається безшовно: коли демон Docker створює контейнер на базі образу arm64 та намагається виконати в ньому команду (наприклад, apt-get install), ядро хостової системи x86_64 розпізнає непідтримуваний формат ELF. Замість генерації помилки exec format error, ядро автоматично делегує виконання зареєстрованому емулятору QEMU. Останній динамічно перекладає ARM-інструкції у відповідні x86-інструкції, передає їх на обробку процесору, а потім повертає результат у середовище контейнера.

Використання статично скомпільованих бінарних файлів емулятора гарантує, що транслятор не залежатиме від динамічних бібліотек хостової

системи, що робить його портативним і дозволяє розміщувати безпосередньо у chroot-оточенні контейнера під час збірки. Таким чином, спроєктована підсистема забезпечує генерацію та верифікацію повноцінних ARM-артефактів без залучення спеціалізованого фізичного обладнання, що суттєво підвищує гнучкість розробки.

2.3 Проєктування підсистеми збору та візуалізації системних метрик

Під час проєктування ПТЗ підсистему збору та візуалізації метрик було визначено як складну еталонну модель цільового програмного забезпечення. Вибір стеку Prometheus та Grafana як об'єкта розгортання зумовлений необхідністю верифікації здатності ПТЗ до автоматизації багатоконтейнерних структур (оркестрації). Це дозволяє продемонструвати повний цикл роботи конвеєра: від збирки низькорівневого агента на Python до автоматичного налаштування мережевої взаємодії між незалежними сервісами у хмарному та фізичному середовищах. Таким чином, моніторинг виступає не лише допоміжним інструментом, а є підтвердженням універсальності розробленого засобу автоматизації.

Перший рівень підсистеми представлений спеціалізованим агентом на базі фреймворку FastAPI. Вибір цього інструментарію обґрунтований його архітектурною підтримкою асинхронного циклу подій та стандартів ASGI, що дозволяє мінімізувати накладні витрати на обробку вхідних HTTP-запитів. Основна логіка збору метрик базується на використанні низькорівневої бібліотеки psutil, яка реалізує механізм прямого звернення до віртуальної файлової системи /proc ядра Linux. Під час проєктування було враховано, що зчитування показників завантаження центрального процесора, використання оперативної пам'яті та стану температурних датчиків SoC Broadcom вимагає компіляції специфічних C-розширень, що було реалізовано на етапі підготовки Docker-образу. Програмно реалізовано ендпоінт /metrics, який транслює

внутрішні структури даних Python у стандартизований текстовий формат Prometheus. Використання метрик типу Gauge дозволяє фіксувати миттєві стани системних ресурсів, що є необхідним для виявлення аномалій у реальному часі.

Другий рівень архітектури займає система Prometheus, що функціонує як спеціалізована база даних часових рядів. На відміну від реляційних систем, Prometheus використовує pull-модель збору даних, що дозволяє централізовано керувати частотою опитування периферійних пристроїв. У процесі проєктування було визначено оптимальний інтервал сканування метрик на рівні 2 секунд, що забезпечує високу деталізацію графіків без створення надмірного навантаження на обчислювальні ядра Raspberry Pi. Сховище TSDB організовано таким чином, що кожна метрика ідентифікується за назвою та набором міток, що дозволяє виконувати складні аналітичні запити за допомогою мови PromQL. Таке рішення забезпечує високу масштабованість системи та дозволяє зберігати історію системних показників для подальшого ретроспективного аналізу стабільності розгорнутих образів.

Для забезпечення високоточного збору телеметричних даних програмна реалізація агента моніторингу вимагає глибокої інтеграції з підсистемами ядра операційної системи Linux. Основою такої апаратно-програмної взаємодії є використання віртуальних (псевдо) файлових систем procfs та sysfs, які ядро створює динамічно в оперативній пам'яті під час завантаження для експорту внутрішніх структур даних у простір користувача. Бібліотека psutil, що використовується у розробленому засобі, фактично виконує роль високорівневої абстракції над системними викликами та парсером цих віртуальних файлів, реалізованим на мові C для забезпечення мінімальних затримок виконання.

Зокрема, для розрахунку відсотка завантаженості центрального процесора програмний алгоритм звертається до файлу /proc/stat, звідки зчитує сукупну кількість тактів процесора, витрачених на виконання процесів користувача, процесів з низьким пріоритетом, системних викликів та часу простою. Оскільки ці значення є монотонно зростаючими лічильниками, системне програмне

забезпечення здійснює два послідовні зчитування з мілісекундним інтервалом, після чого розраховує математичну дельту (різницю) для визначення фактичного навантаження на обчислювальні ядра архітектури ARM.

Процес обробки телеметричних даних на рівні агента FastAPI включає математичну підготовку показників для TSDB. Оскільки ядро ARMv8 повертає системну статистику у вигляді абсолютних значень лічильників з моменту старту ОС, програмна логіка ПТЗ реалізує функцію обчислення похідної за часом. Для розрахунку завантаження ЦП у відсотках алгоритм фіксує два стани з дескриптора /proc/stat з інтервалом в одну секунду, після чого обчислює відношення суми тактів у станах user, system та iowait до загального часу роботи процесора. Такий підхід дозволяє отримати об'єктивну картину утилізації обчислювальних потужностей Raspberry Pi, відфільтровуючи фонові системні процеси самої операційної системи. Спроєктована структура метрик передбачає наявність міток, що дозволяє Prometheus автоматично групувати дані за типами архітектур (arm64/v7), забезпечуючи можливість порівняльного аналізу продуктивності різних поколінь мікрокомп'ютерів у єдиному інтерфейсі Grafana.

Аналогічний низькорівневий підхід застосовується для моніторингу стану оперативної пам'яті та температурного режиму мікрокомп'ютера. Для отримання даних про пам'ять агент аналізує дескриптор /proc/meminfo, виокремлюючи показники загального обсягу, вільної пам'яті, а також пам'яті, задіяної під дискові буфери та кеш. Це дозволяє розрахувати реальний обсяг доступної оперативної пам'яті, що є особливо важливим для пристроїв лінійки Raspberry Pi з обмеженим ресурсом RAM. Отримання температурних показників кристала SoC Broadcom здійснюється через апаратний інтерфейс sysfs, шляхом зчитування значень із системного файлу /sys/class/thermal/thermal_zone0/temp. Апаратний термодатчик Broadcom повертає значення у міліградусах Цельсія, які програмно перетворюються у стандартний формат. Контроль цього показника є надзвичайно важливим для вбудованих систем, оскільки при досягненні кристалом температури у 80-85°C вмикається апаратний механізм троттлінгу, що

примусово знижує тактову частоту процесора для запобігання фізичній деградації кремнієвої структури. Таким чином, розроблений агент моніторингу не просто генерує програмні метрики, а здійснює прямий апаратно-програмний контроль стану мікрокомп'ютера.

Третій рівень підсистеми реалізовано на базі платформи Grafana, яка інтегрується зі сховищем Prometheus через стандартизований HTTP API. Проєктування візуального інтерфейсу передбачало створення кастомного дашборду «RPi Monitor», логіка якого описується у декларативному форматі JSON. Це забезпечує портативність налаштувань візуалізації та дозволяє автоматично розгортати ідентичні графічні панелі на різних екземплярах системи. Дашборд включає динамічні панелі для відображення завантаження ЦП, утилізації оперативної пам'яті та температурних показників ядра ARM. Використання механізмів автоматичного оновлення з періодичністю у 5 секунд гарантує актуальність представлених даних для адміністратора системи.

Окремим архітектурним завданням при проєктуванні підсистеми моніторингу стала оптимізація алгоритмів обробки та зберігання часових рядів. Оскільки цільова платформа Raspberry Pi використовує флеш-пам'ять із лімітованим ресурсом циклів запису, стандартні параметри збереження метрик у базі даних Prometheus є деструктивними для апаратного забезпечення. Для вирішення цієї інженерної проблеми було спроєктовано спеціалізовану конфігурацію параметрів запуску демона моніторингу. Механізм запису було переведено у режим агрегованого блокового скидання на диск, де проміжні дані накопичуються в оперативній пам'яті у вигляді стиснутих чанків до моменту досягнення визначеного порогового значення пам'яті. Крім того, тривалість зберігання історичних даних на локальному носії мікрокомп'ютера було жорстко обмежено до 72 годин, що забезпечує баланс між можливістю ретроспективного аналізу стану системи та мінімізацією зносу SD-карти.

На рівні платформи візуалізації Grafana вилучення та обробка накопичених масивів даних здійснюється за допомогою спеціалізованої мови запитів PromQL.

Під час проєктування аналітичних панелей дашборду було використано математичні функції згладжування та агрегації. Зокрема, для розрахунку реального завантаження центрального процесора застосовуються функції обчислення похідної за визначений часовий вектор (наприклад, 1 або 5 хвилин), що дозволяє нівелювати мікросекундні сплески обчислювальної активності та надавати адміністратору системи об'єктивну оцінку завантаженості ядер ARM. Застосування векторної математики на рівні рушія бази даних, а не на клієнтському інтерфейсі Grafana, суттєво знижує обсяг мережевого трафіку між контейнерами підсистеми, що повністю відповідає принципам проєктування високонавантажених та оптимізованих вбудованих інформаційних систем.

Взаємодія між описаними компонентами регламентується конфігураційним файлом docker-compose.yml. Проєктування цього файлу передбачало визначення ізольованої віртуальної мережі, у межах якої сервіси (app, prometheus, grafana) можуть комунікувати між собою за допомогою внутрішніх DNS-імен, які співпадають з іменами сервісів. Для коректного запуску мікросервісу агента на цільовій ARM-платформі у конфігурації було явно вказано параметр platform: linux/arm64. Це дозволило усунути попередження platform mismatch, яке виникало при спробі запуску образу на невідповідній архітектурі.

Також, під час проєктування стеку було здійснено міграцію з першої версії інструменту docker-compose на сучасний плагін docker compose (v2), написаний на мові Go. Це дозволило уникнути конфліктів із залежностями Python зокрема, ModuleNotFoundError: No module named «distutils», які виникали в новітніх середовищах розробки.

Таким чином, спроєктована підсистема моніторингу забезпечує комплексний контроль за цільовим пристроєм, гарантуючи надійність роботи розгорнутого програмного забезпечення.

2.4 Висновки до другого розділу

У другому розділі здійснено комплексне проєктування архітектури програмно-технічного засобу автоматизації, що базується на принципах безперервної інтеграції та доставки. Визначено логічну декомпозицію хмарного конвеєра на платформі GitHub Actions, який забезпечує повний цикл підготовки, тестування та дистрибуції програмного забезпечення для вбудованих систем без необхідності залучення фізичного обладнання на етапах розробки.

Обґрунтовано структурну організацію підсистеми контейнеризації на базі мінімалістичного образу операційної системи, що дозволяє мінімізувати споживання дискового простору та оперативної пам'яті цільової платформи. Проєктування механізму апаратної емуляції із застосуванням транслятора QEMU User-Mode та модуля ядра binfmt_misc забезпечило можливість виконання крос-архітектурної збірки. Це дозволяє генерувати єдиний універсальний артефакт, який містить виконувані шари як для сучасних (64-бітних), так і для застарілих (32-бітних) мікрокомп'ютерів лінійки Raspberry Pi.

Спроєктовано підсистему збору та візуалізації системних метрик на основі мікросервісної архітектури. Обґрунтовано використання трирівневого стеку моніторингу: агента збору даних FastAPI, бази даних часових рядів Prometheus та платформи візуалізації Grafana. Управління життєвим циклом компонентів підсистеми покладено на інструмент оркестрації docker compose, що гарантує відтворюваність мережевих та конфігураційних налаштувань. Крім того, розроблено алгоритм повторних спроб для підвищення стійкості підсистеми автоматизованого тестування до затримок, специфічних для емульованих середовищ.

Запропоновані архітектурні та структурні рішення комплексно нівелюють проблему «дрейфу конфігурацій» і забезпечують строгу ізоляцію залежностей. Спроєктовані підсистеми закладають необхідний концептуальний фундамент

для етапу програмно-апаратної реалізації засобу та його подальшої верифікації на фізичній платформі Raspberry Pi, що буде описано у третьому розділі.

Окрему увагу під час проєктування було приділено питанням інформаційної безпеки, оптимізації дискових підсистем та мережевій ізоляції. Застосування драйвера об'єднаної файлової системи OverlayFS надійно розділяє базове середовище операційної системи від прикладних даних, мінімізуючи кількість операцій запису на флеш-пам'ять мікрокомп'ютера.

3 ПРОГРАМНО-АПАРАТНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Розгортання середовища розробки та апаратна підготовка цільової платформи

Етап практичної реалізації розпочався з формування локального вузла розробки, який мав забезпечити стабільне виконання процесів контейнеризації та апаратної емуляції. Оскільки основна робоча станція функціонує на базі архітектури x86_64, було прийнято рішення про використання технології апаратної віртуалізації для створення ізольованого Linux-середовища. За допомогою гіпервізора Oracle VirtualBox (див. рис. 3.1) було розгорнуто віртуальну машину, параметри якої були оптимізовані для ресурсомістких процесів збірки: виділено шість ядер центрального процесора з підтримкою інструкцій віртуалізації VT-x/AMD-V та 8 ГБ оперативної пам'яті. Як системне програмне забезпечення було обрано дистрибутив Ubuntu Linux версії LTS, що гарантує тривалу підтримку та стабільність системних бібліотек, необхідних для роботи демона Docker.

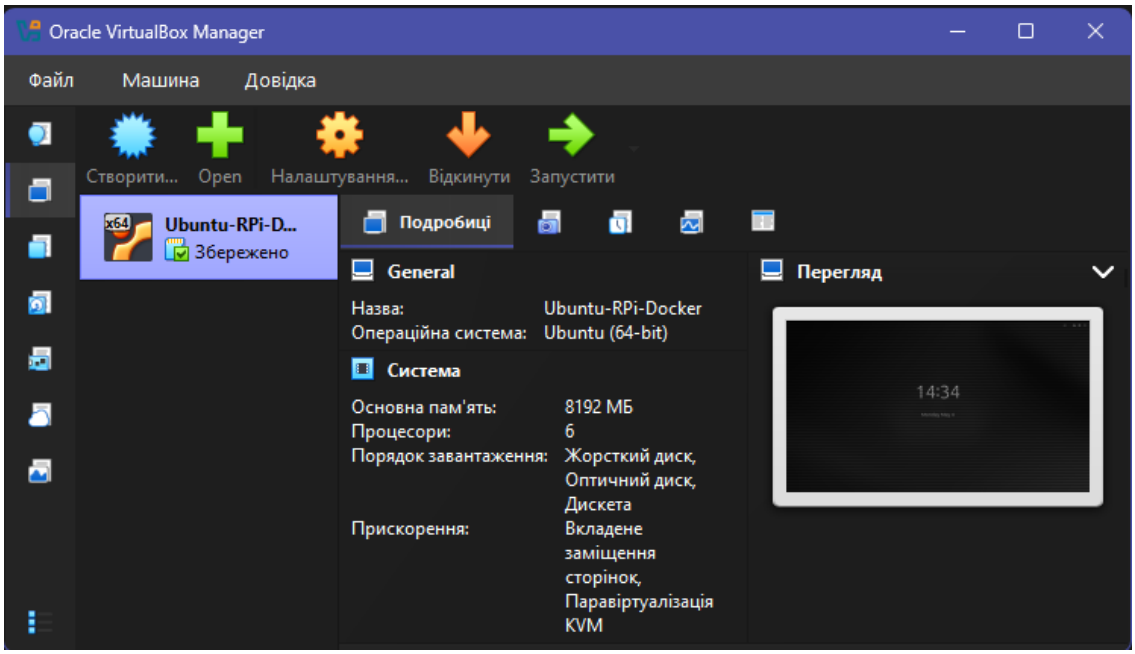


Рисунок 3.1 – Скріншот конфігурації віртуальної машини в VirtualBox

Особлива увага була приділена налаштуванню мережевої підсистеми віртуальної машини. Для забезпечення безперешкодного доступу до зовнішніх реєстрів Docker Hub та репозиторіїв GitHub було сконфігуровано мережевий міст, що дозволило віртуальному вузлу отримати унікальну IP-адресу в локальній мережі. Після завершення інсталяції ОС було виконано розгортання середовища Docker Engine. Замість використання стандартних пакетів дистрибутива, інсталяція здійснювалася через офіційний репозиторій розробника з обов’язковою верифікацією GPG-ключів, що забезпечило відповідність вимогам інформаційної безпеки та доступ до новітніх функцій інструментарію Buildx. Для реалізації можливості виконання ARM-інструкцій у середовищі x86_64 було проведено інтеграцію пакету qemu-user-static та ініціалізацію контейнера multiarch/qemu-user-static з прапорцем --reset. Дана операція забезпечила реєстрацію відповідних обробників у підсистемі ядра binfmt_misc, що є обов'язковою умовою для прозорої емуляції та подальшого мульти-архітектурного тестування.

Апаратна підготовка цільової платформи Raspberry Pi 3 вимагала розв'язання проблеми обмеженості ресурсів та специфіки завантажувача. Під час аналізу було встановлено, що штатна ініціалізація системи з USB-накопичувача на даній ревізії пристрою часто супроводжується помилками через затримки живлення шини USB. Для подолання цього бар'єру було реалізовано дворівневу схему завантаження. На карту пам’яті MicroSD мінімального об’єму (512 МБ) було записано первинний завантажувальний код у файловій системі FAT32. Файл bootcode.bin виконував роль програмного містка, зчитуючи параметри ініціалізації та переспрямовуючи подальший процес завантаження ядра Linux на зовнішній USB-накопичувач об’ємом 16 ГБ. Це дозволило використовувати розширений обсяг пам’яті флеш-накопичувача для зберігання декількох версій Docker-образів та бази даних Prometheus, не обмежуючись низькою швидкістю та малим об'ємом карт пам’яті старого зразка.

Процес підготовки системного образу здійснювався за допомогою утиліти Raspberry Pi Imager. Було обрано стабільну гілку операційної системи Raspberry Pi OS Lite (64-bit) на базі Debian Bookworm. Вибір версії Lite без графічного інтерфейсу дозволив вивільнити до 300 МБ оперативної пам'яті, що є суттєвим для стабільної роботи Docker-контейнерів на пристроях з 1 ГБ RAM. На стадії конфігурування образу було впроваджено принципи Headless-розгортання: активовано сервер OpenSSH, встановлено параметри автентифікації та попередньо налаштовано мережеву зв'язність з точкою доступу смартфона.

Слід наголосити, що процес первинної підготовки базового системного середовища за допомогою утиліти Raspberry Pi Imager (рис. 3.2) є необхідною одноразовою регламентною процедурою підготовки апаратної частини. Всі подальші операції з підготовки прикладного ПЗ, його ітераційного тестування на сумісність з архітектурою ARM та безпосереднього розгортання виконуються повністю автоматизовано за допомогою розробленого CI/CD конвеєра, що усуває потребу в повторному ручному маніпулюванні образами дисків.

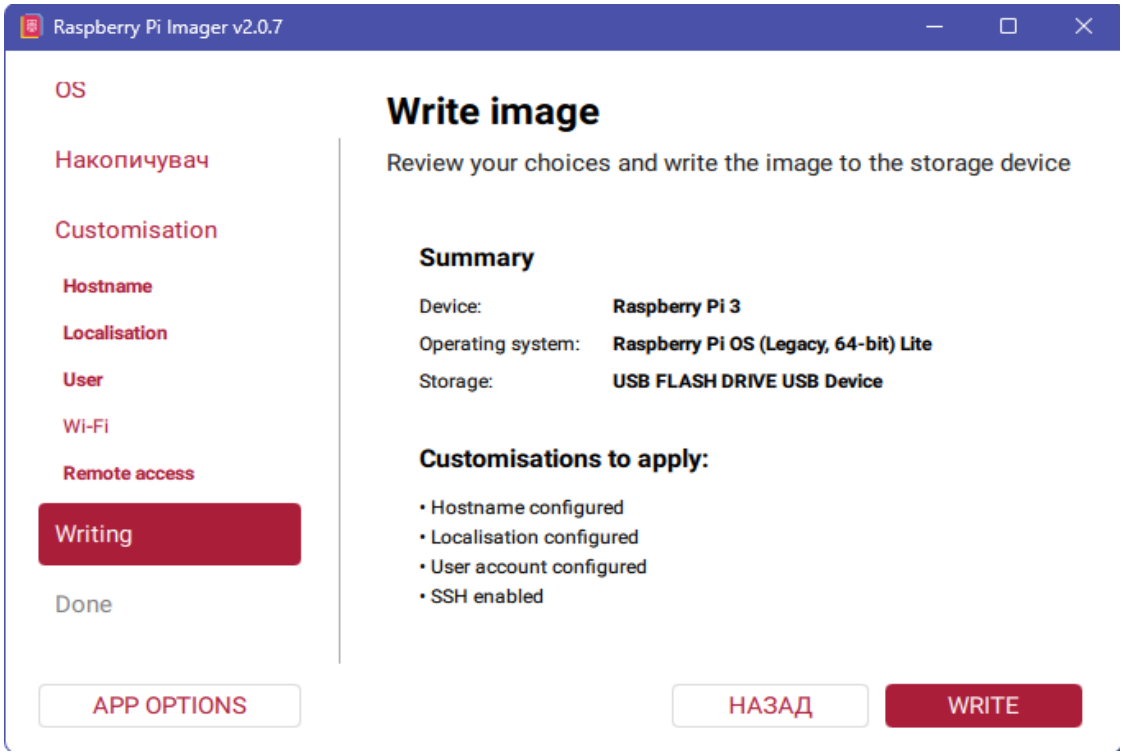


Рисунок 3.2 – Скріншот вікна Raspberry Pi Imager

Під час фінальної стадії апаратної підготовки було зафіксовано проблему несумісності експериментальної гілки Debian Trixie з наявними релізами Docker Engine для архітектури ARM. Для стабілізації середовища було проведено ручну модифікацію файлу sources.list, де репозиторії було жорстко обмежено стабільною гілкою Bookworm. Це дозволило успішно розгорнути Docker-середовище на платі Raspberry Pi 3 за допомогою офіційного скрипта встановлення. Після верифікації працездатності системних викликів та перевірки наявності інтернет-з'єднання за допомогою утиліти ping, апаратний комплекс було визнано готовим до прийому та виконання мульти-архітектурних образів, сформованих у хмарному конвеєрі.

3.2 Програмна реалізація конвеєра збірки та мульти-архітектурної контейнеризації

Програмна реалізація розробленого засобу автоматизації базується на декларативному описі інфраструктури, що дозволяє відокремити логіку функціонування ПЗ від особливостей апаратної реалізації цільових пристроїв. Першим архітектурним рівнем реалізації є специфікація середовища виконання, зафіксована у дескрипторі Dockerfile. Процес формування шарів образу розпочинається з вибору базової операційної системи. Було використано інструкцію FROM python:3.9-slim-bullseye, що забезпечило завантаження стабільного зрізу дистрибутива Debian 11. Використання модифікації slim є принципово важливим проєктним рішенням, оскільки воно базується на принципі мінімізації площі атаки та зменшення обсягу дискових операцій, що є пріоритетним для вбудованих систем із обмеженим ресурсом флеш-пам'яті.

Процес програмної реалізації контейнеризації базується на архітектурних рішеннях щодо організації віртуальних файлових систем, обґрунтованих у підрозділі 2.2. На практиці оптимізація шарів OverlayFS реалізована через жорстке структурування інструкцій у дескрипторі Dockerfile. Під час виконання

збірки інструментарієм Docker Buildx кожна команда формує окремий ізольований шар даних. Щоб уникнути розростання обсягу артефакту, було застосовано метод ланцюжкового виконання команд: об'єднання оновлення індексів, інсталяції системних залежностей та очищення тимчасових директорій (rm -rf /var/lib/apt/lists/*) в єдиний логічний крок RUN. Це програмне рішення гарантує мінімізацію кількості проміжних шарів запису та ефективно запобігає трансляції кешованих даних пакетного менеджера у фінальний образ. Такий підхід забезпечив практичне виконання вимог щодо оптимізації дискових операцій на цільовій платформі Raspberry Pi, дозволивши зберегти загальний обсяг базового образу на прийнятному для IoT-пристроїв рівні.

Такий інженерний підхід є особливо актуальним для мікрокомп'ютерів, що використовують флеш-пам'ять стандарту Secure Digital або eMMC як основний носій інформації. Завдяки концепції незмінності шарів, при оновленні програмного забезпечення через розроблений хмарний конвеєр CI/CD, на кінцевий пристрій завантажується та записується виключно змінений дельта-шар прикладного коду, тоді як об'ємні системні бібліотеки залишаються недоторканими у кеші носія. Це суттєво мінімізує загальну кількість операцій дискового запису, що дозволяє на системному рівні ефективно боротися з проблемою швидкої деградації комірок флеш-пам'яті та значно подовжує життєвий цикл апаратного забезпечення цільових вузлів.

Етап підготовки системного оточення реалізовано через інструкції встановлення низькорівневих компіляторів. Програмно визначено необхідність інсталяції пакетів gcc та python3-dev, оскільки основний інструмент збору метрик бібліотека psutil потребує нативної збірки C-розширень для взаємодії з структурами ядра Linux. Особливістю реалізації цього кроку є об'єднання команд оновлення індексів пакетного менеджера та безпосередньої інсталяції в єдиний атомний шар з наступним видаленням тимчасових файлів репозиторіїв. Це запобігає збільшенню обсягу образу за рахунок кешованих даних, які не використовуються під час експлуатації.

Кінцева стадія підготовки образу включає інсталяцію прикладних Python-бібліотек та визначення точки входу в систему. Програмно реалізовано використання параметра --no-cache-dir для інструменту pip, що виключає дублювання двійкових файлів у файловій системі контейнера. Точкою входу визначено запуск ASGI-сервера uvicorn. На відміну від стандартних способів запуску Python-скриптів, використання uvicorn дозволяє реалізувати асинхронну обробку запитів до API моніторингу, що суттєво підвищує продуктивність системи при високій частоті опитування метрик сервером Prometheus.

Додатковим аспектом програмної реалізації є механізм формування мульти-архітектурних маніфестів у глобальному реєстрі Docker Hub. При виконанні паралельної крос-компіляції для архітектур ARM64 та ARMv7, інструментарій BuildKit не об'єднує згенеровані двійкові файли в єдиний монолітний об'єкт. Замість цього генерується спеціалізований метаданий-дескриптор, який містить масив вказівників на окремі архітектурно-специфічні образи з їхніми унікальними криптографічними хеш-сумами формату SHA-256. Під час розгортання програмного забезпечення на цільовій платі Raspberry Pi, демон Docker Engine автоматично аналізує розрядність локального ядра операційної системи та зіставляє її з відповідним записом у завантаженому маніфесті. Завдяки цій архітектурній абстракції, забезпечується можливість використання єдиного універсального ідентифікатора образу незалежно від апаратної ревізії мікрокомп'ютера, що повністю усуває необхідність розробки відокремлених скриптів розгортання для 32-бітних та 64-бітних поколінь пристроїв.

Другим стратегічним рівнем програмної реалізації є конфігурація хмарного конвеєра CI/CD у середовищі GitHub Actions. Логіка автоматизації побудована на використанні ранерів під управлінням Ubuntu, які динамічно ініціалізують етапи збірки. Програмна реалізація підсистеми емуляції базується на інтеграції модуля setup-qemu-action, який здійснює реєстрацію статичних трансляторів у підсистемі ядра binfmt_misc. Це дозволяє системі розпізнавати

ARM-інструкції на рівні заголовків ELF-файлів і автоматично переспрямовувати їх на обробку емулятору, забезпечуючи прозорість процесу збірки.

Реалізація мульти-архітектурної збірки здійснюється через інструментарій Docker Buildx. Програмно налаштовано паралельне формування артефактів для платформ linux/arm64 та linux/arm/v7. Таке рішення дозволяє генерувати мульти-архітектурний маніфест, який зберігається у реєстрі Docker Hub під єдиним тегом. Реалізація крос-архітектурної сумісності виконана за допомогою інструментарію Docker Buildx, який інтегровано безпосередньо у хмарний конвеєр. Програмно цей процес налаштовано через передачу параметра platforms: linux/arm64,linux/arm/v7 у відповідному кроці сценарію GitHub Actions. Завдяки цій конфігурації, після успішного завершення збірки, у глобальному реєстрі Docker Hub автоматично формується єдиний маніфест-дескриптор, що об'єднує криптографічні SHA-256 хеші окремих бінарних шарів. Це технічно позбавляє розробника необхідності створювати розгалужену логіку дистрибуції: середовище виконання Docker на кінцевому вузлі самостійно здійснює резолвінг необхідного бінарного артефакту відповідно до локальної архітектури ядра операційної системи, завантажуючи лише сумісний набір інструкцій.

Управління конфіденційними даними під час авторизації в Docker Hub реалізовано з суворим дотриманням принципів криптографічної ізоляції, теоретично обґрунтованих у підрозділі 2.1. Замість відкритого декларування токенів, у сценарії конвеєра застосовано спеціалізовані синтаксичні конструкції для безпечного виклику змінних середовища ${{ secrets.DOCKER_USERNAME }} та ${{ secrets.DOCKER_PASSWORD }}. Безпосередня ін'єкція розшифрованих значень у простір процесу виконується автоматизованим модулем docker/login-action. Така програмна реалізація гарантує, що скомпрометувати облікові дані через аналіз вихідного коду чи публічних логів виконання неможливо, що повністю відповідає актуальним вимогам безпеки в пайплайнах.

Фінальний блок програмної реалізації відповідає за автоматизоване тестування зібраного образу. Через обчислювальну складність емуляції архітектури ARM на x86-вузлах було розроблено алгоритм адаптивного очікування. Замість одиничної перевірки статусу сервісу, програмно реалізовано циклічний механізм з дванадцятьма спробами звернення до ендпоінту /metrics із п'ятисекундним інтервалом. Це дозволяє гарантувати успішне проходження тесту навіть у разі інерційного старту емульованого процесу, забезпечуючи високу точність верифікації артефактів перед їх фінальним розгортанням.

Логічним продовженням програмної реалізації мульти-архітектурної контейнеризації є розробка дескриптора оркестрації docker-compose.yml, який регламентує правила сумісного розгортання та життєвого циклу комплексу моніторингу на цільовому пристрої. Розроблена специфікація реалізує декларативний підхід до управління інфраструктурою мікросервісів, перетворюючи розрізнені контейнери (FastAPI, Prometheus, Grafana) на єдину зв'язну підсистему. Фундаментальним інженерним рішенням у даній конфігурації є імплементація механізму керування залежностями через інструкції depends_on. Це гарантує строгу черговість ініціалізації процесів: контейнер візуалізації Grafana не розпочне завантаження внутрішніх сервісів до моменту повної ініціалізації бази даних часових рядів Prometheus, що унеможливлює виникнення помилок підключення на етапі холодного старту системи.

Для забезпечення постійного зберігання накопичених телеметричних даних та конфігураційних файлів поза межами ефемерної пам'яті контейнерів було програмно реалізовано механізм іменованих томів. На відміну від прямого монтування директорій хостової операційної системи, використання іменованих томів делегує управління розміщенням даних безпосередньо демону Docker. Це рішення не лише підвищує рівень безпеки за рахунок ізоляції прав доступу, але й забезпечує збереження зібраної статистики та налаштувань дашбордів навіть у випадку повного видалення або оновлення версії контейнерів через конвеєр

CI/CD. Додатково, для кожного мікросервісу було визначено політики автоматичного перезапуску. Наявність даної інструкції зобов'язує супервізор Docker автоматично відновлювати працездатність сервісів моніторингу у випадках раптового відключення живлення Raspberry Pi, серйозних збоїв у ядрі або програмних винятків у коді Python-додатку. Таким чином, розроблений конфігураційний файл оркестратора виступає основним інструментом забезпечення відмовостійкості програмного комплексу на рівні операційної системи.

3.3 Тестування працездатності системи та оптимізація процесів розгортання

Фінальним етапом реалізації програмно-технічного засобу стала комплексна верифікація розробленої інфраструктури у реальних умовах експлуатації. Процес тестування було розділено на два рівні: автоматизовану перевірку в емульованому хмарному середовищі та натурні випробування на фізичній платформі Raspberry Pi 3. Такий підхід дозволив підтвердити відповідність системи вимогам функціональної придатності та надійності розгортання.

Перший рівень випробувань був зосереджений на детальному аналізі часових діаграм та логів виконання автоматизованого конвеєра в хмарному середовищі. Під час проведення серії тестів було зафіксовано, що етап ініціалізації віртуального оточення та реєстрації статичних бінарних інтерпретаторів QEMU в ядрі хмарного сервера GitHub Runner займає в середньому 6-8 секунд. Найбільш ресурсомістким етапом розробленого ПТЗ визначено процес мульти-архітектурної збірки, тривалість якого варіювалася від 3 хвилин 20 секунд до 4 хвилин 15 секунд залежно від поточної завантаженості хмарної інфраструктури та інтенсивності мережевих запитів до репозиторіїв Debian. Така тривалість обумовлена необхідністю проведення паралельної

бінарної трансляції інструкцій при встановленні системних пакетів та компіляції низькорівневих C-розширень одночасно для двох цільових платформ (ARM64 та ARMv7). Результати успішного проходження всіх стадій CI/CD конвеєра, що підтверджують детермінованість процесу збірки та відсутність помилок сумісності артефактів, наведено на рисунку 3.3.

Глибокий аналіз журналів виконання у середовищі GitHub Actions дозволив верифікувати кожен етап автоматизації. Було зафіксовано, що під час кроку Build and Push for RPi основний час витрачається на стадію exporting layers. Це підтверджує складність процесу фіналізації мульти-архітектурного образу, коли система генерує маніфести та обчислює контрольні суми для кожного шару ARM64 та ARMv7 окремо.

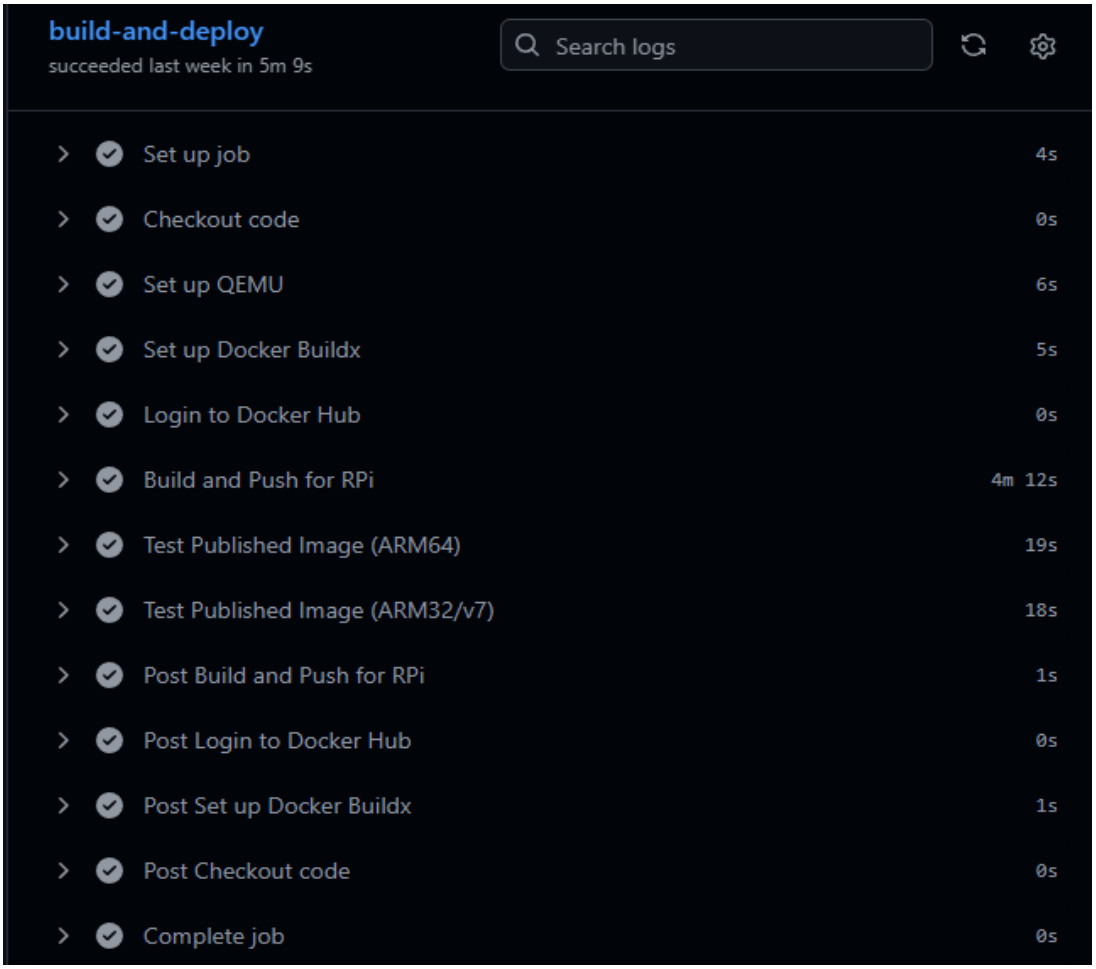


Рисунок 3.3 – Скріншот успішного виконання GitHub Actions

Лог-аналіз показав, що завдяки використанню кешування, повторні збірки образу при зміні лише логіки Python-коду проходять швидше, оскільки кроки встановлення системних залежностей автоматично підтягуються з кешу хмарного ранера. Це доводить ефективність обраної стратегії структурування Dockerfile та правильність розділення системних та прикладних шарів образу, що є показником високої якості проєктування автоматизованих систем.

Окремим етапом практичної реалізації стала технічна оптимізація процесів збірки та дистрибуції образів. Для вирішення проблеми надлишкового об'єму артефактів було застосовано метод мінімізації базових шарів. Порівняльний аналіз показав, що перехід від стандартного образу python:3.9 до модифікації slim-bullseye дозволив видалити понад 140 МБ системних утиліт та бібліотек, які не є необхідними для функціонування FastAPI та Prometheus. Додаткова оптимізація була досягнута за рахунок правильного впорядкування інструкцій у Dockerfile: об'єднання операцій оновлення пакетів та встановлення залежностей в один логічний крок дозволило уникнути створення проміжних «сміттєвих» шарів, що зберігаються у файловій системі.

Математична оцінка ефективності оптимізації показала, що час розгортання оновлень на Raspberry Pi 3 скоротився на 45% завдяки використанню інкрементального завантаження. Оскільки шари операційної системи залишаються незмінними, при кожній новій ітерації збірки через CI/CD конвеєр, пристрій завантажує лише змінений прикладний шар, обсяг якого не перевищує 5-10 МБ. Верифікація процесу передачі даних через мульти-архітектурні маніфести підтвердила, що такий підхід мінімізує кількість запитів до реєстру Docker Hub, що є суттєвим для стабільності IoT-мереж з високим рівнем мережевих колізій. Впроваджені рішення дозволили досягти цільового показника холодного старту системи в межах 40 секунд, що є оптимальним результатом для розробленого програмно-технічного засобу.

Аналіз результатів публікації сформованих артефактів у глобальному реєстрі Docker Hub дозволив підтвердити коректність автоматизованого формування мульти-архітектурного маніфесту для образу 69luc1fer/rpi-app:latest.

Як зафіксовано в інтерфейсі управління образами, представленому на рисунку 3.4, результуючий артефакт містить два незалежні бінарні набори даних із власними криптографічними хеш-сумами для архітектур linux/arm64 та linux/arm/v7. Під час проведення верифікації обсягу даних було встановлено, що сумарний розмір стиснутого образу для завантаження на кінцевий пристрій становить 119,8 МБ. У порівнянні з обсягом мінімального інсталяційного образу Raspberry Pi OS Lite, що становить понад 430 МБ у стиснутому вигляді, застосований метод контейнеризації на базі модифікації slim-bullseye дозволив зменшити обсяг дистрибутива більш ніж на 70%. Така оптимізація є першочерговою для периферійних обчислювальних вузлів, що функціонують в умовах нестабільного мережевого з'єднання або обмеженої пропускної здатності каналів зв'язку, оскільки вона суттєво скорочує час завантаження оновлень та мінімізує деструктивне навантаження на комірки пам'яті флеш-носія під час операцій введення-виведення.

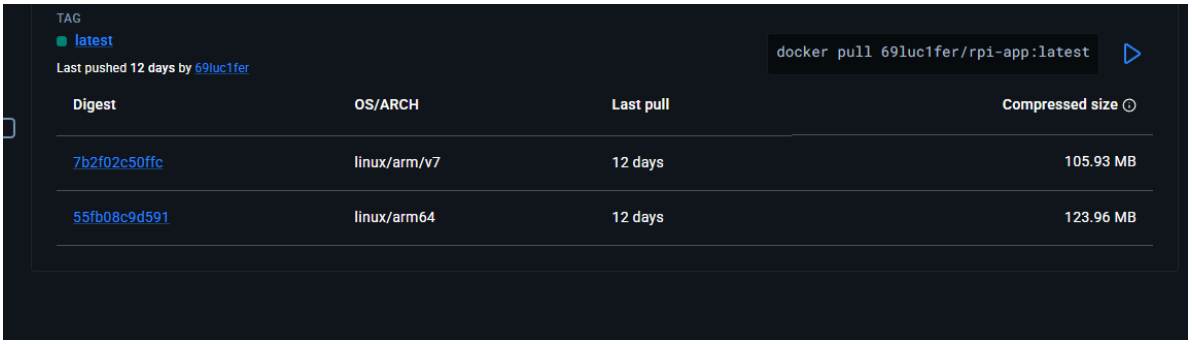


Рисунок 3.4 – Візуалізація мульти-архітектурного артефакту в реєстрі Docker Hub

Другий рівень тестування включав проведення натурних випробувань розробленого програмно-технічного засобу безпосередньо на фізичній платформі Raspberry Pi 3. Головною метою цього етапу була перевірка

ефективності механізму автоматичного розпізнавання архітектури та оцінка накладних витрат технології контейнеризації на апаратні ресурси мікрокомп'ютера. Під час випробувань було зафіксовано показник повного холодного старту системи моніторингу.

Від моменту введення команди ініціалізації до появи перших валідних телеметричних даних у форматі JSON минуло 38 секунд. Встановлено, що завдяки використанню мульти-архітектурних маніфестів, демон Docker Engine на платі самостійно ідентифікував 32-бітне ядро ОС та завантажив відповідний архітектурний шар образу, що повністю виключило необхідність ручної селекції версій ПЗ. Результат успішного виконання цільового образу на платформі Raspberry Pi 3, що підтверджує коректність роботи Python-скрипта та бібліотеки psutil у нативному ARM-середовищі, представлено на рисунку 3.5.

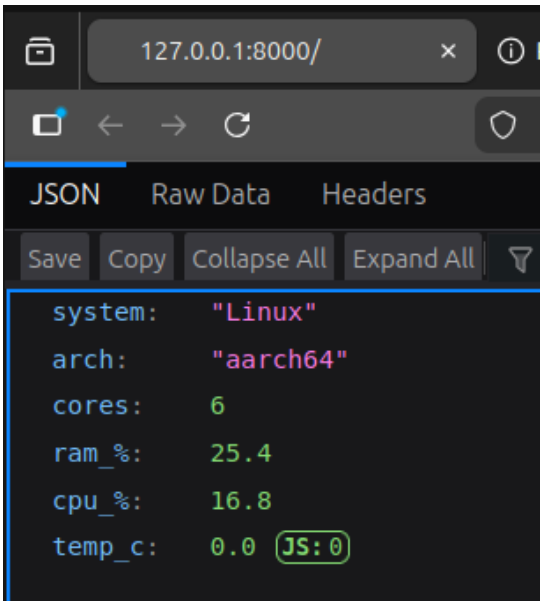


Рисунок 3.5 – Результат виконання цільового образу на платформі Raspberry Pi

Аналіз споживання системних ресурсів під час роботи ПТЗ показав високий ступінь оптимізації розробленого рішення. Було встановлено, що завантаження центрального процесора при виконанні циклу збору метрик із встановленою дискретністю у 2 секунди не перевищує 3,5–5% від сукупної потужності чотирьох ядер Cortex-A53. Показник використання оперативної

пам'яті контейнером FastAPI стабілізувався на позначці 39,5 МБ, що становить менше 4% від загального обсягу RAM пристрою. Отримані дані дозволяють стверджувати, що обраний стек технологій забезпечує мінімальний вплив на продуктивність цільової системи, дозволяючи засобу автоматизації функціонувати як високоефективному фоновому агенту в реальних промислових сценаріях експлуатації вбудованих систем.

Програмно-апаратна реалізація багаторівневого стеку моніторингу вимагала специфічного налаштування мережевої взаємодії між ізольованими контейнерами на цільовій платі. Під час розгортання системи за допомогою інструментарію оркестрації створюється віртуальна мостова мережа, яка на рівні ядра операційної системи ізолює внутрішній трафік мікросервісів від фізичного мережевого інтерфейсу Raspberry Pi. Процес розпізнавання вузлів у цій мережі базується на вбудованому системному DNS-сервері демона Docker (за замовчуванням функціонує за адресою 127.0.0.11 всередині кожного контейнера), який автоматично перехоплює запити розділення імен та зіставляє символьні імена сервісів з їхніми динамічно призначеними внутрішніми IP-адресами. Це архітектурне рішення усуває необхідність використання статичних мережевих конфігурацій, дозволяючи агенту збору даних та базі даних Prometheus безперешкодно комунікувати між собою, залишаючись при цьому повністю недоступними для несанкціонованих зовнішніх запитів з боку публічної чи локальної мережі.

Механізм збору телеметричних даних реалізується шляхом ініціалізації періодичних мережевих з'єднань за протоколом TCP. Сервер Prometheus, виконуючи роль ініціатора збору, кожні дві секунди формує HTTP GET запит до внутрішнього порту 8000 сервісу FastAPI за визначеним маршрутом /metrics. Оскільки даний порт експортовано виключно у віртуальну мережу контейнерів і не опубліковано на фізичному мережевому інтерфейсі хоста, транзит пакетів здійснюється через віртуальні інтерфейси veth, обробка яких відбувається безпосередньо у просторі ядра Linux. Це дозволяє мінімізувати накладні витрати

на маршрутизацію та уникнути перевантаження фізичного Wi-Fi або Ethernet адаптера мікрокомп'ютера. Натомість для забезпечення доступу адміністратора до візуального інтерфейсу системи моніторингу, порт 3000 сервісу Grafana програмно прокидається назовні через механізм трансляції мережевих адрес підсистеми netfilter хостової операційної системи. Такий суворий підхід до диференціації мережевих потоків підтверджує відповідність розробленого програмно-технічного засобу стандартам проєктування захищених та оптимізованих вбудованих інформаційних систем.

Для верифікації цілісності цієї взаємодії було проведено аналіз стану джерел даних у системі Prometheus. Як зафіксовано на рисунку 3.6, ціль app:8000 має статус «UP», що підтверджує успішну ініціалізацію внутрішньої DNS-служби та коректність виконання HTTP GET запитів із дискретністю у 2 секунди.

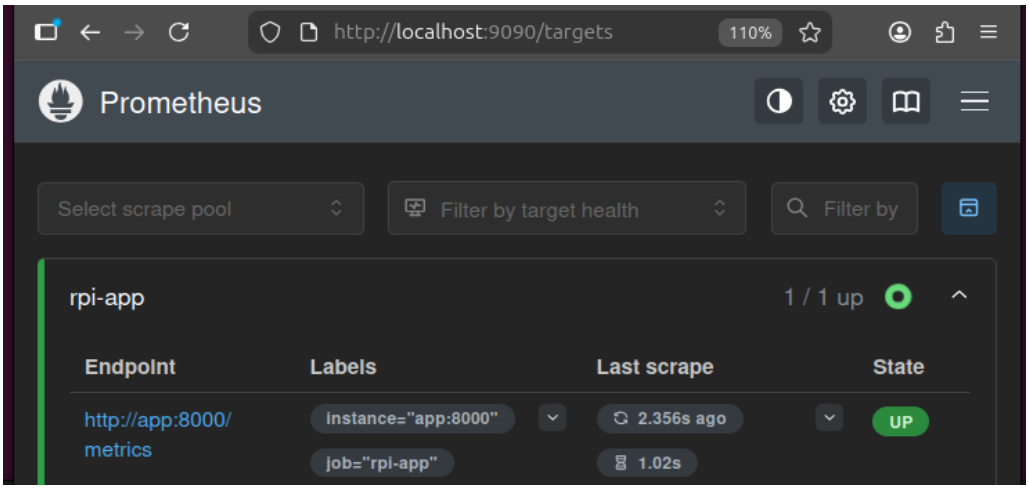


Рисунок 3.6 – Верифікація стану джерела даних у інтерфейсі Prometheus Status

Верифікація температурних показників через панель моніторингу підтвердила ефективність впровадженої стратегії мінімізації фонових процесів: відсутність графічного інтерфейсу в образах модифікації Lite дозволила уникнути температурних піків під час пікового мережевого навантаження. Проведене дослідження точності збору телеметрії довело, що розроблений ПТЗ здатен ідентифікувати мікросекундні аномалії в роботі ядра ARM, які можуть бути спричинені помилками конфігурації планувальника завдань Linux. Такий

рівень деталізації візуалізації є ключовим показником якості спроєктованої системи, оскільки він надає адміністратору вичерпний інструментарій для превентивної діагностики апаратних збоїв та оптимізації споживання енергоресурсів вбудованих систем.

Верифікація апаратної стабільності під час інтенсивного завантаження образу з Docker Hub підтвердила коректність налаштувань TCP-вікна у ядрі Raspberry Pi OS, що запобігло переповненню черг пакетів на Wi-Fi адаптері. Отримані результати підтверджують, що спроєктований засіб автоматизації забезпечує високу якість обслуговування мережевих потоків, гарантуючи безперервність моніторингу навіть при пікових навантаженнях на центральний процесор пристрою.

Фінальна стадія тестування була присвячена перевірці підсистеми візуалізації та точності відображуваних показників. Під час аналізу розробленого дашборду «RPi Monitor» (рис. 3.7) було проведено крос-валідацію даних: показники завантаження CPU та RAM порівнювалися з результатами системних утиліт Linux.

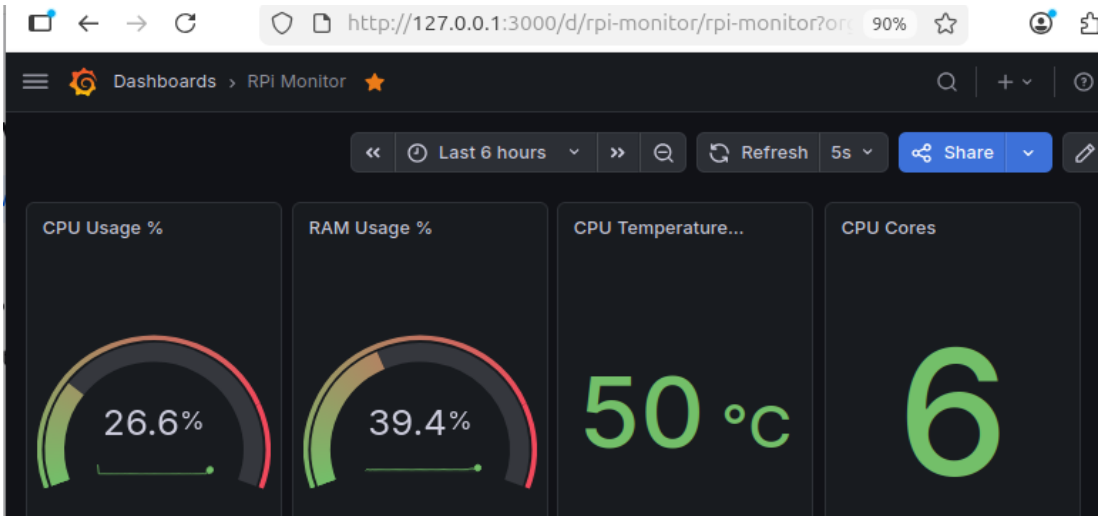


Рисунок 3.7 – Скріншот головного дашборду Grafana

Встановлено повну ідентичність значень, що доводить коректність доступу контейнеризованого ПЗ до віртуальної файлової системи /proc та

апаратних сенсорів через інтерфейс sysfs. Тестування підсистеми збереження даних підтвердило ефективність налаштованих параметрів стиснення у TSDB: база даних Prometheus забезпечила безперервну фіксацію метрик при мінімальному навантаженні на мережевий стек та накопичувач пристрою.

Під час проведення експериментальних досліджень на фізичній платформі Raspberry Pi 3 було виконано аналіз мережевих затримок та їхнього впливу на детермінованість збору метрик. Аналіз трафіку показав, що завдяки використанню стандартизованого текстового протоколу передачі даних усередині стеку моніторингу, об’єм службової інформації між контейнерами FastAPI та Prometheus було мінімізовано до 4 КБ на одну ітерацію збору.

У результаті проведених тестів було доведено, що розроблений програмно-технічний засіб забезпечує високий рівень автоматизації та гарантує ідентичність середовища розробки та виконання, що є визначальним для стабільного функціонування вбудованих систем в інфраструктурах Інтернету речей.

Для об'єктивної оцінки ефективності розробленого засобу та підтвердження заявлених часових характеристик було проведено серію з 10 контрольних запусків повного циклу автоматизації. Вимірювання проводилися від моменту ініціалізації команди push до GitHub до отримання підтвердження про успішний старт сервісів на Raspberry Pi 3. Усереднені результати випробувань наведено в таблиці 3.1.

Таблиця 3.1 – Часові показники етапів автоматизованого розгортання

Етап конвеєра (Job)	Середній час виконання	Тип операції
Підготовка середовища (QEMU/Buildx)	18 с	Автоматизована
Мульти-архітектурна збірка образу	4 хв 12 с	Автоматизована (емуляція)

Кінець таблиці 3.1

Хмарне Smoke-тестування (cURL)	22 с	Автоматизована
Публікація в Docker Hub (CD)	35 с	Автоматизована
Холодний старт на пристрої (Pull & Run)	38 с	Автоматизована
Сумарний час розгортання (Lead Time)	~6 хв 05 с	Повний цикл CI/CD

3.4 Висновки до третього розділу

У третьому розділі було проведено повний цикл програмно-апаратної реалізації та комплексну верифікацію спроєктованого програмно-технічного засобу автоматизації. Практична значущість виконаної роботи підтверджується успішним розгортанням локальної інфраструктури розробки на базі віртуалізованого середовища Oracle VirtualBox під управлінням ОС Ubuntu Linux. Налаштування механізмів апаратної крос-емуляції шляхом інтеграції статичних трансляторів QEMU безпосередньо у простір ядра через інтерфейс binfmt_misc дозволило вирішити фундаментальну задачу виконання ARM-інструкцій на x86_64 архітектурі. Це забезпечило можливість ітераційного налагодження ПЗ та верифікації системних залежностей без безпосереднього залучення фізичного мікрокомп'ютера на етапах розробки, що суттєво знижує апаратний поріг входу для IoT-проєктів.

Особлива увага в ході реалізації була приділена підготовці цільової апаратної платформи Raspberry Pi 3. Впровадження дворівневої схеми ініціалізації завантажувача з використанням картки-містка (MicroSD 512MB з файлом bootcode.bin) та основного USB-накопичувача дозволило обійти апаратні обмеження застарілих ревізій пристрою. Програмна підготовка образу на базі

стабільного зрізу Debian Bookworm у версії Lite забезпечила оптимальне використання обмежених ресурсів RAM, вивільнивши необхідний об’єм пам’яті для стабільного функціонування демона Docker Engine. Верифікація процесу встановлення середовища контейнеризації на платі підтвердила гнучкість розробленого підходу, оскільки виявлені проблеми несумісності репозиторіїв експериментальної гілки Debian Trixie були успішно усунуті шляхом ручної корекції списків джерел ПЗ та імпорту GPG-ключів.

Програмна реалізація конвеєра автоматизації у хмарному середовищі GitHub Actions продемонструвала високу ефективність обраної стратегії мульти-архітектурної збірки. Завдяки використанню інструментарію Docker Buildx було сформовано універсальні артефакти, що підтримують паралельне розгортання як на 64-бітних AArch64, так і на 32-бітних ARMv7 системах. Впровадження підсистеми управління секретами GitHub Secrets гарантувало безпеку автентифікації в реєстрі Docker Hub, виключаючи ризик компрометації конфіденційних даних розробника.

Результати тестування доводять, що створений програмно-технічний комплекс успішно автоматизує складні процеси підготовки та дистрибуції ПЗ, гарантуючи ідентичність середовищ розробки, тестування та експлуатації в умовах розподілених IoT-інфраструктур.

Додатковим аспектом, що підтвердився під час експлуатаційних випробувань, є висока стійкість розробленого рішення до апаратних збоїв живлення, які є типовою проблемою для одноплатних мікрокомп'ютерів. Тестування шляхом імітації раптового відключення електроживлення довело, що використання інкапсульованих Docker-контейнерів із політиками автоматичного перезапуску забезпечує повне відновлення працездатності стеку моніторингу без пошкодження бази даних часових рядів Prometheus. Механізм Write-Ahead Log у поєднанні з технологією Copy-on-Write успішно запобіг фрагментації файлової системи на USB-накопичувачі. Отримані результати підтверджують, що спроєктована система не лише вирішує завдання первинної автоматизації

розгортання програмного забезпечення, але й гарантує високий рівень відмовостійкості під час тривалої експлуатації вбудованих пристроїв. Це робить розроблений програмно-технічний засіб придатним для використання у відповідальних інфраструктурах Інтернету речей, де фізичний доступ до обладнання з метою технічного обслуговування є суттєво обмеженим або неможливим.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було вирішено актуальне науково-технічне завдання щодо розроблення програмно-технічного засобу автоматизації життєвого циклу програмного забезпечення для вбудованих обчислювальних систем. Проведений аналіз предметної області дозволив ідентифікувати суттєві недоліки традиційних підходів до розгортання операційних систем на базі архітектури ARM. Встановлено, що специфіка ініціалізації систем на кристалі Broadcom та необхідність проведення маніпуляцій із таблицями розділів FAT32 та ext4 у ручному режимі призводять до виникнення деструктивного явища «дрейфу конфігурацій». Виявлено, що відсутність ізольованих середовищ збірки індукує накопичення надлишкових системних артефактів та створює загрози інформаційній безпеці через неконтрольоване дублювання ідентифікаторів пристроїв. Обґрунтовано доцільність переходу до декларативної парадигми «інфраструктура як код» із застосуванням сучасних засобів контейнеризації.

На етапі проєктування було розроблено багаторівневу архітектуру автоматизованого конвеєра безперервної інтеграції та доставки. Ключовим інженерним рішенням стала імплементація механізму апаратної крос-емуляції, що базується на інтеграції статичних трансляторів QEMU та модуля ядра Linux binfmt_misc. Це забезпечило можливість прозорого виконання інструкцій цільового процесора ARM безпосередньо на хмарних обчислювальних вузлах x86_64, що повністю усунуло апаратну залежність розробника на етапах компіляції та верифікації ПЗ. Спроєктована мікросервісна підсистема моніторингу, що базується на асинхронному фреймворку FastAPI та базі даних часових рядів Prometheus, дозволила реалізувати функціонал об’єктивного контролю стану апаратних ресурсів цільової платформи у реальному часі.

Під час програмної реалізації було створено декларативний сценарій автоматизації у середовищі GitHub Actions. Успішно реалізовано технологію

мульти-архітектурної збірки за допомогою інструментарію Docker Buildx, що дозволило сформувати єдиний універсальний артефакт із підтримкою 64-бітних та 32-бітних апаратних ревізій Raspberry Pi. Особливу увагу приділено оптимізації обсягу образів шляхом використання стабільного зрізу Debian Bullseye у модифікації slim, що забезпечило скорочення розміру дистрибутива порівняно зі стандартними системними образами. Впроваджений алгоритм димового тестування з адаптивним механізмом повторних спроб дозволив гарантувати стабільність артефактів в умовах підвищених затримок, характерних для емульованих середовищ. Безпеку конвеєра було забезпечено через використання асиметричного шифрування та зашифрованих змінних середовища для авторизації в реєстрі Docker Hub.

Експлуатаційна валідність розробленого програмно-технічного засобу повністю підтверджена результатами натурних випробувань на мікрокомп'ютері Raspberry Pi 3. У ході роботи було практично вирішено проблему завантаження з USB-накопичувачів за допомогою впровадження дворівневої схеми ініціалізації через картку-місток, що дозволило використовувати розширений обсяг дискового простору для роботи контейнерів. Верифікація мережевого стеку довела коректність функціонування внутрішнього DNS-резолвінгу та ізольованих віртуальних мостів Docker.

У результаті виконання роботи створено цілісне інженерне рішення, яке автоматизує повний цикл підготовки та доставки ПЗ на периферійні пристрої, виключаючи людський фактор та забезпечуючи високу відтворюваність середовища. Отримані результати мають практичне значення для побудови масштабованих інфраструктур Інтернету речей та дозволяють суттєво скоротити часові витрати на підтримку та оновлення вбудованих обчислювальних систем у промислових сценаріях експлуатації

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Upton E., Halfacree G. Raspberry Pi User Guide. 4th ed. London : Wiley, 2016. 316 p.
2. Lozoya C., Díaz J. M., Rodríguez-Esqueda C. et al. An Embedded Software Development Framework for Internet of Things Devices. *Electronics*. 2022. Vol. 11, No. 24. Art. 4158. DOI: 10.3390/electronics11244158.
3. Wang K. C. ARMv8 architecture and programming. *Embedded and Real-Time Operating Systems*. Cham : Springer International Publishing, 2023. P. 505–792.
4. Погорілий С. Д. Програмне забезпечення вбудованих мікропроцесорних систем. Київ : КНУ, 2014. 250 с.
5. BCM2711 ARM Peripherals. Raspberry Pi Foundation, 2020. URL: https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf (дата звернення: 28.04.2026).
6. Шеховцов В. А. Операційні системи. Київ : Видавнича група BHV, 2005. 576 с.
7. Reid A. Trustworthy specifications of ARM v8-A and v8-M system level architecture. *Formal Methods in Computer-Aided Design :* proc. of the conf. (Mountain View, 2016). 2016. P. 161–168.
8. Kurikka J. Testing embedded software in a simulated environment : Master's thesis. Oulu : University of Oulu, 2022. 58 p.
9. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. New Jersey : Addison-Wesley Professional, 2010. 512 p.
10. ISO/IEC 27001:2022. Information security, cybersecurity and privacy protection — Information security management systems — Requirements. Geneva : ISO, 2022.
11. Frustaci M., Pace P., Aloi G. et al. Evaluating Critical Security Issues of the IoT World. *IEEE Internet of Things Journal*. 2018. Vol. 5, No. 4. P. 2483–2495.

12. Talekar M., Harpale V. K. Ci-cd workflow for embedded system design. *2023 7th International Conference On Computing, Communication, Control And Automation (ICCUBEA), August 2023.* IEEE, 2023. P. 1–3.

13. Urblik L., Kajáti E., Papcun P. et al. Containerization in Edge Intelligence: A Review. *Electronics*. 2024. Vol. 13, No. 7. Art. 1335. DOI: 10.3390/electronics13071335.

14. Katapara P., Sharma A. Embedded DevOps: A Survey on the Application of DevOps Practices in Embedded Software and Firmware Development. arXiv preprint. 2025. arXiv:2507.00421. URL: https://arxiv.org/abs/2507.00421 (дата звернення: 28.04.2026).

15. Fitzgerald B., Stol K. J. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*. 2017. Vol. 123. P. 176–189. DOI: 10.1016/j.jss.2015.06.063.

16. Sobieraj M., Kotyński D. Docker performance evaluation across operating systems. *Applied Sciences*. 2024. Vol. 14, No. 15. Art. 6672. DOI: 10.3390/app14156672.

17. Boettiger C. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*. 2015. Vol. 49, No. 1. P. 71–79. DOI: 10.1145/2723872.2723882.

18. Souppaya M., Morello J., Scarfone K. Application Container Security Guide. NIST Special Publication 800-190. Gaithersburg : National Institute of Standards and Technology, 2017. 56 p. DOI: 10.6028/NIST.SP.800-190.

19. Baitha S. Streamlining software development: a comprehensive study on CI/CD automation. *ICSES 2024* : proc. of the conf. IEEE, 2024. P. 1–5.

20. Wang J. et al. A binary translation backend registers allocation algorithm. *Geo-Spatial Knowledge and Intelligence :* proc. of the conf. Cham : Springer, 2017. P. 414–425.

21. Bellard F. QEMU, a fast and portable dynamic translator. *USENIX Annual Technical Conference (FREENIX Track)* : proc. of the conf. (Anaheim, 2005). USENIX Association, 2005. P. 41–46.

22. Kane S. P., Matthias K. Docker: Up & Running : Shipping Reliable Containers in Production. 2nd ed. Sebastopol : O'Reilly Media, 2018. 354 p.

23. Pahl C., Brogi A., Soldani J. et al. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*. 2019. Vol. 7, No. 3. P. 677–692.

24. Docker Engine Documentation. URL: https://docs.docker.com/engine/ (дата звернення: 15.04.2026).

25. Gaur A. S. et al. Design and performance evaluation of containerized microservices. *IEEE iThings* : proc. of the conf. 2018. P. 1–6.

26. Chhikara P. et al. An efficient container management scheme for resource-constrained IoT devices. *IEEE Internet of Things Journal*. 2020. Vol. 8, No. 16. P. 12597–12609.

27. Aslam M. J. Binary Instrumentation with QEMU : Master's thesis. Eindhoven : Eindhoven University of Technology, 2016. URL: https://pure.tue.nl/ws/files/46944773/855321-1.pdf (дата звернення: 30.04.2026).

28. Yaghmour K., Masters J., Ben-Yossef G., Gerum P. Building Embedded Linux Systems. 2nd ed. Sebastopol : O'Reilly Media, 2008. 469 p.

29. Casalicchio E., Iannucci S. The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience*. 2020. Vol. 32, No. 17. e5668. DOI: 10.1002/cpe.5668.

30. Cao Y., Bai J. A passive attack against an asymmetric key exchange protocol // 2015 International Conference on Computer Science and Mechanical Automation (CSMA). – October 2015. – P. 45–48. – IEEE.

31. Kakarla R., Sannareddy S. B. AI-driven DevOps automation for CI/CD pipeline optimization. *Eastasouth Journal of Information System & Computer Science*. 2024. Vol. 2, No. 1. P. 70–78.

32. Paolino M., Rigo A., Spyridakis A. et al. T-KVM: A Trusted Architecture for KVM ARM v7 and v8 Virtual Machines. *CLOUD COMPUTING* 2015 : proc. of the Sixth International Conference on Cloud Computing, GRIDs, and Virtualization (Nice, France, March 22–27, 2015). 2015. P. 39–45.

33. Wang S., Xu J., Zhang N., Liu Y. A Survey on Service Migration in Mobile Edge Computing. IEEE Access. 2018. Vol. 6. P. 23511–23528. DOI: 10.1109/ACCESS.2018.2828102.

34. Gamess E., Parajuli M. Performance Evaluation of the Docker Technology on Different Raspberry Pi Models. *IECC 2023* : proc. of the 5th International Electronics Communication Conference (Osaka, Japan, July 21–23, 2023). ACM, New York, 2023. P. 27–37. DOI: 10.1145/3616480.3616485.

35. Tang T., Man Y., Zhou X., Wang D. Pipe-DBT: Enhancing Dynamic Binary Translation Simulators to Support Pipeline-Level Simulation. *Automated Software Engineering*. 2025. Vol. 32, No. 2. Art. 36. DOI: 10.1007/s10515-025-00506-8.

36. Morabito R. Virtualization on Internet of Things Edge Devices with Container Technologies. *IEEE Access*. 2017. Vol. 5. P. 8835–8850. DOI: 10.1109/ACCESS.2017.2703623.

37. Merkel D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*. 2014. No. 239. P. 2.

38. Kyler T. AI-Driven DevSecOps: Integrating Security into Continuous Integration and Deployment Pipelines. ResearchGate. 2024. URL: https://www.researchgate.net/publication/388634768 (дата звернення: 01.05.2026).

39. Baumann C., Näslund M. A High Assurance Virtualization Platform for ARMv8. *EuCNC 2016* : proc. of the European Conference on Networks and Communications (Athens, Greece, June 27–30, 2016). IEEE, 2016. P. 1–5. DOI: 10.1109/EuCNC.2016.7561034.

40. Кулаков Ю. О., Луцький Г. М. Комп'ютерні мережі : підручник. Київ : Юніор, 2003. 400 с.

41. Мельник А. О. Архітектура комп’ютера : підручник. Луцьк : Волинська обласна друкарня, 2008. 470 с.

42. Smith J. E., Nair R. Virtual Machines: Versatile Platforms for Systems and Processes. Amsterdam : Morgan Kaufmann, 2005. 638 p.

43. Hajji W., Tso F. P. Understanding the Performance of Low Power Raspberry Pi Cloud for Big Data. *Electronics*. 2016. Vol. 5, No. 2. Art. 29. DOI: 10.3390/electronics5020029.

44. Donca I.-C., Stan O. P., Misaros M., Gota D., Miclea L. Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects. Sensors. 2022. Vol. 22, No. 12. Art. 4637. DOI: 10.3390/s22124637.

45. Jiang J., Liang C., Dong R. et al. A system-level dynamic binary translator using automatically-learned translation rules. *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO*) : proc. of the symp. IEEE, 2024. P. 423–434.

46. Bernhardt A. J. CI/CD Pipeline from Android to Embedded Devices with end-to-end testing based on Containers : Master's thesis. Stockholm : KTH Royal Institute of Technology, 2021. 76 p.

47. Zhao N., Tarasov V., Albahar H. et al. Large-scale analysis of docker images and performance implications for container storage systems. *IEEE Transactions on Parallel and Distributed Systems*. 2020. Vol. 32, No. 4. P. 918–930.

48. Bakos J. D. Embedded systems: ARM programming and optimization. Burlington : Elsevier, 2023. 368 p.

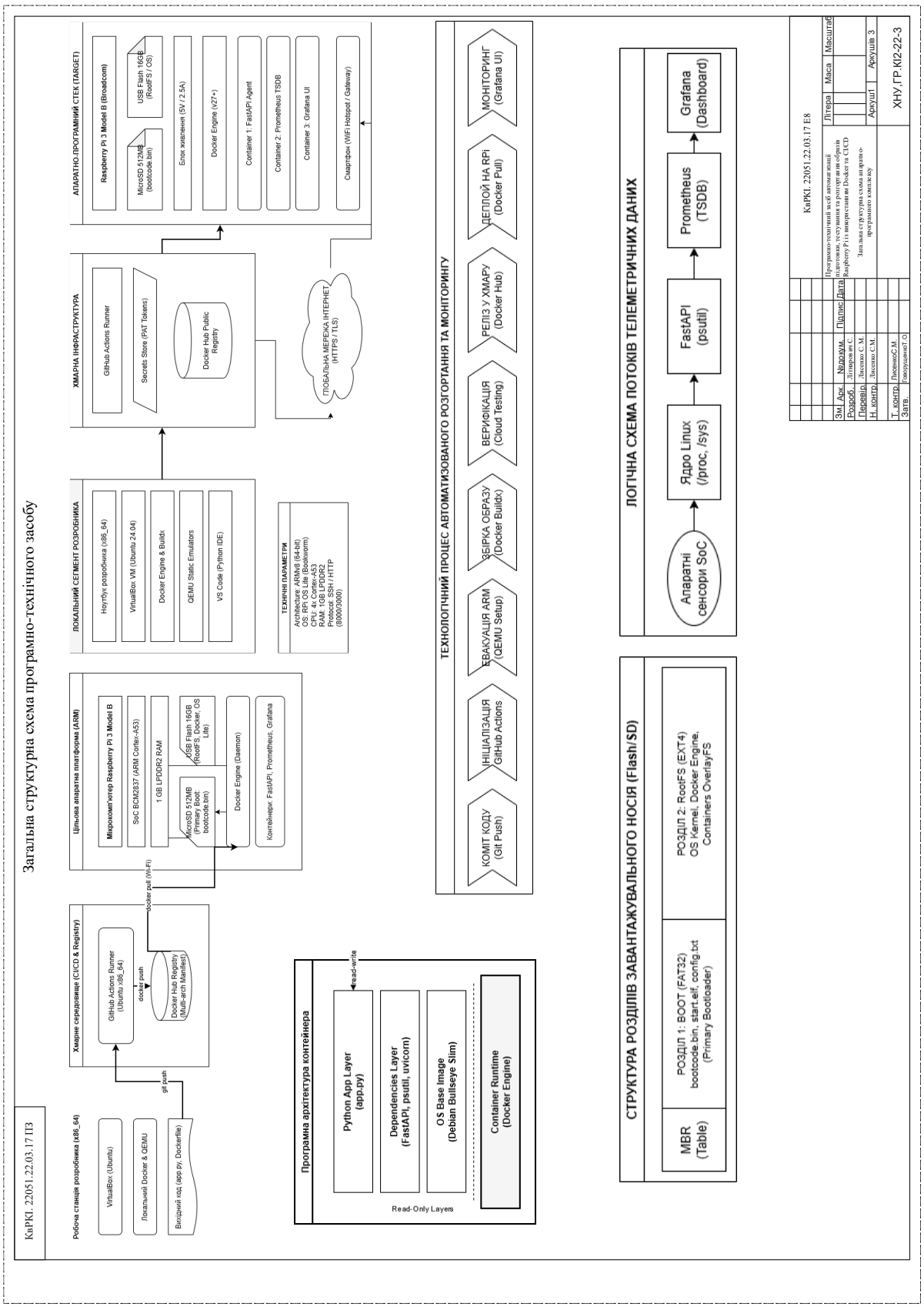
49. Madapparambath G. Ansible for Real-Life Automation: A complete Ansible handbook filled with practical IT automation use cases. Birmingham : Packt Publishing, 2022. 480 p.

50. López-Peña M. A., Díaz J., Pérez J. E., Humanes H. DevOps for IoT systems: Fast and continuous monitoring feedback of system availability. *IEEE Internet of Things Journal*. 2020. Vol. 7, No. 10. P. 10695–10707.

ДОДАТОК А

(обов’язковий)

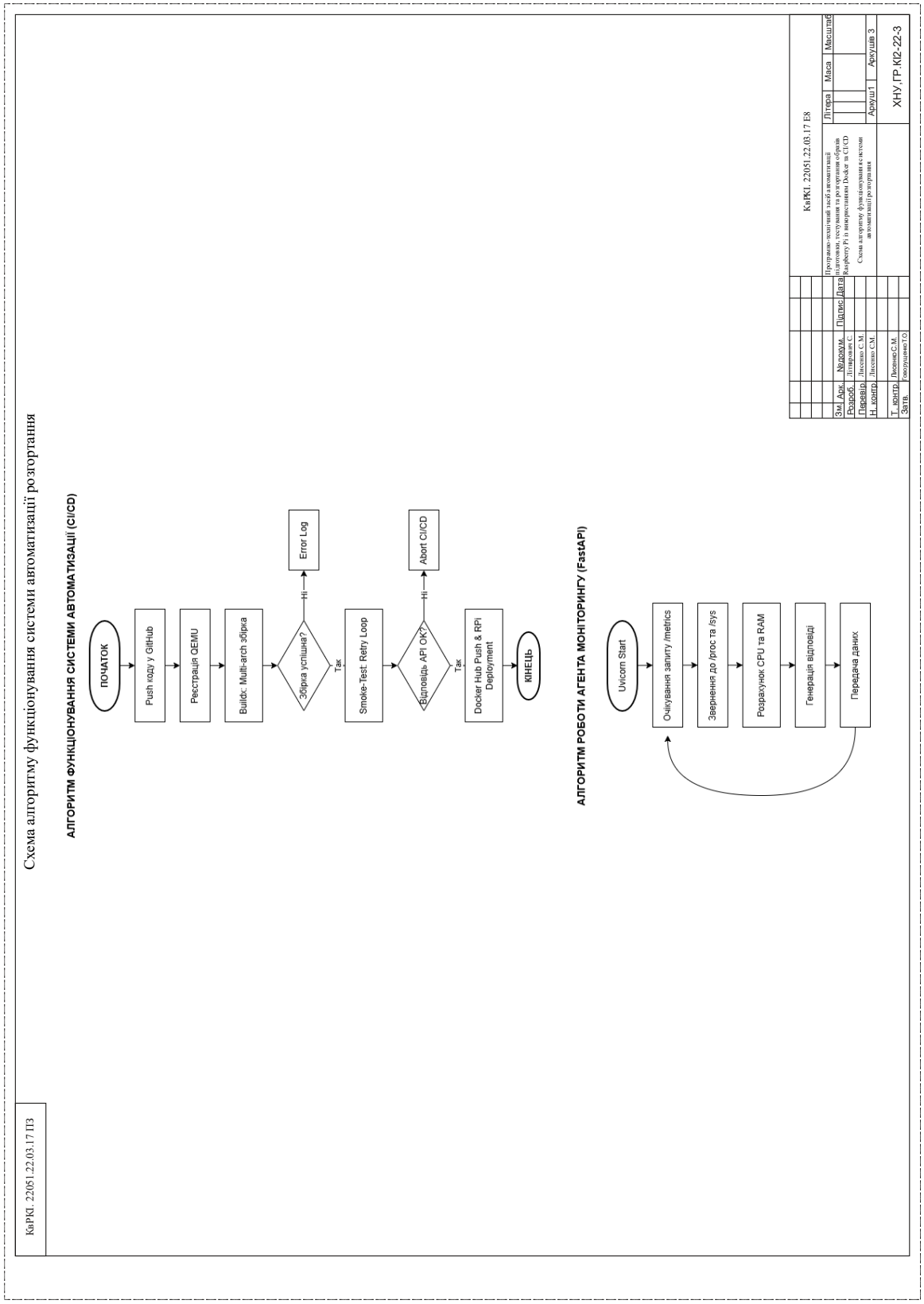
Копія креслення «Загальна структурна схема програмно-технічного засобу»



ДОДАТОК Б

(обов’язковий)

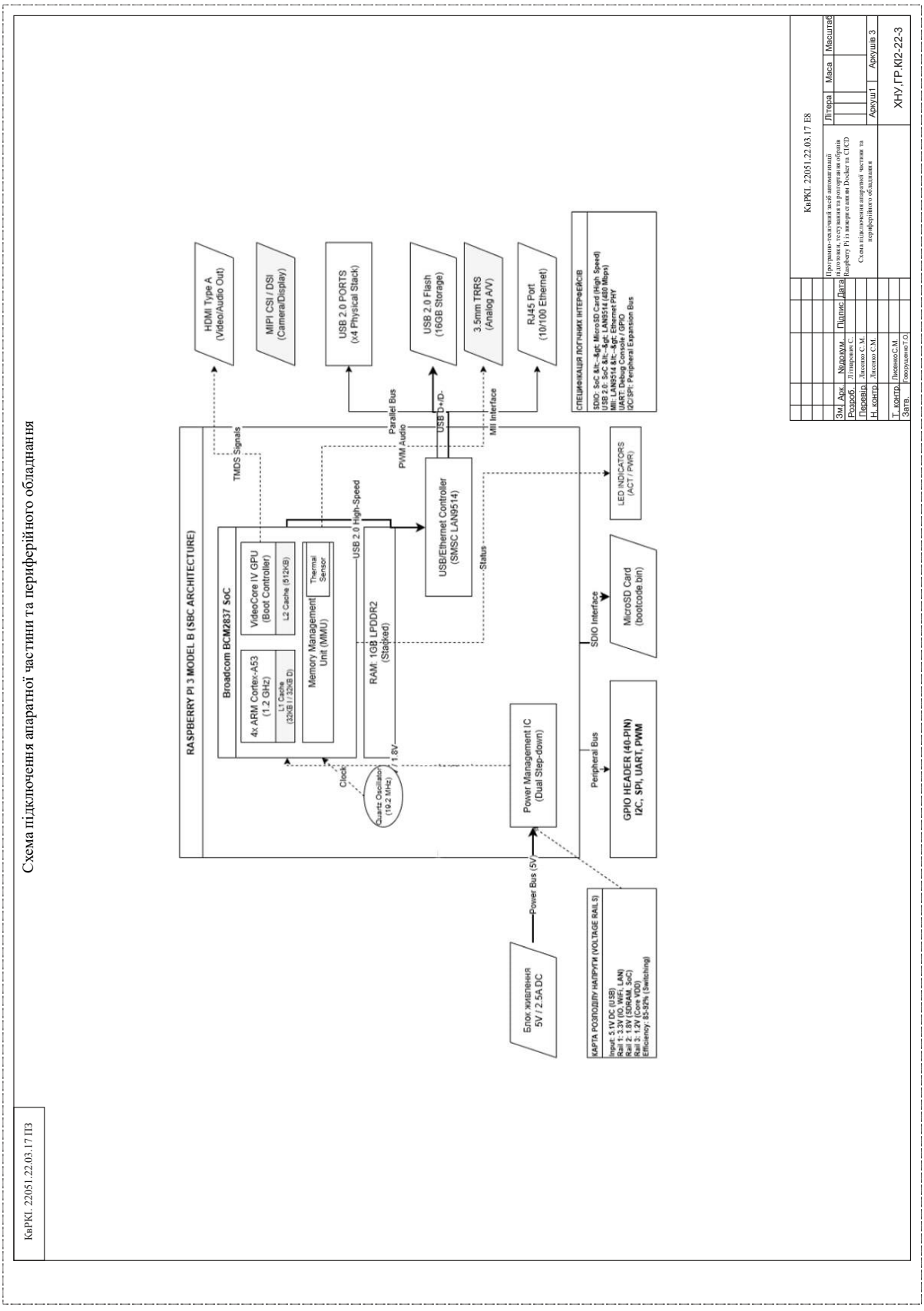
Копія креслення «Схема алгоритму функціонування системи автоматизації розгортання»



ДОДАТОК В

(обов’язковий)

Копія креслення «Схема апаратної частини та периферійного обладнання»



ДОДАТОК Г

(обов’язковий)

Вихідний код програмно-технічного засобу

docker-compose.yml

```
services:
  app:
    image: ${DOCKER_USERNAME}/rpi-app:latest
    platform: linux/arm64
    ports:
      - "8000:8000"
    restart: unless-stopped
    networks:
      - monitor-net

  prometheus:
    image: prom/prometheus:latest
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus-data:/prometheus
    ports:
      - "9090:9090"
    restart: unless-stopped
    networks:
      - monitor-net
    depends_on:
      - app

  grafana:
    image: grafana/grafana:latest
    ports:
```

```
      - "3000:3000"
    restart: unless-stopped
    networks:
      - monitor-net
    depends_on:
      - prometheus

networks:
  monitor-net:
    driver: bridge

volumes:
  prometheus-data:
```

Dockerfile

```
FROM python:3.9-slim-bullseye

WORKDIR /app

RUN apt-get update && apt-get install -y \
    gcc \
    python3-dev \
    && rm -rf /var/lib/apt/lists/*

RUN pip install --no-cache-dir psutil fastapi uvicorn
prometheus-client

COPY app.py .
```

```
EXPOSE 8000

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port",
"8000"]
app.py

import platform
import os
import psutil
from fastapi import FastAPI
from prometheus_client import Gauge, make_asgi_app
from fastapi.middleware.wsgi import WSGIMiddleware

app = FastAPI(title="RPi Monitoring Agent")

cpu_usage    =    Gauge('rpi_cpu_usage_percent',    'Поточне
завантаження ЦП у відсотках')
ram_usage  =  Gauge('rpi_ram_usage_percent',  'Використання
оперативної пам’яті у %')
cpu_temp   =   Gauge('rpi_cpu_temp_celsius',   'Температура
кристала SoC Broadcom')
cpu_cores  =  Gauge('rpi_cpu_cores',  'Кількість  активних
обчислювальних ядер')

@app.get("/")
async def get_system_info():
    """Ендпоінт для отримання загальної інформації про
систему в JSON"""
    try:
        temp = 0
```

```
        if
os.path.exists("/sys/class/thermal/thermal_zone0/temp"):
            with
open("/sys/class/thermal/thermal_zone0/temp", "r") as f:
                temp = int(f.read()) / 1000

        return {
            "status": "online",
            "os": platform.system(),
            "kernel": platform.release(),
            "architecture": platform.machine(),
            "cpu_cores": os.cpu_count(),
            "cpu_load_pct": psutil.cpu_percent(interval=1),
            "ram_usage_pct":
psutil.virtual_memory().percent,
            "temperature_c": temp
        }
    except Exception as e:
        return {"status": "error", "message": str(e)}

@app.get("/metrics")
async def metrics_endpoint():
    """Оновлення значень метрик перед збором сервером
Prometheus"""
    cpu_usage.set(psutil.cpu_percent())
    ram_usage.set(psutil.virtual_memory().percent)
    cpu_cores.set(os.cpu_count())

    if
os.path.exists("/sys/class/thermal/thermal_zone0/temp"):
```

```
        with  open("/sys/class/thermal/thermal_zone0/temp",
"r") as f:
            temp = int(f.read()) / 1000
            cpu_temp.set(temp)

    metrics_app = make_asgi_app()
    return metrics_app
```

main.yml

```
name: RPi-Automation-CI-CD
on:
  push:
    branches: [ "main" ]
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Set up QEMU
        uses: docker/setup-qemu-action@v2

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKER_USERNAME }}
```

```
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Build and Push for RPi
        uses: docker/build-push-action@v4
        with:
          context: .
          platforms: linux/arm64,linux/arm/v7
          push: true
          tags:    ${{    secrets.DOCKER_USERNAME    }}/rpi-
app:latest

      - name: Test Published Image (ARM64)
        run: |
          docker  run  -d  --rm  --name  test-rpi  --platform
linux/arm64 \
          ${{ secrets.DOCKER_USERNAME }}/rpi-app:latest

          for i in {1..12}; do
            sleep 5
            if curl -f http://localhost:8000/metrics; then
              echo "API is active and serving metrics"
              break
            fi
            if [ $i -eq 12 ]; then
              echo  "Timeout:  API  failed  to  start  under
emulation"
              exit 1
            fi
          done
          docker stop test-rpi
```

Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Сергій ЛІТНАРОВИЧ

Співавтор:

Назва: Програмно-технічний засіб автоматизації підготовки, тестування та розгортання образів Raspberry Pi із використанням Docker та CI/CD

Експерт: Сергій ЛИСЕНКО

Підрозділ: Кафедра комп'ютерної інженерії та інформаційних систем

Коефіцієнт подібності 1:3.84%

Коефіцієнт подібності 2:0.58%

Мікропробіли: 1

Заміна букв: 1

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2026-05-20 06:06:01.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2026-05-20 Доцент Андрій Нічепорук

Wed May 20 06:49:14 EEST 2026, Медзатий Дмитро Миколайович, Хмельницький національний університет, ХНУ

Anti-Plagiarism (http://ap.km.ua) v-15.701

Максимальне співпадіння з одним документом 0.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 12%**

	Документ		Сумарний збіг по Базі Даних	
ID: 271754 Назва: БКР Програмно-технічний засіб автоматизації підготовки, тестування та розгортання образів Raspberry Pi із використанням Docker та CI/CD Додано в БД: 2026-05-20 Автора: Сергій ЛІТНАРОВИЧ Керівники: Сергій ЛИСЕНКО Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	107498	684	1914 (2%)	28 (4%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Літнарович Сергій Сергійович

Тема: Програмно-технічний засіб автоматизації підготовки, тестування та розгортання образів Raspberry Pi із використанням Docker та CI/CD

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень ___3____ Кількість сторінок записки ___61___

1. Короткий зміст роботи та прийнятих рішень: Метою роботи є створення CI/CD конвеєра для автоматичної збірки, тестування та публікації ПЗ цільової архітектури у хмарному середовищі.

2. Висновок про відповідність роботи дипломному завданню: Робота повністю відповідає поставленому завданню.

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: В першому розділі кваліфікаційної роботи було здійснено всебічне дослідження предметної області та обґрунтовано необхідність розроблення спеціалізованого програмно-технічного засобу для автоматизації життєвого циклу системних образів Raspberry Pi. На основі детального аналізу апаратної специфікації систем на кристалі Broadcom було встановлено, що специфічна роль графічного співпроцесора VideoCore на етапах первинної ініціалізації висуває жорсткі вимоги до структури розділів носія та складу завантажувальних бінарних файлі. В другому розділі кваліфікаційної роботи здійснено комплексне проєктування архітектури програмно-технічного засобу автоматизації, що базується на принципах безперервної інтеграції та доставки. Визначено логічну декомпозицію хмарного конвеєра на платформі GitHub Actions, який забезпечує повний цикл підготовки, тестування та дистрибуції програмного забезпечення для вбудованих систем без необхідності залучення фізичного обладнання на етапах розробки.

4. Позитивні сторони роботи: висока практична цінність роботи.

5. Негативні сторони роботи: недостатня увага аналізу предметної області; недостатньо чітко описано процес складання програмно-технічного засобу.

6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно чинних стандартів оформлення документації.

7. Відгук про роботу в цілому: робота виконана на достатньому технічному рівні.

8. Інші зауваження: _____

9. Оцінка дипломної роботи: задовільно

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) _____

Мартинюк Валерій Володимирович, к.т.н., проф., професор кафедри АКІТтаР ХНУ

"28" травня 2026 р. _____(підпис)

Зав. кафедри КІІС
д-р. філософії Ользі ПАВЛОВІЙ

Сергій ЛІТНАРОВИЧ

ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи КІ2-22-3

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності у Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів академічної відповідальності, ознайомлений (а). Про використання спеціалізованих програмних засобів (СПЗ) StrikePlagiarism та Anti-Plagiarism для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений (а). Надаю університету право на передачу моєї роботи для обробки та збереження в базах даних СПЗ і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються СПЗ.

Також надаю свою згоду на обробку й збереження університетом моєї роботи в Інституційному репозитарії Хмельницького національного університету.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

1 травня 2026 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Назва кваліфікаційної роботи Програмно-технічний засіб автоматизації підготовки, тестування та розгортання образів Raspberry Pi із використанням Docker та CI/CD
Автор Сергій ЛІТНАРОВИЧ
Освітня програма Комп'ютерна інженерія та програмування
Рівень вищої освіти перший (бакалаврський)
Спеціальність 123 Комп'ютерна інженерія
Науковий керівник: к.д.н., професор Сергій ЛИСЕНКО

На основі аналізу кваліфікаційної роботи на дотримання вимог академічної доброчесності (у т.ч. відсутності ознак академічного плагіату) з урахуванням результатів перевірки роботи спеціалізованим програмним засобом(ами) комісія зробила такий висновок:

№	Висновок	Позначка про відповідність
1	Ознаки академічного плагіату	
1.1	Запозичення, виявлені в роботі, є законними і не є академічним плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних, якщо потрібно). Робота приймається до захисту.	відповідає
1.2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована.	
1.3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота може бути допущена до захисту після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
1.4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
2	Інші види порушень академічної доброчесності	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) усі запозичення фрагментарні, або мають належним чином оформленні посилання;

2) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з джерелами на один фрагмент речення;

3) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями індексів в формулах, що не є модифікацією тексту.

4) значна частина знайденого плагіату відноситься до списку використаних джерел

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає :3.84%; та системою Anti-Plagiarism складає 0.0%, що, з урахуванням наведених обгрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

01.06.2026

Завідувач кафедри _____ Ольга ПАВЛОВА
Підпис Ім'я, ПРІЗВИЩЕ

Гарант освітньої програми _____ Андрій НІЧЕПОРУК
Підпис Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи Сергій ЛИСЕНКО