МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Навчально-науковий інститут інноваційних освітніх технологій
Кафедра інформаційно-обчислювальних систем і управління

Шлапак Марта Сергіївна

Контейнерний стек для ARM-вузлів з безперервним розгортанням для IoT-пристроїв / Container Stack for ARM Nodes with Continuous Deployment for IoT Devices

Спеціальність 122 – Комп’ютерні науки
Освітньо-професійна програма – Комп’ютерні науки

Кваліфікаційна робота

Виконала студентка групи КНз-41
М. С. Шлапак

Науковий керівник:
к.т.н., доцент, О.Р.Осолінський

Кваліфікаційну роботу допущено до захисту
«___»_____ 2026 р.

Завідувач кафедри
_____ Н.В. Дзюбановська

Тернопіль – 2026

Факультет комп'ютерних інформаційних технологій
Навчально-науковий інститут інноваційних освітніх технологій
Освітній ступінь «бакалавр»
спеціальність 122 – Комп'ютерні науки
освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Н.В. Дзюбановська
«____» _____20__ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ
Шлапак Марта Сергіївна
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: Контейнерний стек для ARM-вузлів з безперервним розгортанням для IoT-пристроїв / Container Stack for ARM Nodes with Continuous Deployment for IoT Devices
керівник роботи О.Р. Осолінський, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджений наказом по університету від 01 грудня 2025 року № 894.
2. Строк подання студентом закінченої кваліфікаційної роботи 25 травня 2026 р.
3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити:
- провести аналіз предметної області та вимог до контейнерного розгортання на ARM-вузлах;
- провести огляд і аналіз існуючих рішень;
- сформувати постановку задачі;
- розробити алгоритмічне та інформаційне забезпечення контейнерного стеку для ARM-вузлів;
- розробити архітектури системи
- розробити архітектуру системи;
- розробити алгоритмічне забезпечення модуля;
- розробити інформаційне забезпечення;
- провести вибір та аналіз програмно-апаратних засобів для ARM-вузла;
- реалізувати програмно-технологічне забезпечення ARM;
- провести тестування системи.

5. Перелік графічного матеріалу в роботі:

– детальна архітектура модуля OTA-оновлення;

– діаграма розподілу повного циклу CI/CD та OTA-оновлення Контейнерного сервісу.

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

7. Дата видачі завдання 01 грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Затвердження теми кваліфікаційної роботи, ознайомлення з літературними джерелами та складання плану роботи	до 01.01.2026 р.	
2	Написання 1 розділу кваліфікаційної роботи	до 01.02.2026 р.	
3	Написання 2 розділу кваліфікаційної роботи	до 15.03.2026 р.	
4	Написання 3 розділу кваліфікаційної роботи	до 01.05.2026 р.	
5	Представлення попереднього варіанту кваліфікаційної роботи, перевірка та внесення змін керівником	до 15.05.2026 р.	
6	Опрацювання зауважень та представлення завершеного варіанту кваліфікаційної роботи. Підготовка супроводжуючих документів	до 25.05.2026 р.	
7	Перевірка кваліфікаційної роботи на оригінальність тексту	до 30.05.2026 р.	
8	Оформлення кваліфікаційної роботи та отримання допуску до захисту	до 10.06.2026 р.	
9	Подання кваліфікаційної роботи до захисту на засіданні атестаційної комісії	до 10.06.2026 р.	

Студентка_____ М. С. Шлапак
(підпис) (прізвище та ініціали)

Керівник кваліфікаційної роботи_____ О.Р.Осолінський
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи: 64 с., 31 рис., 4 додатки, 35 джерел.

Метою кваліфікаційної роботи є розробка контейнерного стеку для ARM-вузлів з безперервним розгортанням для IoT-пристроїв, що забезпечує віддалене оновлення контейнеризованих сервісів, перевірку їх працездатності та автоматичне повернення до попередньої стабільної версії у разі помилки.

Об’єкт дослідження: процеси розгортання та супроводу контейнеризованих сервісів на ARM edge-вузлі.

Предмет дослідження: методи, алгоритми та програмні засоби організації оновлення контейнеризованих сервісів на ARM edge-вузлі з контролем працездатності, журналюванням результатів і автоматичним відновленням після невдалого оновлення.

Методи дослідження: аналіз технологій контейнеризації та оркестрації, проєктування архітектури контейнерного стеку, тестування сценаріїв успішного оновлення, пропуску оновлення.

Результатом роботи є розроблений прототип контейнерного стеку для ARM-вузла на базі Raspberry Pi. В роботі реалізовано механізм віддаленого оновлення контейнеризованого сервісу за допомогою сценарію оновлення, публікацію багатоплатформних контейнерних образів у віддаленому реєстрі, запуск сервісів через засоби опису контейнерного стеку, періодичне виконання оновлення за допомогою системного таймера, перевірку працездатності сервісу через HTTP-інтерфейс та збереження результатів у локальній базі даних.

Практичне значення роботи полягає в тому, що запропонований підхід дозволяє оновлювати сервіси IoT-вузла без фізичного доступу до пристрою та забезпечує фіксацію результатів кожної спроби розгортання.

Ключові слова: IoT, ARM-ВУЗОЛ, EDGE-COMPUTING, DOCKER, DOCKER COMPOSE, КОНТЕЙНЕРНИЙ СТЕК, OTA-ОНОВЛЕННЯ, RASPBERRY PI, MULTI-ARCH, SYSTEMD, SQLITE, HEALTH-CHECK, ROLLBACK.

ANNOTATION

Explanatory note to the qualification thesis: 64 pages, 31 figures, 4 annexes, 35 references.

The purpose of the qualification thesis is to develop a container stack for ARM nodes with continuous deployment for IoT devices, which provides remote updating of containerized services, verification of their operability, and automatic return to the previous stable version in case of an error.

Object of research: the processes of deployment and maintenance of containerized services on an ARM edge node.

Subject of research: methods, algorithms, and software tools for organizing the updating of containerized services on an ARM edge node with operability control, result logging, and automatic recovery after a failed update.

Research methods: analysis of containerization and orchestration technologies, design of the container stack architecture, testing of successful update and update skip scenarios.

The result of the work is a developed prototype of a container stack for an ARM node based on Raspberry Pi. The work implements a mechanism for remote updating of a containerized service using an update script, publication of multi-platform container images in a remote registry, service launch through container stack description tools, periodic update execution using a system timer, service operability verification through an HTTP interface, and storage of results in a local database.

The practical significance of the work is that the proposed approach allows updating IoT node services without physical access to the device and provides recording of the results of each deployment attempt.

Keywords: IoT, ARM NODE, EDGE COMPUTING, DOCKER, DOCKER COMPOSE, CONTAINER STACK, OTA UPDATE, RASPBERRY PI, MULTI-ARCH, SYSTEMD, SQLITE, HEALTH CHECK, ROLLBACK.

ЗМІСТ

ВСТУП

IoT-пристрої та edge-вузли вже широко використовуються в системах моніторингу, розумного дому, промислових датчиках і невеликих локальних сервісах. Дуже часто такі вузли працюють на ARM-платформах, тому що вони є досить дешевими та енергоефективними, що дозволяє їм працювати цілодобово. Але такі пристрої мають менш потужний процесор, невеликий обсяг пам’яті, microSD-носій замість повноцінного диска, а також ризик нестабільного інтернету чи живлення. Через це ручне оновлення програмного забезпечення стає незручним і ризикованим. Одна помилка може зламати сервіс і залишити пристрій без робочого застосунку до моменту ручного втручання.

Тому зараз актуальною є тема контейнерного підходу та безперервного розгортання для ARM-вузлів. Контейнери забезпечують однакове середовище запуску та зменшують залежність сервісу від конкретних налаштувань операційної системи. Якщо сервіс зібраний у контейнер то його простіше переносити, відтворювати і оновлювати. Для IoT-систем важливо оновити контейнер і проконтролювати весь процес розгортання. Важливо також мати перевірку працездатності після оновлення і мати механізм відкату, якщо нова версія не запускається або не відповідає. В реальній експлуатації автоматичний відкат зменшує час простою, так як повертає попередню робочу версію без втручання адміністратора.

Актуальність теми дослідження. Зі збільшенням кількості IoT-вузлів зростає потреба у простому й надійному механізмі віддаленого оновлення через мережу (OTA-оновлення). Особливо це актуально для ARM-платформ, де немає запасних ресурсів і де важливо не допустити ситуації, коли після оновлення сервіс не працює. Тому потрібен підхід, який забезпечує повторюване розгортання програмного забезпечення, ведення журналу подій і контроль результатів оновлення.

Метою кваліфікаційної роботи є розробка контейнерного стеку для ARM-вузлів з безперервним розгортанням для IoT-пристроїв

Для досягнення мети роботи необхідно виконати такі завдання:

– провести аналіз предметної області та вимог до контейнерного розгортання на ARM-вузлах;

– провести огляд і аналіз існуючих рішень;

– сформувати постановку задачі;

– розробити алгоритмічне та інформаційне забезпечення контейнерного стеку для ARM-вузлів;

– розробити архітектуру системи

– розробити алгоритмічне забезпечення модуля;

– розробити інформаційне забезпечення;

– провести вибір та аналіз програмно-апаратних засобів для ARM-вузла;

– реалізувати програмно-технологічне забезпечення ARM;

– провести тестування системи.

Об’єктом дослідження є процеси розгортання та супроводу контейнеризованих сервісів на ARM edge-вузлі.

Предмет дослідження — методи, алгоритми та програмні засоби організації оновлення сервісів на ARM edge-вузлі

Практичне значення одержаних результатів полягає у створенні прототипу контейнерного стеку для ARM-вузла, який можна використати як основу для IoT-систем. Реалізований підхід дозволяє без фізичного доступу оновлювати сервіси та автоматично відновлювати робочий стан після невдалого оновлення. Це спрощує експлуатацію IoT-вузлів і зменшує ризики простою сервісів під час оновлень.

Результати кваліфікаційної роботи апробовані та опубліковані в матеріалах X Міжнародної студентської наукової конференції «Діджиталізація науки як виклик сьогодення» (м. Миколаїв, Україна).

Кваліфікаційна робота складається із вступу, трьох розділів, висновків, списку використаних джерел та додатків.

1. АНАЛІЗ ПРЕДИЕТНОЇ ОБЛАСТІ ТА ВИМОГ ДО КОНТЕЙНЕРНОГО РОЗГОРТАННЯ НА ARM-ВУЗЛАХ

1.1 Особливості edge-IoT інфраструктури на ARM та вимоги до контейнерного стеку

За останні роки кількість пристроїв IoT зросла до мільярдів пристроїв, що призвело до значного збільшення передачі даних через мережу. Це в свою чергу поклало надмірне навантаження на існуючу хмарну інфраструктуру. Щоб вирішити цю проблему були запропоновані периферійні обчислення (edge), які передбачають перенесення частини обробки ближче до джерела даних. Так як периферійні обчислення та IoT можна ефективно поєднати було запропоновано створити нову екосистему edge-IoT, яка покращує автономність, зменшує затримки та дає можливість обслуговувати локалізовані розумні застосунки навіть за нестабільного зв’язку з хмарою [1].

Для edge-рівня часто використовують ARM-вузли, тому що вони енергоефективні, недорогі та придатні для тривалої автономної роботи, хоча поступаються серверним системам за ресурсами. Це означає, що процесор, оперативна пам’ять, microSD-носій і мережевий канал можуть стати вузькими місцями під час роботи сервісів [3]. Також важливо враховувати реальні умови експлуатації пристрою який працюватиме без постійного фізичного доступу, з ризиком короткочасних відключень живлення, перебоями зв’язку та високими затримками. Такі умови ускладнюють супровід програмного забезпечення, так як оновлення має бути передбачуваним, відтворюваним і безпечним, тому що помилка може спричинити простій або потребу ручного відновлення. [1]. На рисунку 1.1 зображено принцип оновлення прошивки.

Рисунок 1.1 - Основні етапи процесу оновлення прошивки

У цьому контексті контейнеризація розглядається як практичний підхід до уніфікації доставки та запуску прикладних сервісів на ARM-вузлі. Контейнерний підхід дозволяє ізолювати компоненти, стандартизувати середовище виконання та зменшити залежність від ручної конфігурації ОС. В даній роботі розглядається не один контейнер, а контейнерний стек де узгоджено набір сервісів разом з параметрами запуску, мережевою взаємодією, конфігураціями та даними які зберігаються між перезапусками.

Незважаючи на переваги, edge-IoT має деякі обмеження, які потрібно враховувати під час проектування системи. Однією з основних проблем є різнорідність пристроїв і протоколів, які мають працювати в межах одного середовища. Також виникає потреба в узгодженні комунікацій та обчислень. Перенесення обробки ближче до джерела даних зменшує затримки, але потребує правильного розподілу задач між edge-вузлом і хмарною інфраструктурою. Крім того ускладнюється питання безпеки і приватності, оскільки компоненти будь-якої ланки куди входить IoT пристрій або edge-вузол може створити загрозу для всієї системи. Усі ці чинники визначають вимоги до програмної платформи та механізмів її подальшого супроводу [2].

Стандартний сценарій для роботи ARM edge-вузла полягає у використанні його як локального шлюзу, що запускає сервіси збору та попередньої обробки даних. На вузлі розгортається контейнерний стек де окремі сервіси виконують функції прийому або зчитування телеметрії, нормалізації та агрегування

локального буферного зберігання і передачі даних у зовнішню систему [3]. За умов нестабільного інтернет-з'єднання ключовою вимогою є можливість автономної роботи вузла, тимчасового локального накопичення даних та їх подальшої передавання після відновлення зв'язку. Це зменшує втрати та робить систему стійкішою до реальних умов експлуатації.

З врахуванням наведених особливостей вимоги до контейнерного стеку на ARM edge-вузлі узагальнено в таблиці 1.1.

Таблиця 1.1 - Вимоги до контейнерного стеку на ARM edge-вузлі

Фактор в edge-IoT на ARM	Ризик / наслідок	Вимога до контейнерного стеку
Нестабільний інтернет, затримки	Втрата даних/затримка доставки	Буферизація даних, повторні спроби передавання, контроль черг та пакетне надсилання.
Обмежена RAM/CPU	Взаємні просідання сервісів	Легкі контейнерні образи, обмеження використання ресурсів та мінімальна базова конфігурація сервісів
microSD/повільний I/O	Деградація при логуванні/БД	Мінімізація кількості записів на диск, ротація журналів та раціональне зберігання службових даних
Периферія	Надмірне навантаження на операції I/O, затримки в ланцюжку обробки даних	Пріоритет I/O, оптимальні налаштування контейнерів
Холодний старт контейнерів	Різке зростання часу відповіді сервісу після перезапуску	Початкова ініціалізація сервісу, перевірка працездатності, контроль залежностей і порядку запуску
Відсутність фізичного доступу	Дороге/повільне відновлення	Віддалене керування: логи/статус/перезапуск/версії

OTA без перевірок	Ризик шкідливого/битого оновлення	Перевірка автентичності/цілісності, журнал версій
Помилка оновлення	Простій сервісів	Перевірка працездатності після оновлення та автоматичне повернення до попередньої стабільної версії
Варіативність мережі в контейнерах	Зменшується пропускна здатність і зростає затримка	Вибір оптимального режиму мережевої взаємодії та тестування роботи контейнерів під навантаженням

1.2 Огляд і аналіз існуючих рішень

Найпоширенішим інструментом контейнеризації є Docker. Docker часто розглядають як фактичний стандарт, так як він пропонує зрозумілу модель запуску застосунку разом з його залежностями в ізольованому контейнері. Завдяки цьому сервіс легше переносити між різними вузлами. Для edge це критично оскільки пристрої можуть відрізнятися за апаратним забезпеченням, версіями ОС або налаштуваннями а розгортання має бути максимально однаковим. Додатково Docker має перевагу завдяки великій екосистемі, оскільки для нього доступна значна кількість готових образів, прикладів використання та напрацьованих підходів до автоматизованого збирання, тестування і розгортання програмного забезпечення [4]. Крім того під час використання Docker потрібно враховувати питання безпеки. В класичній схемі демон і контейнери часто працюють з правами root а на периферійних пристроях це небажано, бо будь-яка вразливість потенційно дає зловмиснику ширші можливості впливу на систему. Для зменшення цього ризику існує режим Docker Rootless. У цьому режимі Docker і контейнери можуть працювати без прав адміністратора, а ізоляція процесів забезпечується за допомогою окремого простору імен користувачів. Це підсилює безпеку бо навіть якщо контейнер буде скомпрометований йому значно важче отримати повний контроль над ОС. Такий режим ускладнює налаштування, оскільки потребує

додаткових утиліт і правильно заданих діапазонів ідентифікаторів користувачів [5]. На рисунку 1.2 зображено архітектура роботи Docker.

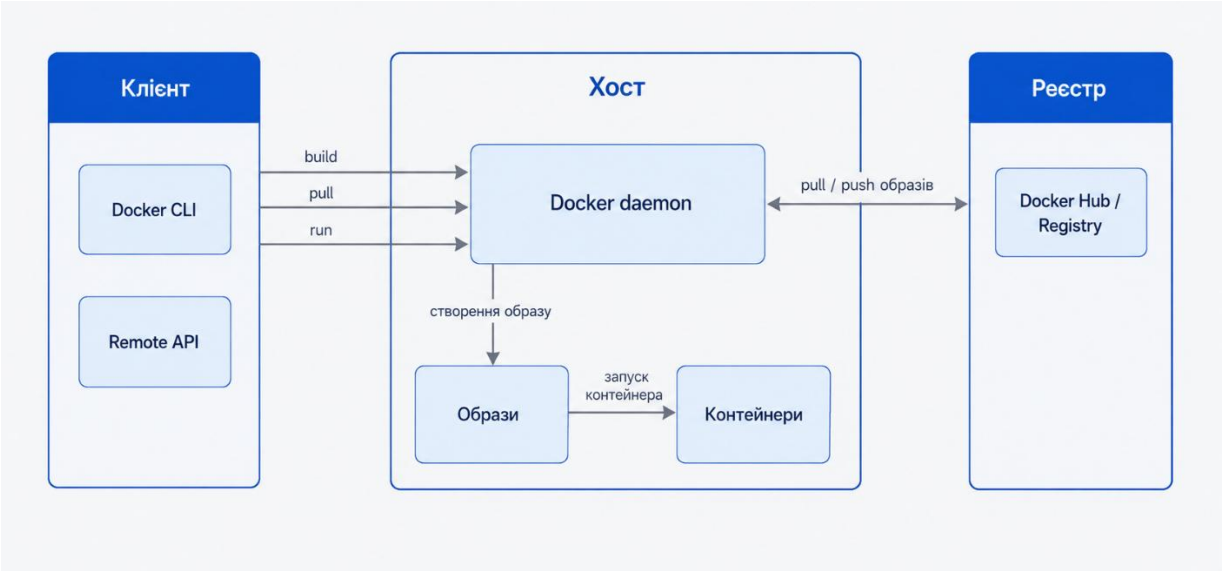


Рисунок 1.2 – Архітектура Docker

Альтернативою Docker є Podman — платформа, що розвивається в екосистемі Red Hat. Головна ідея Podman у тому, що він працює без постійного демона. Тобто немає центрального процесу, який постійно працює в системі, а контейнери запускаються як звичайні процеси в Linux.

Така модель спрощує керування контейнерами та краще підходить для сценаріїв без прав суперкористувача, тому Podman часто вважають безпечнішим за замовчуванням. Важливою ідеологічною відмінністю є мікросервісний підхід до самих інструментів керування, замість монолітного рішення Podman часто використовується у зв'язці з Buildah для збирання та Skopeo для публікації образів.

На практиці це дає більшу гнучкість, тому що набір інструментів можна підібрати під конкретний сценарій використання. Ще одна важлива особливість Podman це підтримка pods, тобто можливість групувати кілька контейнерів в одну логічну одиницю. Це зручно коли сервіс складається з кількох частин, які повинні працювати разом. Але під час переходу на Podman може виникати потреба в адаптації наявних сценаріїв автоматизації, так як більша частина готових прикладів, командних файлів і процесів розгортання орієнтована на Docker [6]. На рисунку 1.3 зображено архітектуру Podman.

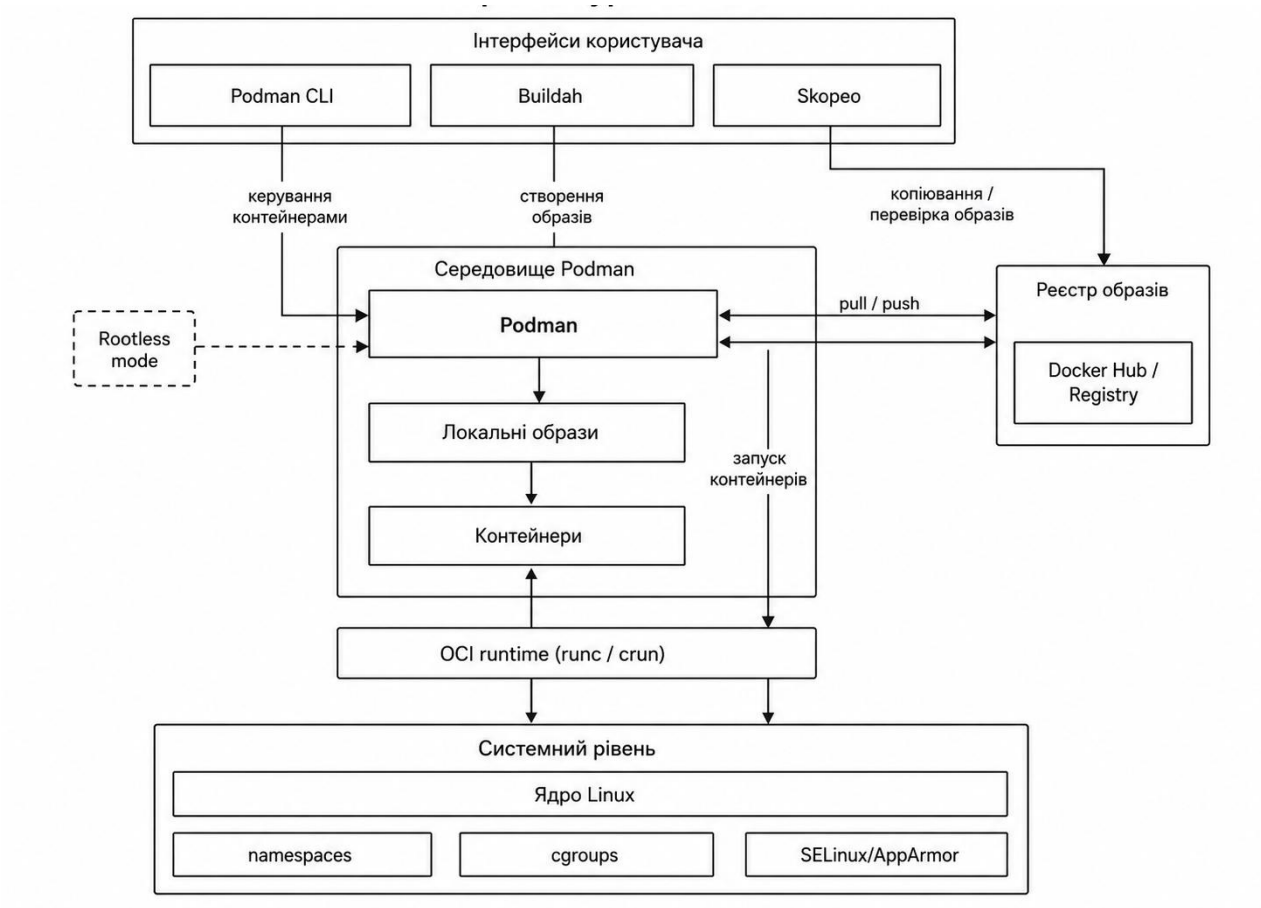


Рисунок 1.3 - Архітектура Podman

Коли на edge-вузлі з’являється кілька сервісів, ручне керування контейнерами стає незручним і ненадійним. Для цього потрібен механізм оркестрації, який підтримує потрібний стан системи, автоматично перезапускає сервіси та забезпечує кероване оновлення. Для середовищ де вже використовується Docker, логічним кроком часто стає Docker Swarm. Він надає базові можливості для декларативного опису сервісів, підтримки їхнього бажаного стану, балансування навантаження та поетапного оновлення без повної зупинки роботи системи. Це зручно, оскільки поріг входу невисокий, особливо якщо інфраструктура вже побудована на Docker. Але Swarm в більшості розглядають як простіший варіант, тому що у нього менше розширень і менше типових рішень для складних сценаріїв [7]. На рисунку 1.4 зображено архітектурна схема Swarm.

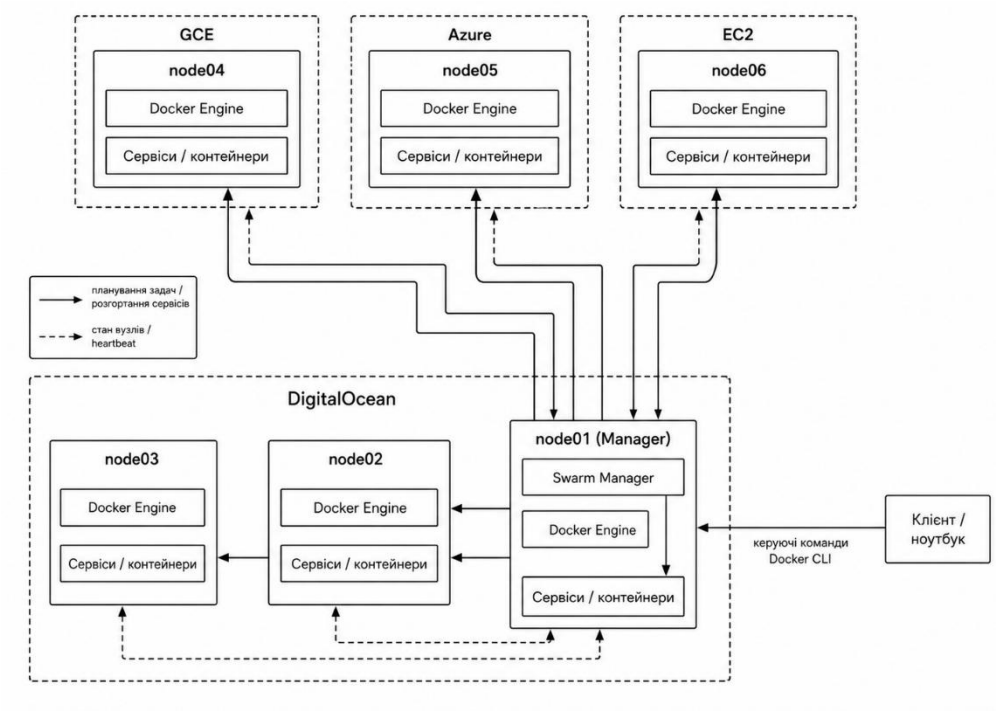


Рисунок 1.4 – Принцип роботи Swarm

Для складніших інфраструктур часто розглядають Kubernetes як промисловий стандарт оркестрації контейнерів. Проблема в тому, що класичний Kubernetes може бути важким для слабких edge-вузлів, тому для периферії існують легкі дистрибутиви. Одним із найпоширеніших легких дистрибутивів є k3s. Його перевага полягає у спрощеному встановленні та зменшенні системних витрат, водночас він зберігає основні об’єкти Kubernetes, зокрема групи контейнерів, розгортання та сервіси, керовані оновлення та можливість відкату. Для невеликих ARM-вузлів це часто виглядає як компроміс між функціональними можливостями та системним навантаженням. Водночас важливо розуміти, що навіть легкі дистрибутиви не роблять Kubernetes простим. Він все одно вимагає складних налаштувань, контролю ресурсів і тестування під конкретне навантаження бо результати сильно залежать від конфігурації і умов експлуатації [8]. На рисунку 1.5 зображено архітектурна схема дистрибутиву K3s.

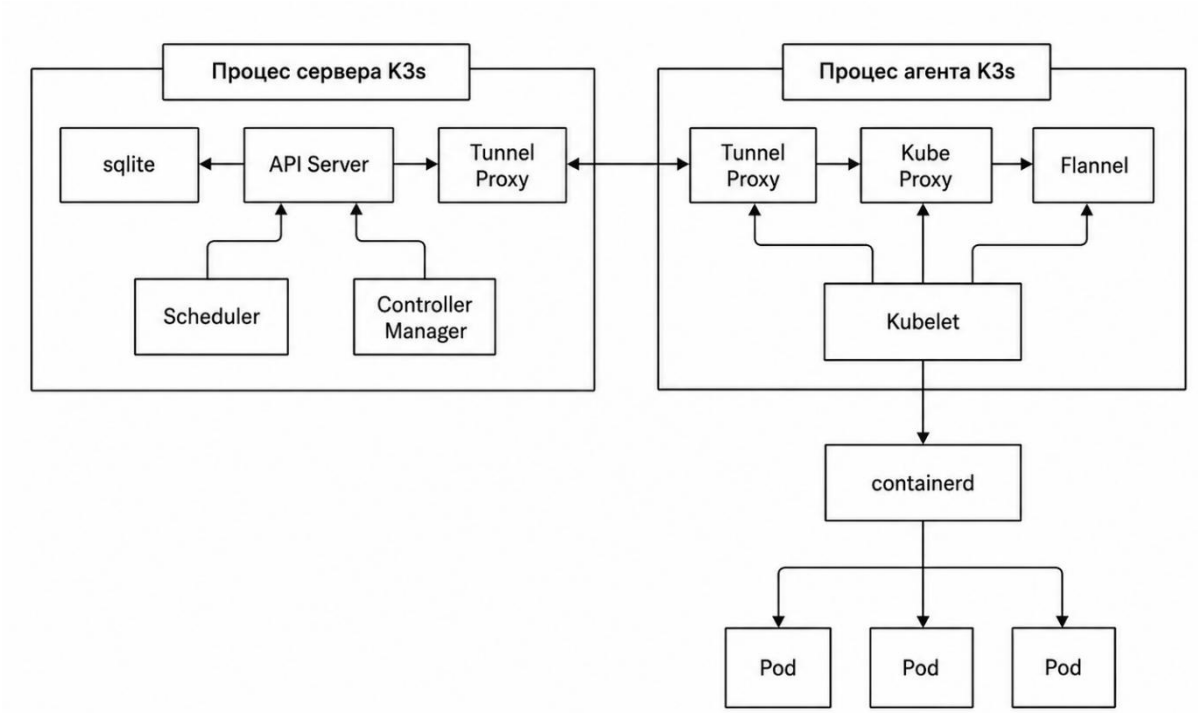


Рисунок 1.5 - Архітектурна схема дистрибутиву K3s

Ще одним підходом до оркестрації контейнерів є HashiCorp Nomad. Його часто описують як більш простий планувальник задач, який може бути зручним в edge-сценаріях через свою легкість і орієнтацію на роботу в умовах обмежених ресурсів та нестабільного зв’язку.

В багатьох випадках Nomad є простішим для впровадження, але для побудови повноцінної інфраструктури потрібні додаткові компоненти, зокрема засоби керування секретними даними та механізми автоматичного виявлення сервісів. Тому під час вибору Nomad потрібно наперед визначити спосіб запуску контейнерів, роботу з секретами, виявлення сервісів і загальну схему підтримки стеку [9]. На рисунку 1.6 зображено схему розгортання HashiCorp Nomad.

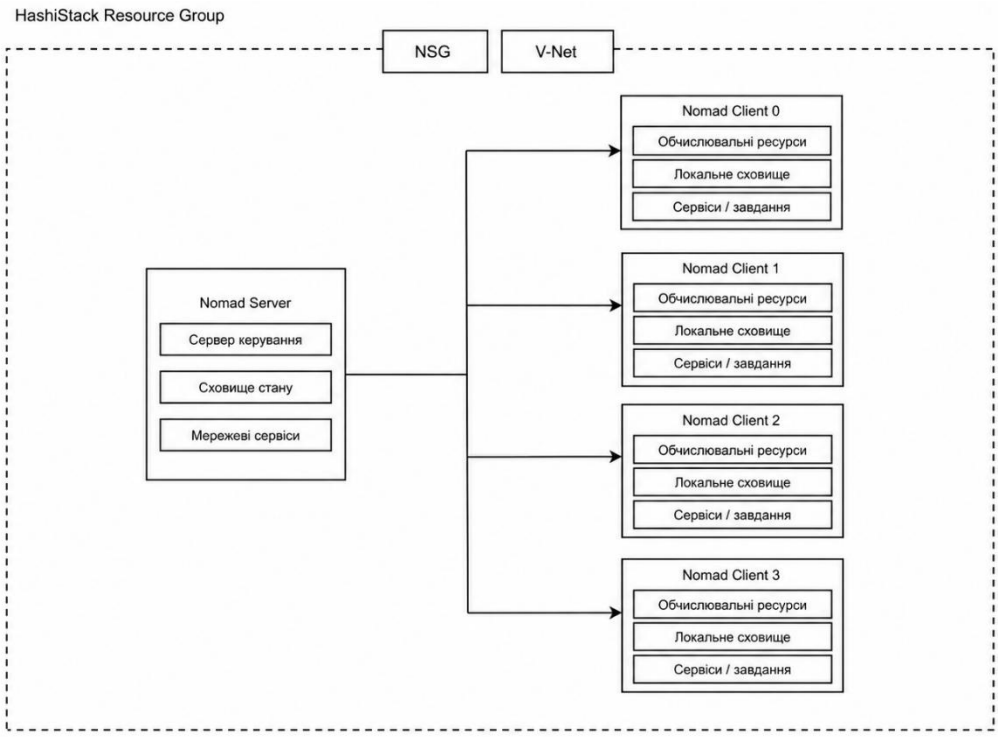


Рисунок 1.6 - Схема розгортання HashiCorp Nomad

Окрему групу становлять керовані IoT-платформи, які одразу надають засоби централізованого керування пристроями. Основна ідея таких платформ полягає у використанні готового середовища для розгортання, моніторингу та керування IoT-пристроями без необхідності самостійно будувати всі компоненти інфраструктури. Наприклад, у Azure IoT Edge є модель runtime-модулів і маніфестів, які описують, що саме має працювати на пристрої та як дані маршрутизуються між модулями і хмарою [10]. В AWS IoT Greengrass також використовується механізм віддаленого розгортання оновлень на окремі пристрої або групи пристроїв. Але під час його застосування потрібно контролювати формування конфігурації, тому що відсутній у новому описі компонент може бути видалений з пристрою, що впливає на коректність оновлення [11].. Це робить акцент на спрощенні підготовки пристроїв і зменшенні проблем через різне апаратне забезпечення, що корисно коли пристроїв багато. Такі платформи зручні, але вони сильніше прив’язують рішення до конкретної екосистеми та її інструментів [12].

Окремо ще треба розглядати не тільки OTA-оновлення контейнерів, а і самої операційної системи. Для цього часто застосовують підхід з двома системними розділами, за якого поточна версія працює з активного розділу, а нове оновлення

встановлюється на резервний розділ і активується лише після успішної перевірки. Якщо після перезавантаження все працює нормально, він стає активним, а якщо ні пристрій повертається до попереднього робочого стану. Це знижує ризик повної втрати працездатності після невдалого оновлення. Недолік такого підходу в тому що потрібен додатковий дисковий простір, а також складніша інтеграція з завантажувачем і перевіркою працездатності. На пристроях з обмеженим накопичувачем цей компроміс треба враховувати на початку проектування [13].

В загальному видно, що контейнерний стек для ARM-вузлів на edge будується навколо однієї ідеї - це зробити роботу сервісів однаковою на різних пристроях, забезпечити контрольований запуск і оновлення, а також не створювати зайвих ризиків безпеки. Контейнеризація дає переносимість і повторюваність, оркестрація додає керованість і автоматизацію, а OTA-підхід закриває питання стабільних оновлень в реальних умовах, коли доступ до пристрою може бути обмеженим або взагалі відсутнім.

1.3 Постановка задачі

В більшості сценаріїв коли edge-вузли використовуються як локальні міні-сервери застосовуються ARM-платформи. Проте разом із перевагами з’являються типові проблеми експлуатації: слабкий процесор, менше пам’яті, накопичувач. То ручні оновлення програм стають ризикованими.

Щоб зменшити ці ризики, доцільно використовувати контейнерний підхід. Контейнери дають однакове середовище запуску, зменшують залежність від налаштувань ОС і спрощують переносимість сервісу. Крім того потрібно мати механізм перевірки працездатності після оновлення і можливість автоматичного відкату, якщо нова версія не запускається або працює некоректно. Тому актуальною є задача побудови контейнерного стеку для ARM-вузла з безперервним розгортанням і OTA-оновленням, який працює стабільно навіть за обмежених ресурсів та не вимагає постійного ручного адміністрування.

Метою роботи є розробка та перевірка контейнерного стеку для ARM-вузла на прикладі Raspberry Pi. Стек має забезпечувати безперервне розгортання прикладного сервісу, віддалене оновлення контейнерів, перевірку працездатності та автоматичне повернення до попередньої стабільної версії в разі помилки.

Для досягнення поставленої мети необхідно виконати такі завдання:

1. провести аналіз предметної області edge-IoT на ARM та визначити вимоги до контейнерного розгортання в умовах обмежених ресурсів і нестабільного зв’язку;
2. провести огляд і аналіз існуючих рішень;
3. сформувати постановку задачі;
4. розробити алгоритмічне та інформаційне забезпечення контейнерного стеку для ARM-вузлів;
5. розробити архітектуру системи
6. Розробити алгоритм автоматизованого збирання, перевірки та публікації контейнерних образів у багатоплатформному форматі для подальшого запуску на ARM-вузлі;
7. розробити інформаційне забезпечення;
8. провести вибір та аналіз програмно-апаратних засобів для ARM-вузла;
9. реалізувати програмно-технологічне забезпечення ARM;
10. виконати тестування роботи стеку і модуля оновлення на типових сценаріях..

2 АЛГОРИТМІЧНЕ ТА ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ КОНТЕЙНЕРНОГО СТЕКУ ДЛЯ ARM-ВУЗЛІВ

2.1 Розробка архітектури системи

Для базової конфігурації ARM-вузла достатньо архітектури, розрахованої на один пристрій. Вони включають середовища зберігання коду, платформу автоматизованої збірки, централізоване сховище образів та крайовий (edge) вузол, відповідальний за виконання та оновлення сервісів. На рисунку 2.1 зображено загальну архітектуру системи.

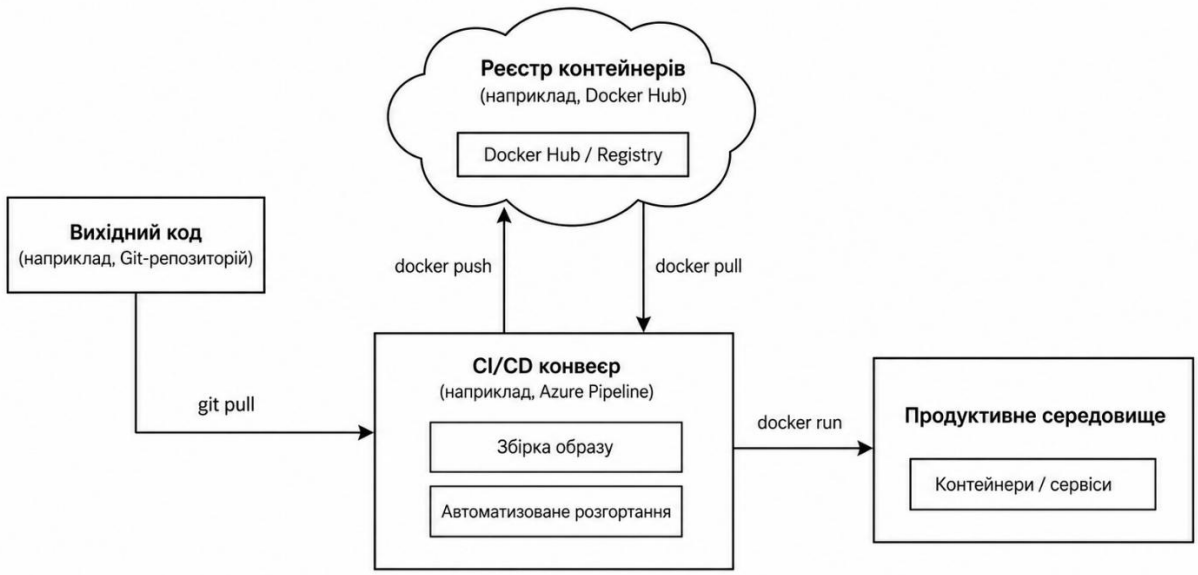


Рисунок 2.1 - Загальна архітектура компонентів системи

Життєвий цикл функціонування системи починається з Git-репозиторію. В репозиторії зберігаються вихідні тексти сервісів та файли, які описують контейнеризацію і запуск. Тут знаходяться Dockerfile, а також файл docker-compose.yml, де задається склад стека і параметри запуску. Репозиторій надає чітку прив’язку версії коду та версії розгортання. Коли розробник робить коміт або створює тег релізу з’являється формальний тригер для CI. Це означає, що кожне оновлення можна потім пояснити через конкретну версію. Після появи змін запускається CI-процес. Його завдання полягає в тому, щоб отримати код, зібрати контейнерний образ, перевірити коректність збірки та підготувати образ до

доставки на пристрій. Тому результат збірки повинен бути доступний вузлу у вигляді готових образів. Для цього використовується Container Registry, який являє собою опис складу контейнерних образів де зберігаються версії і звідки їх можна отримати по мережі.

У документації Docker реєстр визначається як централізоване місце для зберігання і поширення контейнерних образів також вказано, що Docker Hub є публічним реєстром і часто використовується як реєстр за замовчуванням. Для цієї реалізації не принципово, який саме реєстр використовується. Важливо, що реєстр зберігає історію версій, і дозволяє оновитися на новий тег та повернутися на попередній у разі проблем.

На Raspberry Pi управління цими образами здійснюється за допомогою Docker Compose який дозволяє описати кілька контейнерів в одному файлі і запускати їх разом. Завдяки цьому експлуатація зводиться до оновлення, запуску, зупинки та перезапуску всього стеку без ручного керування кожним контейнером окремо. В такому середовищі один сервіс може бути основним прикладним модулем, який імітує або приймає IoT-дані і віддає їх через API або простий веб-інтерфейс. Другий сервіс виступає як брокер повідомлень, щоб показати типовий IoT-підхід обміну даними. За потреби додається легке сховище або сервіс візуалізації.

Ключовим елементом забезпечення автономності системи є модуль розгортання та оновлення, реалізований у вигляді скрипта, що працює під керуванням системного менеджера systemd. Використання служби або таймера дозволяє автоматизувати регулярні перевірки оновлень і гарантує запуск агента після перезавантаження пристрою. На рисунку 2.2 зображено архітектурну схему модуля для оновлення.

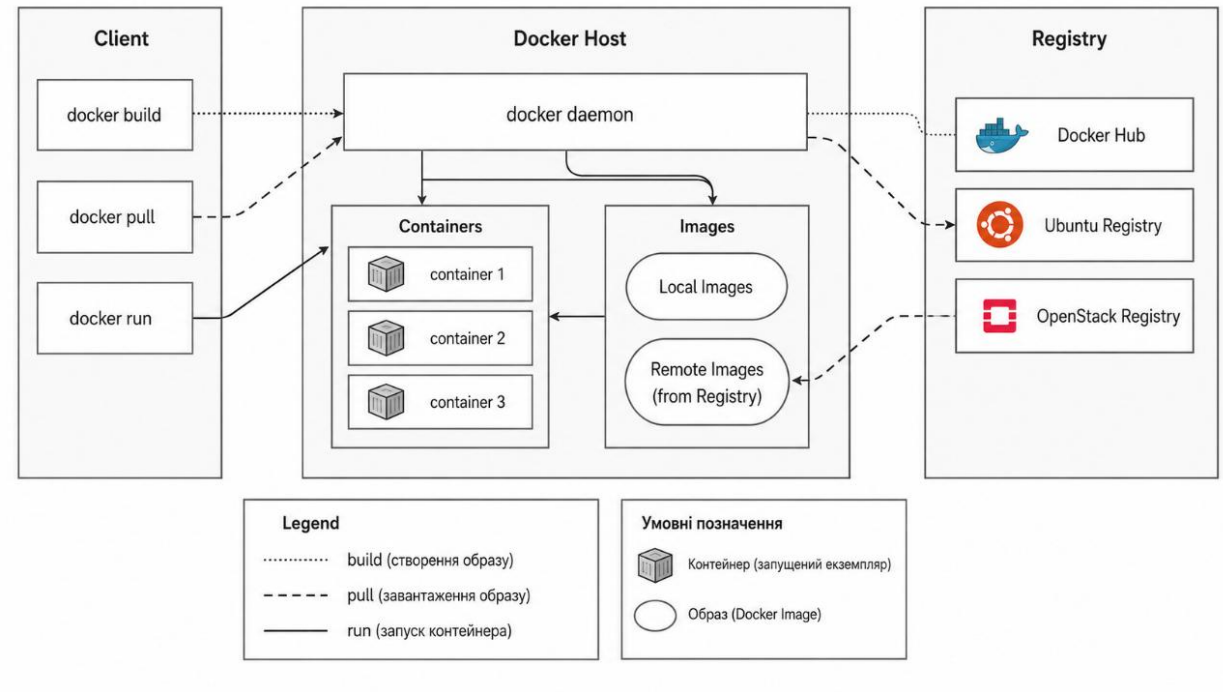


Рисунок 2.2 – Архітектура модуля оновлення

Модуль послідовно перевіряє мережеве з’єднання, доступність Container Registry і відповідність поточної версії стеку цільовій версії. В разі виявлення нових образів агент виконує керовану процедуру оновлення, яка включає завантаження артефактів через docker compose pull та їх розгортання за допомогою docker compose up -d. Крім того агент не обмежується перевіркою факту запуску контейнерів, адже активний процес ще не гарантує працездатність сервісу. Для цього в Docker Compose-стеку використовуються механізми перевірки працездатності сервісів (healthcheck) і контролю залежностей між компонентами (depends_on), що дає змогу забезпечити правильну послідовність запуску контейнерів та перевірку їх готовності до роботи. Після оновлення система перевіряє, чи перейшли сервіси у працездатний стан і чи готові вони виконувати свої функції. Якщо протягом встановленого часу працездатність сервісу не підтверджується, модуль оновлення автоматично повертає систему до попередньої стабільної версії, що дає змогу уникнути тривалого простою периферійного вузла.

Блок логування та моніторингу в межах даної архітектури є мінімально необхідним але достатнім інструментом для забезпечення прозорості роботи системи. Такий запис дозволяє оперативно діагностувати причини збоїв під час

старту або переходу сервісів у стан помилки. Додатково модуль оновлення зберігає результати кожної спроби розгортання у локальному сховищі даних. Такі записи фіксують час оновлення, номер активної версії, результат перевірки працездатності та причину помилки, якщо вона виникла. Функціональна структура системи автоматично перетворює зміни вихідного коду на готові контейнерні образи та зберігає їх у реєстрі для подальшого мережевого доступу. На самому периферійному вузлі Raspberry Pi сервіси запускаються як єдиний стек через Docker Compose, що виключає ризик виникнення помилок. Система автономно оновлює сервіси до нових версій, проводить верифікацію їхнього стану і, в разі невдалого оновлення, здійснює автоматичне повернення до попередньої стабільної конфігурації. Ведення журналу станів дає адміністратору змогу відстежити послідовність подій і швидко визначити подальші дії.

Ще однією важливою умовою роботи периферійного вузла є стабільне мережеве з’єднання, так як без доступу до інтернету Raspberry Pi не зможе завантажити оновлення. Хоча в реальних умовах експлуатації важливими є також параметри живлення та стабільність зв’язку, у межах структурної схеми їх доцільно відобразити, як зовнішні фактори, що безпосередньо впливають на загальну доступність системи. Також архітектура дозволяє сформувати базовий рівень безпеки без впровадження складних механізмів. Це забезпечує розмежування ручних та автоматизованих операцій коли розробник звільняється від необхідності щоразу заходити на Raspberry Pi для виконання команд в ручному режимі, тому що пристрій оновлюється за чітко визначеним сценарієм. Такий підхід мінімізує ризик виникнення випадкових помилок, зумовлених людським фактором, та робить процес розгортання ПЗ повністю повторюваним і прогнозованим.

2.2 Алгоритмічне забезпечення модуля

Модуль працює за циклічною схемою доставки програмного забезпечення на периферійний вузол. Процес починається з фіксації змін у Git-репозиторії, де

розробник оновлює код сервісів або конфігурацію стека. Ця подія запускає середовище автоматизованого збирання, яке формує контейнерні образи, адаптовані для архітектури ARM, та виконує їх базову перевірку перед публікацією (рисунок 2.3).

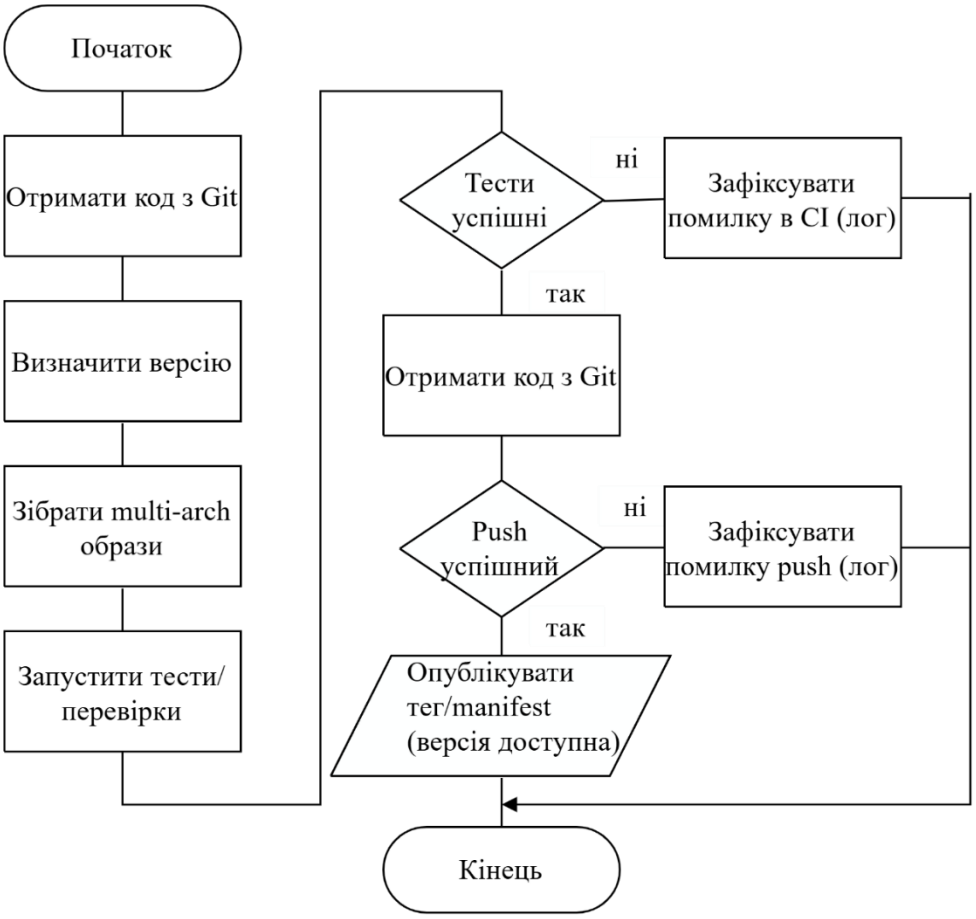


Рисунок 2.3 - Схема алгоритму CI-частини

Зібрані артефакти передаються до централізованого реєстру (Container Registry), який виступає проміжним сховищем та гарантує доступність потрібної версії для кінцевого пристрою.

На самому Raspberry Pi розгорнуто агент оновлення який постійно моніторить реєстр на предмет появи нових версій. Коли оновлення виявлено агент ініціює процедуру завантаження нових образів та переконфігурацію Docker Compose стека. Основне завдання цього етапу — замінити поточні сервіси новими версіями з мінімальним простоєм. Після перезапуску система проводить

автоматичну перевірку працездатності, і якщо сервіси функціонують коректно, цикл оновлення вважається завершеним.

Після завершення процесу підготовки образів система переходить до фази безпосередньої експлуатації на пристрої (рисунок 2.4).

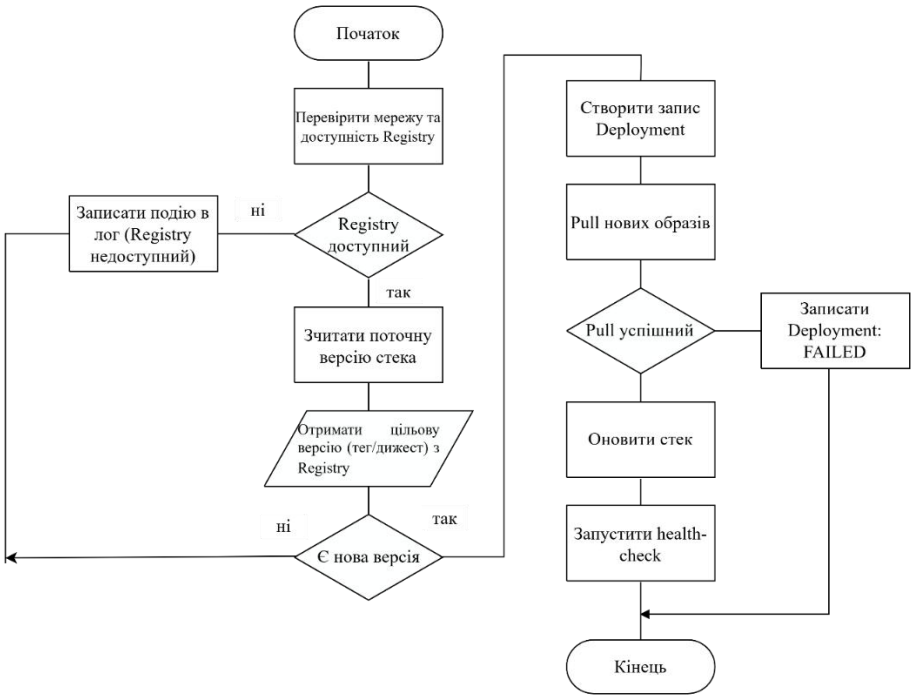


Рисунок 2.4 - Схема алгоритму OTA-оновлення на Raspberry Pi

Ключовим елементом забезпечення автономності та надійності є алгоритм роботи модуля оновлення. Робота модуля починається за сигналом системного таймера systemd, що забезпечує циклічність перевірок. На першому етапі система виконує діагностику мережевого середовища та доступності Container Registry. Якщо зв’язок відсутній, агент фіксує подію в журналі та завершує цикл, не змінюючи поточну конфігурацію. Це захищає вузол від помилкових оновлень за нестабільного з’єднання. В разі успішного з'єднання система зчитує ідентифікатор локальної версії та порівнює його з цільовим тегом опублікованим у реєстрі. Якщо виявлено нову версію, створюється запис в базі даних після чого ініціює завантаження артефактів командою docker compose pull. У випадку коли процес завантаження переривається помилкою система фіксує це і зупиняє розгортання не порушуючи роботу чинного стека сервісів.

Щоб зберегти стабільність системи навіть у разі невдалого оновлення, процес розгортання не завершується одразу після запуску контейнерів. На цьому етапі перевіряється факт запуску процесів та реальна готовність сервісів до роботи. В додатку Б зображено діаграму розподілу повного циклу CI/CD та OTA-оновлення контейнерного сервісу.

Після оновлення стека система витримує технологічну паузу, необхідну для ініціалізації всіх внутрішніх компонентів та встановлення мережевих з'єднань між контейнерами. Перевіряється статус контейнерів, щоб переконатися, що вони перейшли в стан «running». Далі алгоритм ініціює процедуру повернення до попередньої версії.

Ключовою особливістю цього модуля є перевірка працездатності сервісу на рівні прикладного інтерфейсу /health. Так як IoT-сервіси можуть запускатися але не виконувати свої функції через внутрішні помилки коду, система використовує циклічний метод опитування. Алгоритм передбачає кілька повторних спроб перевірки з визначеними інтервалами очікування, що дає змогу уникнути помилкового спрацювання під час повільного запуску окремих компонентів сервісу. Лише після вичерпання ліміту спроб без позитивного результату система офіційно визнає оновлення невдалим і запускає механізм автоматичного відкату.

Процедура повернення до попередньої версії також є контрольованою. Після відновлення попереднього стабільного стану система повторно перевіряє працездатність сервісу. Це дозволяє чітко розмежувати ситуації, коли проблема була в оновленні, від ситуацій критичного збою апаратної частини або інфраструктури, фіксуючи відповідний статус у базі даних моніторингу для подальшого аналізу адміністратором.

2.3 Інформаційне забезпечення

Для стабільної роботи контейнерного стеку безперервного розгортання визначено інформаційні потоки, з якими працює система. Взаємодія між віддаленим реєстром контейнерних образів, локальним модулем розгортання та

середовищем виконання Docker базується на обміні службовими даними. Такий обмін даними дає змогу автоматизувати оновлення та контролювати поточний стан IoT-пристрою.

Вхідна інформація охоплює зовнішні сигнали, параметри реєстру контейнерних образів і показники середовища виконання, на основі яких модуль визначає потребу в оновленні. До цих даних входить інформація про нові релізи контейнерних образів у Container Registry, представлені у вигляді тегів версій або унікальних цифрових відбитків (digest). До вхідних даних належать результати виконання системних команд безпосередньо на Raspberry Pi, поточні статуси контейнерів після маніпуляцій з Docker Compose.

Важливою частиною вхідного потоку є показники працездатності сервісів, які надходять від прикладного інтерфейсу перевірки стану /health або від внутрішніх перевірок Docker. Ці дані підтверджують, що програма запущена та готова виконувати свої функції.

Вихідна інформація потрібна для прозорого контролю процесів і подальшого аналізу адміністратором. Основним елементом цього є журнал розгортання, який фіксує кожну спробу оновлення з вказанням часових міток, номеру версії, фінального результату та причин можливих збоїв.

Крім того система генерує дані про поточний операційний стан, що відображають список активних сервісів та їхній стан працездатності в режимі реального часу.

Окрема частина вихідної інформації складає телеметрія самого пристрою, яка включає мережеву адресу, архітектуру процесора, тип операційної системи та час останньої активності вузла в мережі. Ці дані представлено в таблиці 2.1, де подано характеристики вхідної та вихідної інформації системи.

Таблиця 2.1 - Формат вхідної та вихідної інформації

№	Назва поля	Тип даних	Призначення
1	device_id	текст	ідентифікатор вузла
2	arch	текст	архітектура пристрою
3	os_name	текст	ОС на вузлі
4	ip_addr	текст	IP-адреса вузла

5	last_seen	дата/час	час останнього “heartbeat”
6	service_name	текс	назва сервісу в Compose
7	current_version	текст	активний тег сервісу
8	health_status	текст	стан сервісу
9	deployment_id	ціле	спроби оновлення
10	target_version	текст	версія, на яку оновлювались
11	prev_version	текст	версія до оновлення
12	start/ end _time	дата/час	початок і кінець оновлення
13	result	текст	підсумок
14	reason	текст	коротка причина помилки
15	check_status	текст	результат health-check
16	check_details	текст	деталі перевірки

Також для системи необхідно забезпечити логічну структуру зберігання даних. База даних побудована за реляційним принципом і складається з чотирьох основних сутностей: Device, Service, Deployment та HealthCheck. Схема зв’язків між сутностями представлена на рисунку 2.5.

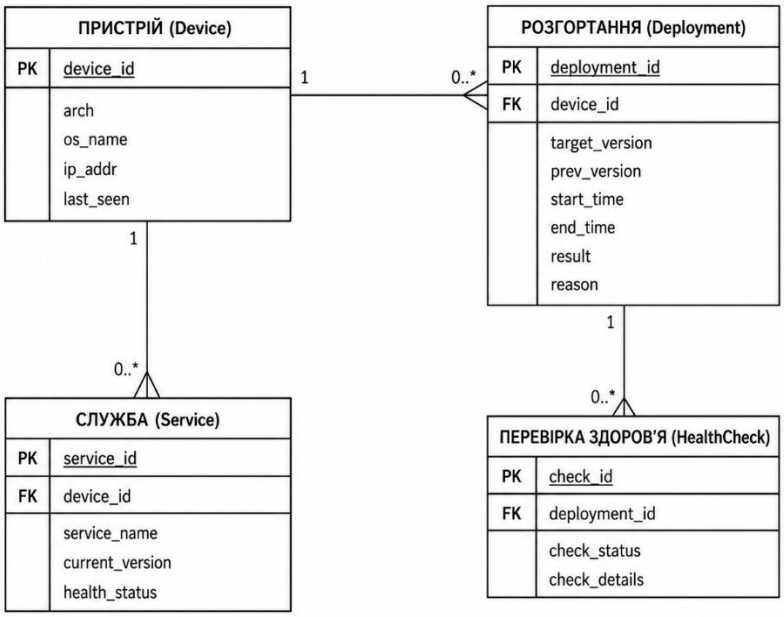


Рисунок 2.5 - ER-діаграма логічної структури бази даних

Основною є таблиця Deployment, у якій зберігається інформація про кожну спробу оновлення сервісів. Для зберігання ж стану використовується таблиця Service яка містить назву сервісу, поточну версію та статус працездатності. Результати перевірок після оновлення фіксуються в таблиці HealthCheck,

пов’язаній з Deployment через deployment_id. Така структура дозволяє вести історію оновлень і контролювати стан сервісів у системі.

3. ПРОГРАМНО-ТЕХНОЛОГІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Вибір та аналіз програмно-апаратних засобів для ARM-вузла

Для побудови контейнерного стеку на edge-рівні підібрано апаратні та програмні засоби, достатні для стабільного запуску й оновлення сервісів. В якості апаратної основи ARM-вузла було обрано мікрокомп’ютер Raspberry Pi (рисунок 3.1).

Рисунок 3.1 - Модуль розробки Raspberry Pi

Вибір Raspberry Pi є доцільним завдяки підтримці Linux і наявності мережевих інтерфейсів та достатній продуктивності для роботи базових компонентів контейнерного стеку. Крім того модуль має низьке енергоспоживання, що дозволяє використовувати вузол як постійно увімкнений edge-шлюз. Для зберігання системи та контейнерних даних використовується microSD-носій. Тому в реалізації враховуються обмеження щодо обсягу логів і записів на диск.

У програмній частині використано Docker Engine як контейнерний рушій, а Docker Compose — для опису та запуску набору сервісів. Це спрощує розгортання тому що, вся конфігурація стеку задається у файлах compose і може зберігатися у репозиторії (рисунок 3.2).

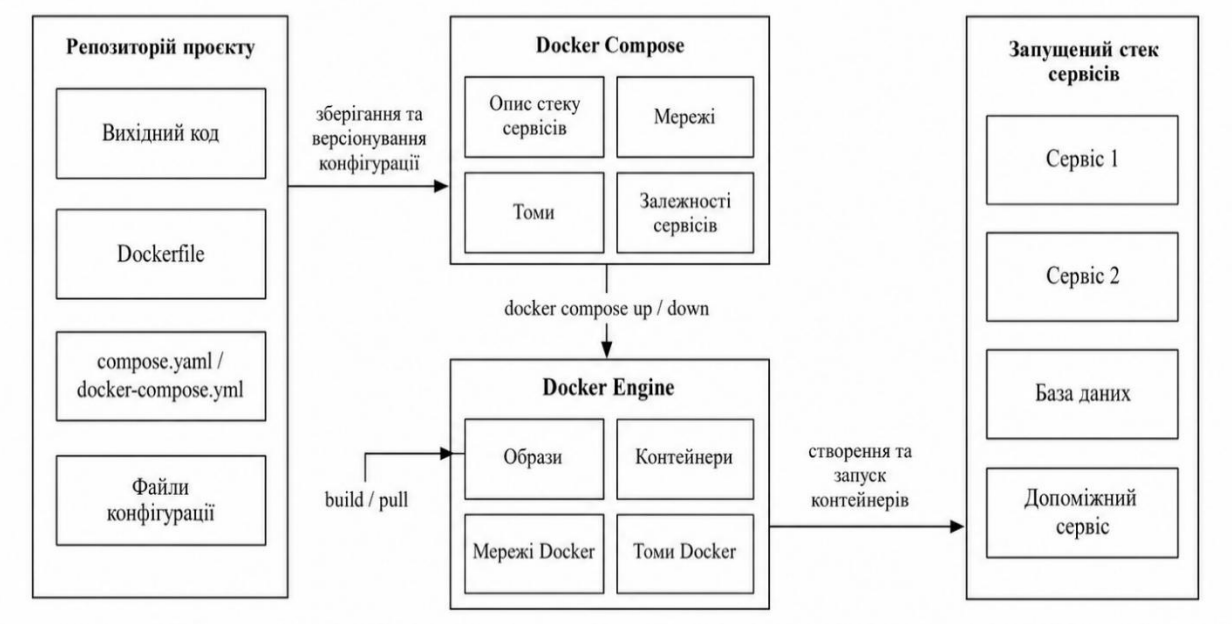


Рисунок 3.2 - Роботи контейнерного стеку

У реалізації використовується репозиторій з програмним кодом і конфігураційними файлами. Після внесення змін контейнерні образи формуються автоматизовано та публікуються у віддаленому реєстрі контейнерних образів, звідки їх може завантажити ARM-вузол. Завдяки цьому ARM-вузол отримує готові образи через стандартний механізм pull, без ручної збірки на Raspberry Pi.

Після визначення апаратної та програмної складових системи було виконано перехід до етапу тестування. Як тестове середовище обрано платформу Oracle VM VirtualBox із встановленим дистрибутивом Debian, що дає змогу перевірити працездатність основних компонентів програмного стеку в ізольованому середовищі (рисунок 3.3).

Використання віртуальної машини дозволяє забезпечити швидке розгортання системи, повторюваність експериментів та можливість безпечної перевірки сценаріїв оновлення контейнерів без впливу на основну операційну систему.

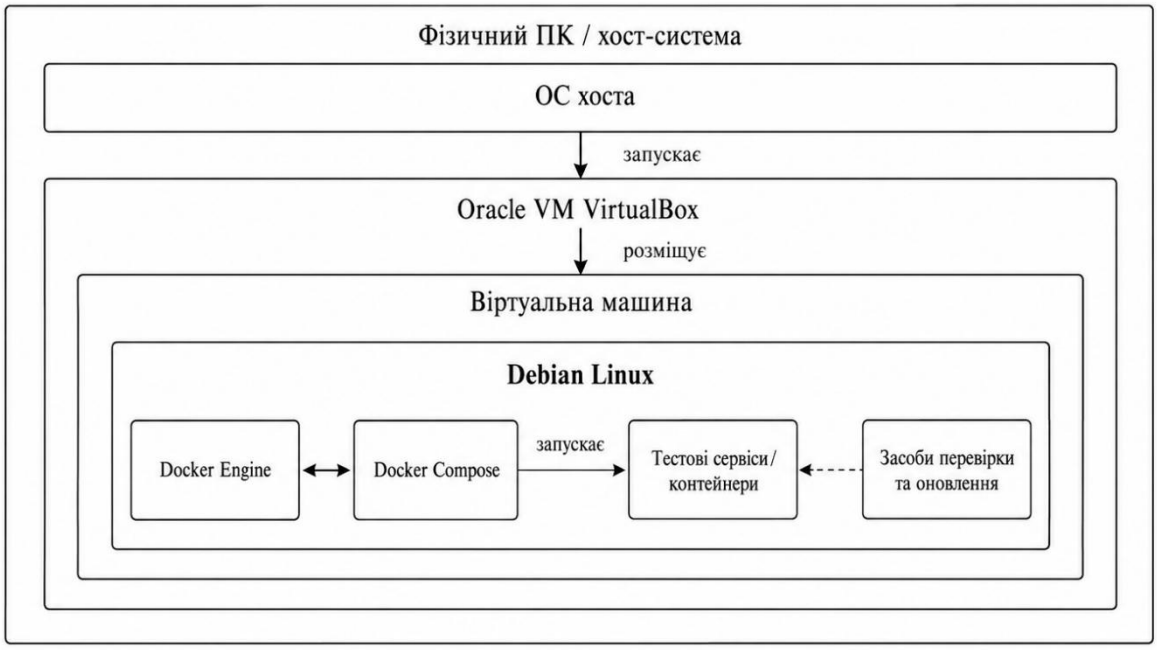


Рисунок 3.3 – Середовище Oracle VM VirtualBox з Debian

3.2 Програмно-технологічне забезпечення ARM

Програмна реалізація побудована за модульним принципом. Структура охоплює всі основні етапи автоматизації: збірку контейнерних образів, збереження їх в реєстрі, розгортання на периферійному пристрої та подальший контроль стану сервісів. Це дозволяє ізольовано тестувати окремі модулі, адаптувати стек під різні апаратні конфігурації IoT-вузлів та вдосконалювати механізми доставки контенту без необхідності повної перебудови коду системи.

На початковому етапі програмної реалізації виконується розгортання та базова конфігурація середовища Docker Engine і Docker Compose. Для цього у середовищі Debian використовується команда встановлення пакетів docker.io та docker-compose, що забезпечує підготовку операційної системи до запуску контейнеризованих сервісів (рисунок 3.4).

В процесі встановлення система автоматично додає основні пакети Docker, та необхідні залежності, встановлюються компоненти containerd та runc, які відповідають за керування життєвим циклом контейнерів і їх запуск відповідно до OCI-стандарту. Також встановлюються допоміжні пакети для мережевої взаємодії,

такі як iptables, що використовується для налаштування правил маршрутизації та ізоляції контейнерних мереж.

```
~$ sudo apt -y install docker.io docker-compose
Installing:
  docker-compose  docker.io

Installing dependencies:
  containerd      iptables          libip6tc2                 python3-protobuf
  criu            libcompel1        libmodule-find-perl       python3-pycriu
  docker-buildx   liberror-perl     libproc-processtable-perl runc
  docker-cli      libintl-perl      libsort-naturally-perl    tini
  git             libintl-xs-perl   libterm-readkey-perl
  git-man         libip4tc2         needrestart

Suggested packages:
  containernetworking-plugins  xfsprogs          git-cvs
  docker-doc                   zfs-fuse          git-mediawiki
```

Рисунок 3.4 – Встановлення Docker Engine та Docker Compose у середовищі Debian

Після встановлення Docker Engine і Docker Compose налаштовується віддалений реєстр контейнерних образів. Для цього використовується сервіс Docker Hub, який дає змогу зберігати створені Docker-образи, керувати репозиторіями та надалі завантажувати їх на цільовий edge-вузол під час розгортання або оновлення системи.

На рисунку 3.5 показано інтерфейс Docker Hub із створеним репозиторієм для збереження образу програмного компонента.

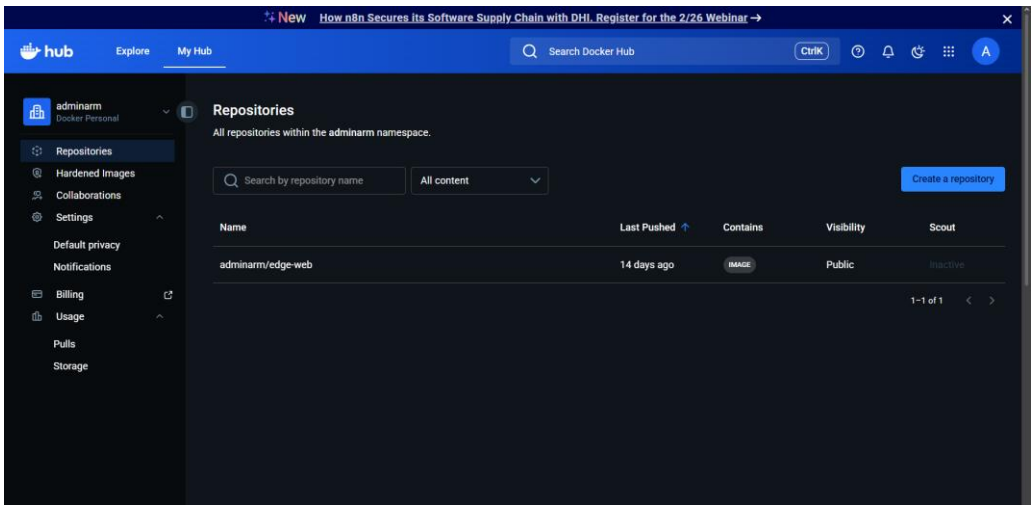


Рисунок 3.5 - Платформа для автоматизації розгортання

Такий репозиторій потрібен для централізованого зберігання образів, контролю версій і подальшого оновлення контейнеризованих сервісів. Після формування локального Docker-образу він може бути завантажений до Docker Hub за допомогою команди `docker push`, а на стороні вузла — отриманий командою `docker pull`.

Це спрощує поширення оновлень, повторне розгортання сервісів і створює основу для контрольованого OTA-оновлення програмних компонентів.

На рисунку 3.6 показано приклад створення файлу docker-compose.yml для першого тестового сервісу на основі образу nginx:alpine. В конфігурації вказано прокидання порту 8080:80, що дає змогу отримати доступ до вебсервісу з хост-системи через порт 8080. Також у файлі задано параметр healthcheck, який періодично перевіряє доступність сервісу за допомогою HTTP-запиту до локального вебсервера.

Рисунок 3.6 – Створення docker-compose.yml з тестовим сервісом і healthcheck

Після створення файлу docker-compose.yml виконується запуск контейнерного сервісу та перевірка правильності його конфігурації. Для цього використовується команда docker-compose up -d, яка запускає стек та автоматично завантажує необхідний Docker-образ, якщо він відсутній у локальному сховищі.

На рисунку 3.7 показано процес завантаження образу nginx:alpine, створення Docker-мережі stack_default та запуск контейнера stack-web-1.

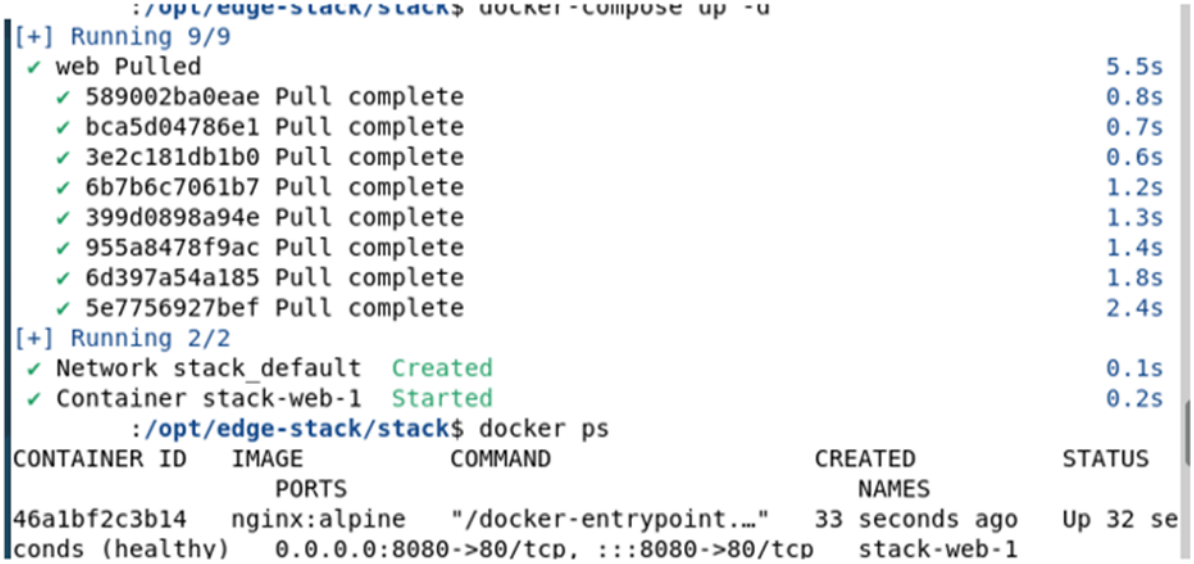


Рисунок 3.7 – Запуск контейнерного сервісу та перевірка його стану

Після завершення запуску додатково використовується команда docker ps, яка перевіряє стан активних контейнерів. Статус healthy підтверджує, що механізм healthcheck, заданий у конфігурації, працює коректно.

Завдяки healthcheck, Docker може визначати стан сервісу після запуску або оновлення та фіксувати, чи працює контейнер коректно. Якщо виникає помилка, її можна використати як підставу для повторного запуску сервісу або повернення до попередньої стабільної версії.

Наступним етапом є налаштування автоматичного запуску контейнерного стеку за допомогою systemd. Для цього створюється окремий сервісний файл edge-stack.service, в якому визначаються параметри запуску та зупинки стеку на основі docker-compose (рисунок 3.8).

В конфігурації сервісу задається робочий каталог, в якому розміщено файл docker-compose.yml, а також команди ExecStart і ExecStop, що відповідають за запуск і зупинку контейнерів. Крім того, вказуються залежності від мережевих служб і docker.service.

```
:/opt/edge-stack/stack$ sudo tee /etc/systemd/system/edge-stack.service
> /dev/null <<'UNIT'
[Unit]
Description=Edge stack (docker-compose)
After=network-online.target docker.service
Wants=network-online.target
Requires=docker.service

[Service]
Type=oneshot
RemainAfterExit=yes
WorkingDirectory=/opt/edge-stack/stack
ExecStart=/usr/bin/docker-compose up -d
ExecStop=/usr/bin/docker-compose down
TimeoutStartSec=0

[Install]
WantedBy=multi-user.target
UNIT
```

Рисунок 3.8 – Налаштування автозапуску контейнерного стеку через systemd

Після створення сервісу systemd виконується перевірка його працездатності та коректність запуску контейнерного стеку. Для цього застосовуються команди systemctl daemon-reload, systemctl enable --now edge-stack.service та systemctl status edge-stack.service, які дають змогу оновити конфігурацію служб, увімкнути автозапуск і переглянути поточний стан створеного сервісу (рисунок 3.9).

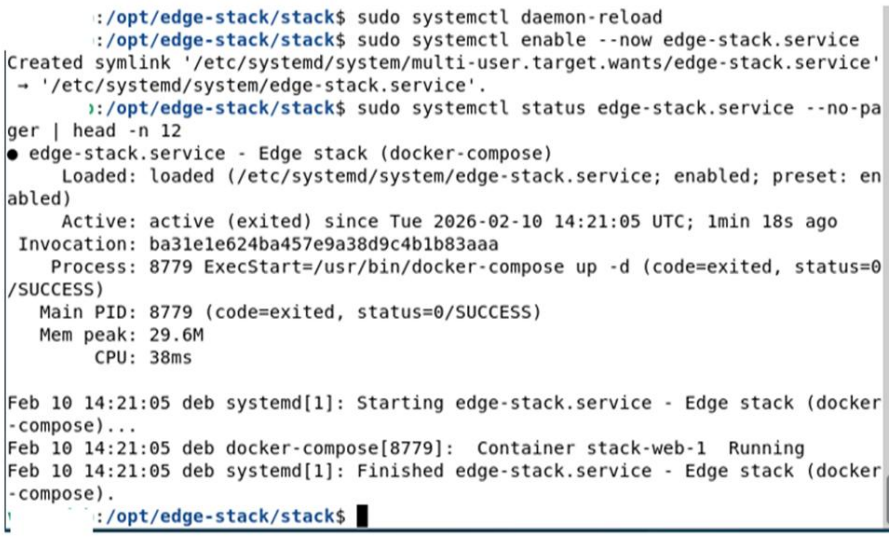


Рисунок 3.9 – Перевірка роботи сервісу автозапуску контейнерного стеку

На рисунку 3.9 показано, що системна служба edge-stack.service успішно завантажена та має стан active (exited). Для служби типу oneshot такий стан означає, що команда запуску була виконана коректно. У журналі виконання також

підтверджується успішний запуск контейнера stack-web-1 в межах описаного Docker Compose-стеку.

Другим етапом програмної реалізації є створення локальної бази даних SQLite, призначеної для збереження службової інформації про процес оновлення контейнеризованих сервісів. Такий підхід дає змогу вести журнал оновлень, фіксувати їх результати та зберігати службову інформацію про стан системи без окремого сервера баз даних (рисунок 3.10).

На рисунку 3.10 показано процес створення структури бази даних deployments.db. В ній формуються таблиці Device, Service, Deployment та HealthCheck. Таблиця Device містить відомості про цільовий пристрій, його архітектуру, операційну систему, IP-адресу та час останньої активності.

```
:/opt/edge-stack/data$ sqlite3 /opt/edge-stack/data/deployments.db <<'SQL'
PRAGMA foreign_keys = ON;

CREATE TABLE IF NOT EXISTS Device (
  device_id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  arch TEXT,
  os_name TEXT,
  ip_addr TEXT,
  last_seen TEXT
);

CREATE TABLE IF NOT EXISTS Service (
  service_id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL UNIQUE,
  current_version TEXT,
  health_status TEXT,
  updated_at TEXT
);

CREATE TABLE IF NOT EXISTS Deployment (
  deployment_id INTEGER PRIMARY KEY AUTOINCREMENT,
  service_name TEXT NOT NULL,
  prev_version TEXT,
  target_version TEXT NOT NULL,
  time_start TEXT NOT NULL,
  time_end TEXT,
  result TEXT NOT NULL,
  reason TEXT
SQLECT name FROM sqlite_master WHERE type='table' ORDER BY name;d) ON DELETE CASCADE
Deployment
Device
HealthCheck
Service
sqlite_sequence
```

Рисунок 3.10 – Створення SQLite бази даних для журналу оновлень

Таблиця Service використовується для збереження інформації про сервіси, їх поточні версії та стан працездатності. Основною для журналу оновлень є таблиця Deployment, у якій фіксуються попередня і цільова версії сервісу, час початку і завершення оновлення, результат виконання та причина можливої помилки.

Таблиця HealthCheck - результати перевірки працездатності сервісу після виконання оновлення.

Далі середовище підготовлено до роботи з мультиархітектурними контейнерними образами. Це необхідно для забезпечення сумісності програмних компонентів із різними типами вузлів, зокрема з платформами arm64 та amd64, які можуть використовуватися в межах системи edge-обчислень (рисунок 3.11).

На рисунку 3.11 показано процес налаштування підтримки багатоплатформного збирання та використання образів за допомогою допоміжного контейнера tonistiigi/binfmt.

```
:~/edge-web$ docker run --privileged --rm tonistiigi/binfmt --install all
Unable to find image 'tonistiigi/binfmt:latest' locally
latest: Pulling from tonistiigi/binfmt
704356aed36b: Pull complete
979978315595: Pull complete
Digest: sha256:8f58e6214f4cc9dc83ce8f5acad1ece508eb6b20e696a8c1e9f274481982c541
Status: Downloaded newer image for tonistiigi/binfmt:latest
installing: arm OK
installing: s390x OK
installing: riscv64 OK
installing: mips64le OK
installing: arm64 OK
installing: ppc64le OK
installing: mips64 OK
installing: loong64 OK
{
  "supported": [
    "linux/amd64",
    "linux/amd64/v2",
    "linux/arm64",
    "linux/riscv64",
    "linux/ppc64le",
    "linux/s390x",
    "linux/386",
    "linux/mips64le",
    "linux/mips64",
    "linux/loong64",
    "linux/arm/v7",
    "linux/arm/v6"
```

Рисунок 3.11 – Налаштування підтримки multi-arch образів для платформ arm64 та amd64

Після виконання команди в системі реєструються механізми емуляції для різних апаратних архітектур, таких як linux/amd64, linux/arm64, linux/arm/v7 та інші. Це дозволяє створювати, тестувати та використовувати Docker-образи, орієнтовані на різні типи процесорних платформ.

Підтримка multi-arch образів дає змогу формувати один набір образів для різних edge-вузлів без окремих конфігурацій розгортання для кожної архітектури.

На рисунку 3.12 показано фрагмент програмної реалізації update-скрипту, в якому задаються основні параметри середовища: шлях до каталогу стеку, розташування бази даних deployments.db, файл журналу оновлень, назва сервісу та цільовий Docker-образ.

```
#!/usr/bin/env bash
set -euo pipefail

STACK_DIR="/opt/edge-stack/stack"
DB="/opt/edge-stack/data/deployments.db"
LOG="/opt/edge-stack/logs/update.log"
SERVICE_NAME="web"
DEFAULT_TARGET="nginx:alpine"

ts() { date -Iseconds; }

mkdir -p "$(dirname "$LOG")"
cd "$STACK_DIR"

HTTP_CODE="$(curl -sS -o /dev/null -w "%{http_code}" -I https://registry-1.docker.io/v2/ || echo 0
if [[ "$HTTP_CODE" != "200" && "$HTTP_CODE" != "401" ]]; then
  echo "[$(ts)] registry unreachable (http=$HTTP_CODE)" | tee -a "$LOG"
  exit 1
fi
```

Рисунок 3.12 – Фрагмент update-скрипту з фіксацією результатів у SQLite

Окремо реалізовано перевірку доступності Docker Registry за допомогою HTTP-запиту. Якщо реєстр недоступний, скрипт фіксує відповідний запис в журналі та завершує роботу, що дає змогу уникнути некоректного виконання оновлення. Ще однією важливою функцією скрипту є взаємодія з базою SQLite, в якій зберігаються дані про хід і результат оновлення.

В цьому ж update-скрипті реалізовано автоматичний rollback — повернення до попередньої стабільної версії контейнера в разі помилки оновлення або невдалої перевірки працездатності.

На рисунку 3.13 показано фрагмент скрипту, в якому після виявлення помилки виконується повернення до попереднього образу контейнера. Після цього повторно запускається сервіс, а його стан перевіряється за допомогою HTTP-запиту. Залежно від отриманого результату в базі даних SQLite фіксується один із підсумкових статусів: ROLLBACK_OK в разі успішного відновлення або CRITICAL, якщо повернення до попередньої версії також не забезпечило працездатність сервісу.

```
echo "[$(ts)] health failed -> rollback to $PREV_RESOLVED" | tee -a "$LOG"
echo "WEB_IMAGE=$PREV_RESOLVED" > .env
docker-compose up -d >>"$LOG" 2>&1 || true
sleep 2

if curl -fsS http://localhost:8080/ >/dev/null; then
  sqlite3 "$DB" "UPDATE Deployment SET time_end='$(ts)', result='ROLLBACK_OK', reason='health_fail
  sqlite3 "$DB" "INSERT INTO Service(name,current_version,health_status,updated_at)
    VALUES('$SERVICE_NAME','$PREV_RESOLVED','healthy','$(ts)')
    ON CONFLICT(name) DO UPDATE SET
      current_version=excluded.current_version,
      health_status=excluded.health_status,
      updated_at=excluded.updated_at;"
  echo "[$(ts)] rollback OK" | tee -a "$LOG"
  exit 1
else
  sqlite3 "$DB" "UPDATE Deployment SET time_end='$(ts)', result='CRITICAL', reason='health_failed_
  sqlite3 "$DB" "INSERT INTO Service(name,current_version,health_status,updated_at)
    VALUES('$SERVICE_NAME','$PREV_RESOLVED','unhealthy','$(ts)')
    ON CONFLICT(name) DO UPDATE SET
      current_version=excluded.current_version,
      health_status=excluded.health_status,
      updated_at=excluded.updated_at;"
  echo "[$(ts)] rollback FAIL" | tee -a "$LOG"
  exit 1
fi
```

Рисунок 3.13 – Реалізація функції автоматичного rollback в update-скрипті

Наступним етапом є налаштування автоматичного періодичного запуску процедури оновлення за допомогою механізму systemd timer. Для цього створюється таймер edge-update.timer, який забезпечує регулярний виклик відповідного сервісу оновлення без участі користувача (рисунок 3.14).

```
        :/opt/edge-stack/stack$ sudo tee /etc/systemd/system/edge-update.timer > /dev/null <<'UNIT'

[Unit]
Description=Run edge update periodically

[Timer]
OnBootSec=2min
OnUnitActiveSec=10min
Persistent=true

[Install]
WantedBy=timers.target
UNIT
        :/opt/edge-stack/stack$ sudo systemctl daemon-reload
        :/opt/edge-stack/stack$ sudo systemctl enable --now edge-update.timer
Created symlink '/etc/systemd/system/timers.target.wants/edge-update.timer' → '/etc/systemd/system/
edge-update.timer'.
        :/opt/edge-stack/stack$ systemctl list-timers --all | grep edge-update
Tue 2026-02-10 15:45:41 UTC     9min Tue 2026-02-10 15:35:41 UTC      42s ago edge-update.timer
        edge-update.service
```

Рисунок 3.14 – Налаштування таймера для автоматичного запуску сервісу оновлення

Параметр OnBootSec=2min визначає запуск сервісу через дві хвилини після завантаження системи, а OnUnitActiveSec=10min встановлює інтервал повторного виконання кожні 10 хвилин. Після створення таймера виконується оновлення

конфігурації systemd, його активація та перевірка через команду systemctl list-timers.

3.3 Тестування системи

Після налаштування основних компонентів контейнерного стеку проведено тестування модуля безперервного розгортання та OTA-оновлення на ARM-вузлі. Оновлення запускається скриптом update.sh, який виконується вручну або автоматично через systemd-таймер edge-update.timer. На рисунку 3.15 наведено результат оновлення systemd-таймера.

Рисунок 3.15 – Оновленням таймер systemd

Стан роботи перевіряється через веб-ендпоінт сервісу, а результати фіксуються у лог-файл update.log (рисунок 3.16).

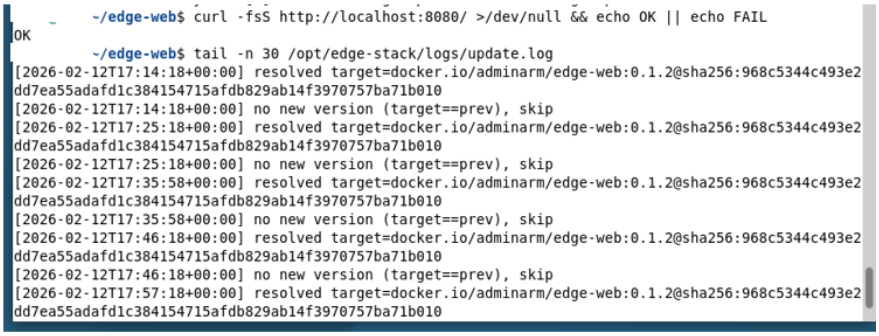


Рисунок 3.16 – Перевірка веб-ендпоінт сервісу

Додатково результати кожної спроби оновлення записуються в локальну базу даних deployments.db, де зберігаються відомості про події оновлення, статус виконання та результати перевірки працездатності сервісу. Також в таблиці

deployments.db фіксуються як успішні оновлення (OK), так і помилки на етапі pull (FAIL, reason=pull_failed) (рисунок 3.17).

```
        :~/edge-web$ sqlite3 /opt/edge-stack/data/deployments.db \
"SELECT deployment_id, time_start, time_end, result, reason
 FROM Deployment ORDER BY deployment_id DESC LIMIT 3;"
18|2026-02-12T15:38:12+00:00|2026-02-12T15:38:17+00:00|OK|
17|2026-02-12T15:29:19+00:00|2026-02-12T15:29:21+00:00|FAIL|pull_failed
16|2026-02-12T13:33:52+00:00|2026-02-12T13:33:58+00:00|OK|
        :~/edge-web$ sqlite3 /opt/edge-stack/data/deployments.db \
"SELECT deployment_id, check_time, status, details
 FROM HealthCheck
 WHERE deployment_id=(SELECT MAX(deployment_id) FROM Deployment)
 ORDER BY check_time DESC LIMIT 10;"
18|2026-02-12T15:38:17+00:00|OK|try=1
        :~/edge-web$
```

Рисунок 3.17 - Приклад записів deployments.db та health-check

Для імітації появи нової версії в registry зібрано та опубліковано multi-arch образ для платформ linux/amd64 і linux/arm64 у Docker Hub (рисунок 3.18). Після цього ARM-вузол отримує нову версію через скрипт update.sh, який виконує pull і оновлює стек командою up -d.

```
        :~/edge-web$ sed -i 's/version: v0\.1\.2/version: v0.1.3/' index.html
        :~/edge-web$ grep -n "version" index.html
6:    <p>version: v0.1.3</p>
          ~/edge-web$ git add index.html && git commit -m "bump version to 0.1.3"
[master 278afb4] bump version to 0.1.3
 1 file changed, 1 insertion(+), 1 deletion(-)
          ~/edge-web$ docker buildx build --platform linux/amd64,linux/arm64 \
  -t adminarm/edge-web:0.1.3 -t adminarm/edge-web:latest \
  --push .
[+] Building 11.3s (12/12) FINISHED                                  docker-container:multiarch
 => [internal] load build definition from Dockerfile                                      0.0s
 => => transferring dockerfile: 104B                                                      0.0s
 => [linux/arm64 internal] load metadata for docker.io/library/nginx:alpine               0.8s
 => [linux/amd64 internal] load metadata for docker.io/library/nginx:alpine               0.8s
 => [auth] library/nginx:pull token for registry-1.docker.io                              0.0s
 => [internal] load .dockerignore                                                         0.0s
 => => transferring context: 2B                                                           0.0s
 => CACHED [linux/amd64 1/2] FROM docker.io/library/nginx:alpine@sha256:1d13701a5f9f3fb01aaa  0.0s
 => => resolve docker.io/library/nginx:alpine@sha256:1d13701a5f9f3fb01aaa88cef2344d65b6b5bf6  0.0s
 => CACHED [linux/arm64 1/2] FROM docker.io/library/nginx:alpine@sha256:1d13701a5f9f3fb01aaa  0.0s
 => => resolve docker.io/library/nginx:alpine@sha256:1d13701a5f9f3fb01aaa88cef2344d65b6b5bf6  0.0s
 => [internal] load build context                                                         0.0s
 => => transferring context: 199B                                                         0.0s
 => [linux/amd64 2/2] COPY index.html /usr/share/nginx/html/index.html                    0.0s
```

Рисунок 3.18 - Появи нової версії в registry

Після появи нової версії виконується публікація програмного забезпечення у віддалений репозиторій Docker Hub. Мультиархітектурний образ уніфікує

розгортання периферійного вузла автоматично завантажуючи варіант образу, сумісний з архітектурою ARM64 (рисунок 3.19).

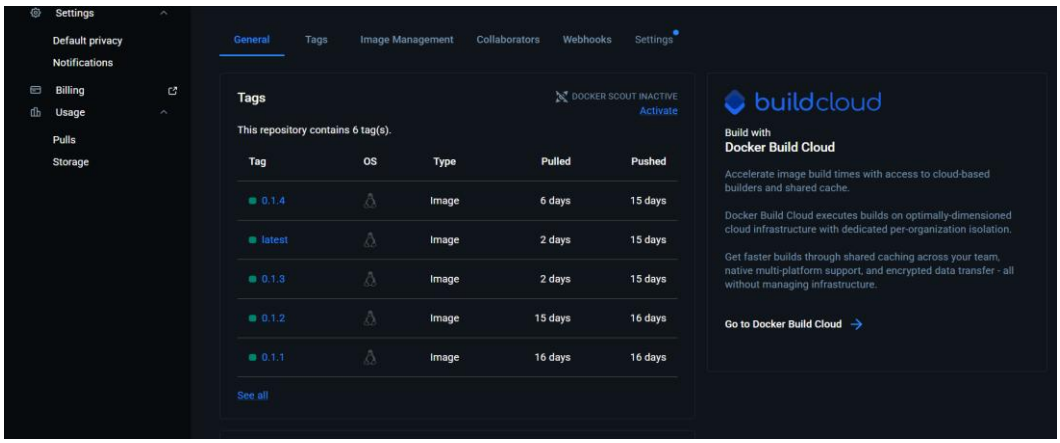


Рисунок 3.19 - Мультиархітектурного образи в Docker Hub

Далі було перевірено сценарій некоректного оновлення контейнерного стеку. Для цього використовувалася тестова версія сервісу, у якій прикладний інтерфейс перевірки стану не повертав коректної відповіді. В такому випадку перевірка працездатності завершується помилкою, після чого модуль автоматично повертає систему до попередньої стабільної версії. Після відновлення попередньої версії виконується повторна перевірка працездатності сервісу (рисунок 3.20).

```
[2026-02-13T21:29:07+00:00] rollback OK
        :~/edge-web$ grep -n "rollback\|health" /opt/edge-stack/logs/update.log | tail -n 30
49:[2026-02-10T16:27:44+00:00] update FAIL (health)
54:[2026-02-10T16:30:40+00:00] update FAIL (health)
74:[2026-02-11T15:30:23+00:00] health failed -> rollback to nginx:alpine
76:[2026-02-11T15:30:25+00:00] rollback FAIL
81:[2026-02-11T15:32:26+00:00] health failed -> rollback to nginx:alpine
83:[2026-02-11T15:32:28+00:00] rollback FAIL
92:[2026-02-11T15:34:35+00:00] health failed -> rollback to nginx:alpine
97:[2026-02-11T15:34:37+00:00] rollback OK
118:[2026-02-11T16:16:48+00:00] health failed -> rollback to nginx:alpine
123:[2026-02-11T16:16:51+00:00] rollback OK
350:[2026-02-13T21:20:25+00:00] health failed -> rollback to docker.io/adminarm/edge-web:0.1.3@sha2
56:ec3a4179fdd0357f31ff119b917b94da227f4a4dfd4c26ef3454570fb0c6030b
356:[2026-02-13T21:20:38+00:00] rollback OK
366:[2026-02-13T21:28:55+00:00] health failed -> rollback to docker.io/adminarm/edge-web:0.1.3@sha2
56:ec3a4179fdd0357f31ff119b917b94da227f4a4dfd4c26ef3454570fb0c6030b
372:[2026-02-13T21:29:07+00:00] rollback OK
```

Рисунок 3.21 - Помилка health-check та успішний rollback

Якщо інтерфейс перевірки працездатності повертає коректну відповідь, система вважає оновлення успішним. В локальній базі даних фіксується результат

OK, а сервіс продовжує роботу на новій версії контейнерного образу. Якщо сервіс не відповідає або перевірка завершується помилкою, модуль переходить до повернення попередньої стабільної версії. В цьому випадку відновлюється попередній контейнерний образ, Docker Compose-стек запускається повторно, після чого виконується додаткова перевірка працездатності. Якщо після повернення попередньої версії сервіс стає доступним, в журналі та базі даних фіксується результат ROLLBACK_OK. Якщо ж сервіс не відновлюється навіть після повернення до попередньої версії, стан системи позначається як критичний із результатом CRITICAL.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було досліджено проблему розгортання та підтримки сервісів на ARM edge-вузлах для IoT-пристроїв.

Було проаналізовано підходи до контейнеризації та безперервного розгортання для IoT/edge середовищ.

Було встановлено, що ручне оновлення є нестабільним і важко відтворюваним коли працюють декілька сервісів декілька. Контейнерний підхід спрощує підтримку, тому що сервіс постачається як образ із зафіксованими залежностями а запуск контролюється єдиним конфігураційним описом стеку.

Враховано особливість ARM-платформ для яких важливо мати multi-arch образи, щоб одна й та сама версія могла коректно працювати і на arm64 і на amd64.

На основі проведеного аналізу було розроблено проект контейнерного стеку для ARM-вузла, який включає:

- організацію запуску прикладного сервісу на Raspberry Pi в складі контейнерного стеку з відтворюваною конфігурацією;

- побудову ланцюжка доставки версій через container registry з застосуванням multi-arch образів (arm64/ amd64);

- реалізацію механізму OTA-оновлення на вузлі через скрипт update.sh, який виконує pull та up -d;

- реалізацію перевірки працездатності сервісу через HTTP-інтерфейс після виконання оновлення;

- реалізацію автоматичного повернення до попередньої стабільної версії у випадку, якщо перевірка працездатності сервісу завершується помилкою;

- ведення журналу подій оновлення у update.log та фіксацію результатів у локальній базі deployments.db (SQLite);

-організацію періодичного запуску оновлення через systemd (service/timer), щоб вузол міг оновлюватися без ручного запуску.

Було створено:

- контейнерний стек для ARM-вузла з прикладним web-сервісом;

- multi-arch pipeline збірки і публікації образів у Docker Hub за допомогою docker buildx;

- скрипт оновлення update.sh із логікою порівняння версій, застосування оновлення, перевірки endpoint та виконання rollback;

- конфігурацію systemd для автоматизації запуску оновлення за розкладом;

- базу deployments.db для збереження історії оновлень та результатів перевірок.

Тестування показало коректну роботу механізму оновлення в основних сценаріях.

Підтверджено сценарій SKIP, коли цільова версія збігається з поточною і оновлення не виконується.

Перевірено сценарій pull_failed, якщо заданий тег образу відсутній у registry, система фіксує помилку отримання образу..

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kong L., Tan J., Huang J., Chen G., Wang S., Jin X., Zeng P., Khan M. K., Das S. K. Edge-computing-driven Internet of Things: A Survey. *ACM Computing Surveys*. 2022. Vol. 55, No. 8. Article 174. DOI: https://doi.org/10.1145/3555308.

2. Urblik L., Kajati E., Papcun P., Zolotová I. Containerization in Edge Intelligence: A Review. *Electronics*. 2024. Vol. 13, No. 7. Article 1335. DOI: https://doi.org/10.3390/electronics13071335.

3. Čilić I., Krivić P., Podnar Žarko I., Kušek M. Performance Evaluation of Container Orchestration Tools in Edge Computing Environments. *Sensors*. 2023. Vol. 23, No. 8. Article 4008. DOI: https://doi.org/10.3390/s23084008.

4. Vaño R., Lacalle I., Sowiński P., S-Julián R., Palau C. E. Cloud-Native Workload Orchestration at the Edge: A Deployment Review and Future Directions. *Sensors*. 2023. Vol. 23, No. 4. Article 2215. DOI: https://doi.org/10.3390/s23042215.

5. Wang Z., Goudarzi M., Aryal J., Buyya R. Container Orchestration in Edge and Fog Computing Environments for Real-Time IoT Applications. 2022. URL: https://arxiv.org/abs/2203.05161 (дата звернення: 07.05.2026).

6. Usman M., Ferlin S., Brunstrom A., Taheri J. A Survey on Observability of Distributed Edge & Container-Based Microservices. *IEEE Access*. 2022. Vol. 10. P. 86904–86919. DOI: https://doi.org/10.1109/ACCESS.2022.3193102.

7. Jin X., He S., Chen Y. Container Migration for Edge Computing in Industrial Internet with Joint Latency Reduction and Reliability Enhancement. *Scientific Reports*.

2024. Vol. 14. Article 25394. DOI: https://doi.org/10.1038/s41598-024-77086-2.

8. Al-Doghman F., Moustafa N., Khalil I., Sohrabi N., Tari Z., Zomaya A. Y. AI-Enabled Secure Microservices in Edge Computing: Opportunities and Challenges. *IEEE Transactions on Services Computing*. 2023. Vol. 16, No. 2. P. 1485–1504. DOI: https://doi.org/10.1109/TSC.2022.3155447.

9. Devi Priya V. S., Sethuraman S. C., Khan M. K. Container Security: Precaution Levels, Mitigation Strategies, and Research Perspectives. *Computers & Security*. 2023. Vol. 135. Article 103490. DOI: https://doi.org/10.1016/j.cose.2023.103490.

10. Ajith V., Cyriac T., Chacko C., Ali K. Analyzing Docker Vulnerabilities through Static and Dynamic Analysis. *Journal of Cybersecurity and Privacy*. 2024. Vol. 5, No. 3. Article 26. URL: https://www.mdpi.com/2624-831X/5/3/26 (дата звернення: 07.05.2026).

11. Haq M. S., Tosun A. Ş., Korkmaz T. Security Analysis of Docker Containers for ARM Architecture. *Proceedings of the IEEE/ACM Symposium on Edge Computing*. 2022. P. 224–236. DOI: https://doi.org/10.1109/SEC54971.2022.00025.

12. El Jaouhari S., Bouvet E. Secure Firmware Over-The-Air Updates for IoT: Survey, Challenges, and Discussions. *Internet of Things*. 2022. Vol. 18. Article 100508. DOI: https://doi.org/10.1016/j.iot.2022.100508.

13. Noprianto, Funabiki N. та ін. A Proposal of Secure and Automated Over-the-Air Firmware Update Mechanism for IoT Devices Using Continuous Integration and Continuous Delivery. *Sensors*. 2026. Vol. 26, No. 5. Article 1535. URL: https://www.mdpi.com/1424-8220/26/5/1535 (дата звернення: 07.05.2026).

14. Villegas M. M. та ін. DeOTA-IoT: A Techniques Catalog for Designing Over-the-Air Update Systems for IoT. *Sensors*. 2026. Vol. 26, No. 1. Article 193. URL: https://www.mdpi.com/1424-8220/26/1/193 (дата звернення: 07.05.2026).

15. Park C. Y. та ін. Secure and Lightweight Firmware Over-the-Air Update Method for Resource-Constrained IoT Devices. *Electronics*. 2025. Vol. 14, No. 8. Article 1583. URL: https://www.mdpi.com/2079-9292/14/8/1583 (дата звернення: 07.05.2026).

16. Docker Engine Documentation. Docker Docs. URL: https://docs.docker.com/engine/ (дата звернення: 07.05.2026).

17. Docker Compose Documentation. Docker Docs. URL: https://docs.docker.com/compose/ (дата звернення: 07.05.2026).

18. Multi-platform builds. Docker Docs. URL: https://docs.docker.com/build/building/multi-platform/ (дата звернення: 07.05.2026).

19. systemd.timer — Timer Unit Configuration. Freedesktop.org. URL: https://www.freedesktop.org/software/systemd/man/systemd.timer.html (дата звернення: 07.05.2026).

20. K3s Documentation. Lightweight Kubernetes. URL: https://docs.k3s.io/ (дата звернення: 07.05.2026).

21. Nomad Architecture. HashiCorp Developer. URL: https://developer.hashicorp.com/nomad/docs/architecture (дата звернення: 07.05.2026).

22. Podman Documentation. Podman. URL: https://podman.io/docs (дата звернення: 07.05.2026).

23. SQLite Documentation. SQLite. URL: https://sqlite.org/ (дата звернення: 07.05.2026).

24. Загвойський Р., Казимира І., Ткачук К., Дмитрів О. Масштабовані механізми віддаленого оновлення програмного забезпечення в периферійній робототехніці. *Вимірювальна та обчислювальна техніка в технологічних процесах*. 2025. № 84(4). С. 7–15. DOI: https://doi.org/10.31891/2219-9365-2025-84-1.

25. Степанов Д. С., Сенів М. М. Інтеграція захищеної інфраструктури, контейнеризації та DevSecOps для підвищення надійності роботи вебпорталів. *Науковий вісник НЛТУ України*. 2024. № 34(5). С. 144–150. DOI: https://doi.org/10.36930/40340519.

26. Степанов Д. С., Сенів М. М. Застосування технологій незмінної інфраструктури для забезпечення прозорого масштабування без простоїв у вебпорталах. *Вчені записки ТНУ імені В. І. Вернадського. Серія: Технічні науки*. 2025. Т. 36(75), № 3. Ч. 2. DOI: https://doi.org/10.32782/2663-5941/2025.3.2/49.

27. Ворона Д. О. Дослідження методів оптимізації продуктивності хмарних веб-додатків контейнеризованих у Kubernetes. *Радіоелектроніка та молодь у XXI столітті*. Харків : ХНУРЕ, 2025. Т. 6. С. 244–246. URL: https://openarchive.nure.ua/entities/publication/f6be08e0-ccc6-49ff-85c4-b9dbab3351b2 (дата звернення: 07.05.2026).

28. Романенко Б. В. Метод забезпечення надійності контейнеризованого додатку в розподіленому середовищі Docker : кваліфікаційна робота бакалавра. Харків : ХНУРЕ, 2025. URL: https://openarchive.nure.ua/entities/publication/5e35874f-12d5-429d-94c7-687cd65f4883 (дата звернення: 07.05.2026).

29. Романцова В. В. Застосування технологій контейнеризації для створення середовищ розгортання та запуску додатків : кваліфікаційна робота бакалавра.

Харків : ХНУРЕ, 2025. URL: https://openarchive.nure.ua/entities/publication/0dc382f9-f258-4442-9ff8-524bdd8ab3ea (дата звернення: 07.05.2026).

30. Прес Р. Д. Аналіз методів контейнеризації програмного забезпечення. *Радіоелектроніка та молодь у XXI столітті*. Харків : ХНУРЕ, 2024. Т. 6. С. 248–249. DOI: https://doi.org/10.30837/IYF.IIS.2024.248.

31. Бойко В. С. Створення Docker образів IoT систем у PyCharm та Visual Studio Code. *Радіоелектроніка та молодь у XXI столітті*. Харків : ХНУРЕ, 2024. Т. 3. С. 77–79. DOI: https://doi.org/10.30837/IYF.IRTTPI.2024.77.

32. Семчишин П. М. Організація взаємодії мікросервісів у задачах розробки веб-застосунків : кваліфікаційна робота магістра. Тернопіль : ТНТУ ім. І. Пулюя, 2025. URL: https://elartu.tntu.edu.ua/handle/lib/50852 (дата звернення: 07.05.2026).

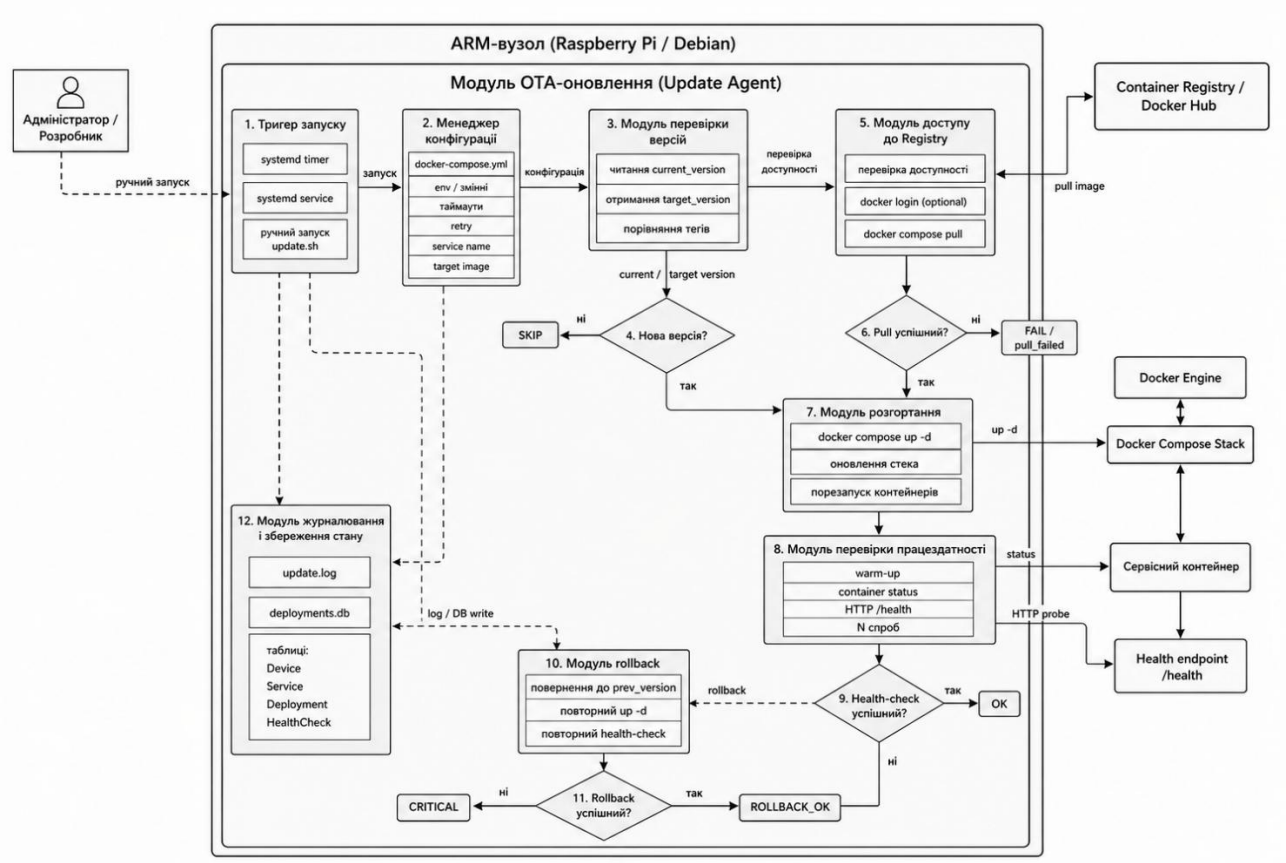
33. Покидько О. В. Реалізація системи моніторингу вебдодатків : кваліфікаційна робота. Тернопіль : ТНТУ ім. І. Пулюя, 2026. URL: https://elartu.tntu.edu.ua/bitstream/lib/51500/1/Pokydko_Oleksandr_SBmz-62_2025.pdf (дата звернення: 07.05.2026).

34. Комар М.П., Саченко А.О., Васильків Н.М., Гладій Г.М., Коваль В.С., Ліп’яніна-Гончаренко Х.В. Методичні рекомендації до виконання кваліфікаційної роботи з освітньо-професійної програми «Комп’ютерні науки» спеціальності 122 «Комп’ютерні науки» за першим (бакалаврським) рівнем вищої освіти. Тернопіль: ЗУНУ, 2024. 52 с.

35. Островерхов В. М., Біловус Л. І., Возьний К. З., Луцишин О. О., Монастирський Г. Л., Надвиничний С. А., Питель С. В., Шандрук С. К. Загальні методичні рекомендації з підготовки, оформлення, захисту та оцінювання кваліфікаційних робіт здобувачів вищої освіти першого (бакалаврського) і другого (магістерського) рівнів. Тернопіль: ЗУНУ, 2024. 83 с.

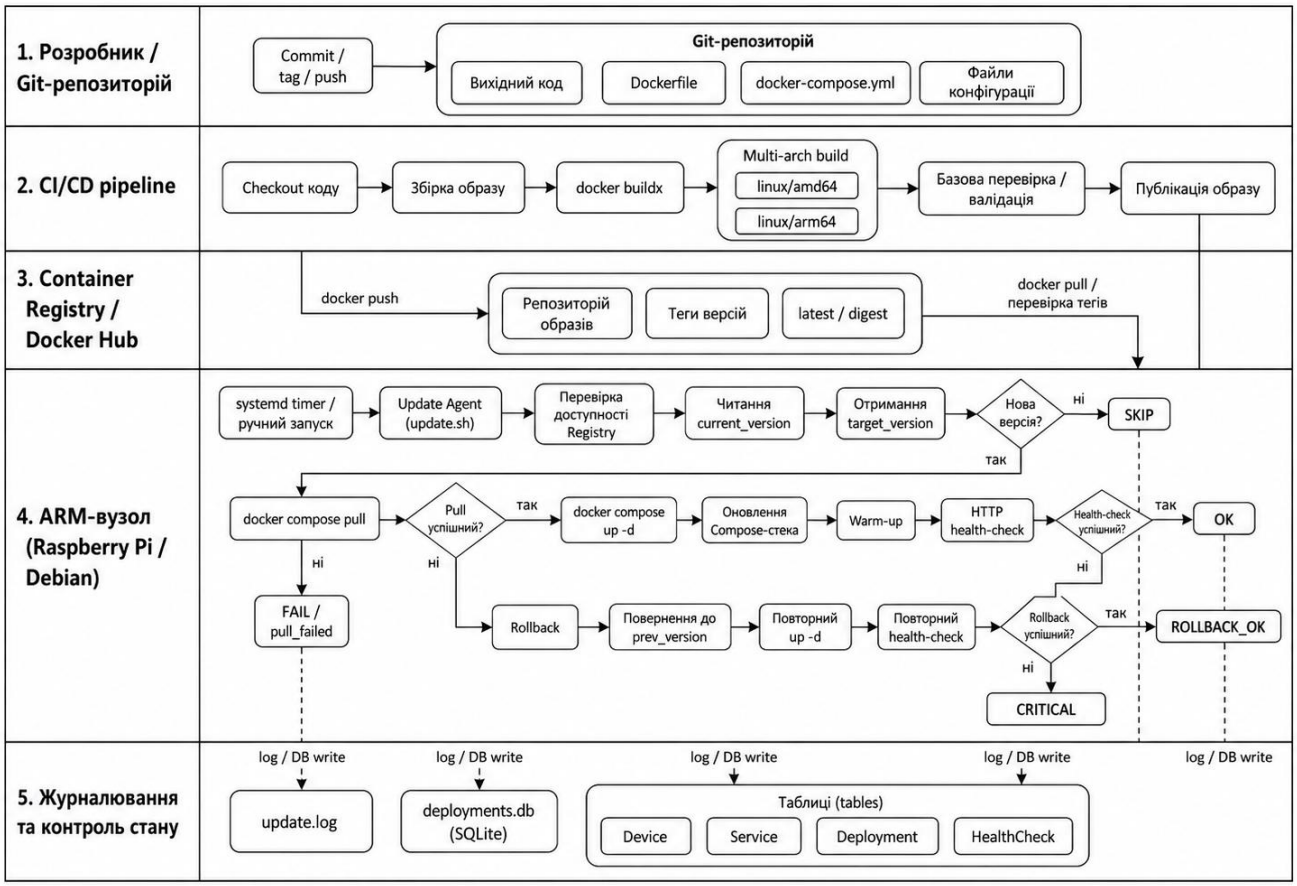
Додаток А

Детальна архітектура модуля OTA-оновлення



Додаток Б

Діаграма розподілу повного циклу CI/CD та OTA-оновлення контейнерного сервісу



Додаток В

Програмний код модуля OTA-оновлення контейнерного сервісу

```

Лістинг В.1 – Створення структури каталогів проєкту

sudo mkdir -p /opt/edge-stack/stack
sudo mkdir -p /opt/edge-stack/data
sudo mkdir -p /opt/edge-stack/logs


Лістинг В.2 – Файл docker-compose.yml для запуску тестового сервісу

cat > /opt/edge-stack/stack/docker-compose.yml <<'YAML'
services:
  web:
    image: ${WEB_IMAGE:-nginx:alpine}
    ports:
      - "8080:80"
    healthcheck:
      test: ["CMD-SHELL", "wget -qO- http://localhost/ >/dev/null || exit 1"]
      interval: 10s
      timeout: 3s
      retries: 5
      start_period: 10s
YAML


Лістинг В.3 – Початковий файл змінних середовища

cat > /opt/edge-stack/stack/.env <<'ENV'
WEB_IMAGE=nginx:alpine
ENV


Лістинг В.4 – Створення systemd-сервісу для автозапуску контейнерного стеку

sudo tee /etc/systemd/system/edge-stack.service > /dev/null <<'UNIT'
[Unit]
Description=Edge stack (docker-compose)
After=network-online.target docker.service
Wants=network-online.target
Requires=docker.service

[Service]
Type=oneshot
RemainAfterExit=yes
WorkingDirectory=/opt/edge-stack/stack
ExecStart=/usr/bin/docker-compose up -d
ExecStop=/usr/bin/docker-compose down
TimeoutStartSec=0

[Install]
WantedBy=multi-user.target
UNIT


Лістинг В.5 – Створення SQLite-бази даних для журналу оновлень

sqlite3 /opt/edge-stack/data/deployments.db <<'SQL'
PRAGMA foreign_keys = ON;

CREATE TABLE IF NOT EXISTS Device (
    device_id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    arch TEXT,
    os_name TEXT,
    ip_addr TEXT,
    last_seen TEXT
);

CREATE TABLE IF NOT EXISTS Service (
    service_id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    current_version TEXT,
    health_status TEXT,
    updated_at TEXT
);

CREATE TABLE IF NOT EXISTS Deployment (
    deployment_id INTEGER PRIMARY KEY AUTOINCREMENT,
    service_name TEXT NOT NULL,
    prev_version TEXT,
    target_version TEXT NOT NULL,
    time_start TEXT NOT NULL,
    time_end TEXT,
    result TEXT NOT NULL,
    reason TEXT
);

```

```
CREATE TABLE IF NOT EXISTS HealthCheck (
  check_id INTEGER PRIMARY KEY AUTOINCREMENT,
  deployment_id INTEGER NOT NULL,
  check_time TEXT NOT NULL,
  status TEXT NOT NULL,
  details TEXT,
  FOREIGN KEY (deployment_id) REFERENCES Deployment(deployment_id) ON DELETE CASCADE
);

SELECT name FROM sqlite_master WHERE type='table' ORDER BY name;
SQL


Лістинг В.6 – Скрипт OTA-оновлення update.sh

sudo tee /opt/edge-stack/update.sh > /dev/null <<'SH'
#!/usr/bin/env bash
set -euo pipefail

STACK_DIR="/opt/edge-stack/stack"
DB="/opt/edge-stack/data/deployments.db"
LOG="/opt/edge-stack/logs/update.log"

SERVICE_NAME="web"
DEFAULT_TARGET="nginx:alpine"
HEALTH_URL="http://localhost:8080/"
HEALTH_RETRIES=3
HEALTH_DELAY=2

ts() {
  date -Iseconds
}

log() {
  echo "[$(ts)] $*" | tee -a "$LOG"
}

sql_escape() {
  printf "%s" "$1" | sed "s/'/''/g"
}

mkdir -p "$(dirname "$LOG")"
mkdir -p "$(dirname "$DB")"
```

```
cd "$STACK_DIR"

HTTP_CODE="$(curl -sS -o /dev/null -w "%{http_code}" -I https://registry-1.docker.io/v2/ || echo 000)"

if [[ "$HTTP_CODE" != "200" && "$HTTP_CODE" != "401" ]]; then
  log "registry unreachable (http=$HTTP_CODE)"
  exit 1
fi

TARGET_IMAGE="$DEFAULT_TARGET"

if [[ -f "$STACK_DIR/target-image.txt" ]]; then
  TARGET_IMAGE="$(tr -d '\r\n' < "$STACK_DIR/target-image.txt")"
fi

resolve_arm64() {
  local img="$1"

  if [[ "$img" == *@sha256:* ]]; then
    echo "$img"
    return 0
  fi

  local resolved=""
  resolved="$(docker buildx imagetools inspect "$img" 2>/dev/null | awk '
    $1=="Name:" {name=$2}
    $1=="Platform:" && $2 ~ /^linux\/arm64/ {print name; exit}
  ')"

  if [[ -n "${resolved:-}" ]]; then
    echo "$resolved"
  else
    echo "$img"
  fi
}

PREV_IMAGE="$(grep -E '^WEB_IMAGE=' "$STACK_DIR/.env" 2>/dev/null | cut -d= -f2- || true)"
PREV_IMAGE="${PREV_IMAGE:-$DEFAULT_TARGET}"

TARGET_RESOLVED="$(resolve_arm64 "$TARGET_IMAGE")"
PREV_RESOLVED="$(resolve_arm64 "$PREV_IMAGE")"

log "resolved target=$TARGET_RESOLVED"
```

```
if [[ "$TARGET_RESOLVED" == "$PREV_RESOLVED" ]]; then
  NOW="$(ts)"

  sqlite3 "$DB" "
    INSERT INTO Deployment(service_name, prev_version, target_version, time_start, time_end, result, reason)
    VALUES(
      '$(sql_escape "$SERVICE_NAME")',
      '$(sql_escape "$PREV_RESOLVED")',
      '$(sql_escape "$TARGET_RESOLVED")',
      '$NOW',
      '$NOW',
      'SKIP',
      'target_equals_current'
    );
  "

  log "no new version (target==prev), skip"
  exit 0
fi

START_TIME="$(ts)"

log "update start (prev=$PREV_RESOLVED target=$TARGET_RESOLVED)"

DEPLOYMENT_ID="$(
sqlite3 "$DB" "
  INSERT INTO Deployment(service_name, prev_version, target_version, time_start, time_end, result, reason)
  VALUES(
    '$(sql_escape "$SERVICE_NAME")',
    '$(sql_escape "$PREV_RESOLVED")',
    '$(sql_escape "$TARGET_RESOLVED")',
    '$START_TIME',
    '',
    'RUNNING',
    ''
  );
  SELECT last_insert_rowid();
"
)"

echo "WEB_IMAGE=$TARGET_RESOLVED" > "$STACK_DIR/.env"

if ! docker-compose pull >>"$LOG" 2>&1; then
  sqlite3 "$DB" "
    UPDATE Deployment
    SET time_end='$(ts)', result='FAIL', reason='pull_failed'
    WHERE deployment_id=$DEPLOYMENT_ID;
  "

  log "pull failed"
  exit 1
fi

if ! docker-compose up -d >>"$LOG" 2>&1; then
  sqlite3 "$DB" "
    UPDATE Deployment
    SET time_end='$(ts)', result='FAIL', reason='up_failed'
    WHERE deployment_id=$DEPLOYMENT_ID;
  "

  log "compose up failed"
  exit 1
fi

sleep "$HEALTH_DELAY"

OK=0

for i in $(seq 1 "$HEALTH_RETRIES"); do
  if curl -fsS "$HEALTH_URL" >/dev/null; then
    sqlite3 "$DB" "
      INSERT INTO HealthCheck(deployment_id, check_time, status, details)
      VALUES($DEPLOYMENT_ID, '$(ts)', 'OK', 'try=$i');
    "

    OK=1
    break
  else
    sqlite3 "$DB" "
      INSERT INTO HealthCheck(deployment_id, check_time, status, details)
      VALUES($DEPLOYMENT_ID, '$(ts)', 'FAIL', 'try=$i');
    "

    sleep "$HEALTH_DELAY"
  fi
done

if [[ "$OK" -eq 1 ]]; then
```

```
    sqlite3 "$DB" "
      UPDATE Deployment
      SET time_end='$(ts)', result='OK', reason=''
      WHERE deployment_id=$DEPLOYMENT_ID;
    "

    sqlite3 "$DB" "
      INSERT INTO Service(name, current_version, health_status, updated_at)
      VALUES(
        '$(sql_escape "$SERVICE_NAME")',
        '$(sql_escape "$TARGET_RESOLVED")',
        'healthy',
        '$(ts)'
      )
      ON CONFLICT(name) DO UPDATE SET
        current_version=excluded.current_version,
        health_status=excluded.health_status,
        updated_at=excluded.updated_at;
    "

    log "update OK"
    exit 0
  fi

  log "health failed -> rollback to $PREV_RESOLVED"

  echo "WEB_IMAGE=$PREV_RESOLVED" > "$STACK_DIR/.env"

  docker-compose up -d >>"$LOG" 2>&1 || true

  sleep "$HEALTH_DELAY"

  if curl -fsS "$HEALTH_URL" >/dev/null; then
    sqlite3 "$DB" "
      UPDATE Deployment
      SET time_end='$(ts)', result='ROLLBACK_OK', reason='health_failed_rollback_ok'
      WHERE deployment_id=$DEPLOYMENT_ID;
    "

    sqlite3 "$DB" "
      INSERT INTO Service(name, current_version, health_status, updated_at)
      VALUES(
        '$(sql_escape "$SERVICE_NAME")',
        '$(sql_escape "$PREV_RESOLVED")',
        'healthy',
        '$(ts)'
      )
      ON CONFLICT(name) DO UPDATE SET
        current_version=excluded.current_version,
        health_status=excluded.health_status,
        updated_at=excluded.updated_at;
    "

    log "rollback OK"
    exit 0
  else
    sqlite3 "$DB" "
      UPDATE Deployment
      SET time_end='$(ts)', result='CRITICAL', reason='health_failed_rollback_failed'
      WHERE deployment_id=$DEPLOYMENT_ID;
    "

    sqlite3 "$DB" "
      INSERT INTO Service(name, current_version, health_status, updated_at)
      VALUES(
        '$(sql_escape "$SERVICE_NAME")',
        '$(sql_escape "$PREV_RESOLVED")',
        'unhealthy',
        '$(ts)'
      )
      ON CONFLICT(name) DO UPDATE SET
        current_version=excluded.current_version,
        health_status=excluded.health_status,
        updated_at=excluded.updated_at;
    "

    log "rollback FAIL"
    exit 1
  fi
  SH

  sudo chmod +x /opt/edge-stack/update.sh


  Лістинг В.7 – systemd-сервіс для запуску update.sh

  sudo tee /etc/systemd/system/edge-update.service > /dev/null <<'UNIT'
  [Unit]
```

```
Description=Edge OTA update service
After=network-online.target docker.service edge-stack.service
Wants=network-online.target
Requires=docker.service

[Service]
Type=oneshot
ExecStart=/opt/edge-stack/update.sh
TimeoutStartSec=0
UNIT


Лістинг В.8 – systemd-таймер для періодичного запуску оновлення

sudo tee /etc/systemd/system/edge-update.timer > /dev/null <<'UNIT'
[Unit]
Description=Run edge update periodically

[Timer]
OnBootSec=2min
OnUnitActiveSec=10min
Unit=edge-update.service
Persistent=true

[Install]
WantedBy=timers.target
UNIT


Лістинг В.9 – Активація сервісів systemd

sudo systemctl daemon-reload

sudo systemctl enable --now edge-stack.service
sudo systemctl enable --now edge-update.timer

sudo systemctl status edge-stack.service --no-pager
sudo systemctl list-timers --all | grep edge-update


Лістинг В.10 – Приклад задання цільового образу для оновлення

echo "nginx:alpine" | sudo tee /opt/edge-stack/stack/target-image.txt

```

Додаток Д
Апробація отриманих результатів

ДОВІДКА

ПРО ПРИЙНЯТТЯ ТЕЗ ДО ПУБЛІКАЦІЇ

08.06.2026

Шановний(і) авторе(и):
Шлапак Марта Сергіївна,

Організаційний комітет з радістю повідомляє, що тези «КОНТЕЙНЕРНИЙ СТЕК ДЛЯ ARM-ВУЗЛІВ З БЕЗПЕРЕРВНИМ РОЗГОРТАННЯМ ДЛЯ IOT-ПРИСТРОЇВ» прийняті до публікації в збірнику за матеріалами X Міжнародної студентської наукової конференції «Діджиталізація науки як виклик сьогодення» (12.06.2026, м. Миколаїв, Україна).

Опубліковані тези будуть доступні з 12.06.2026 за посиланням:
https://archive.liga.science/index.php/conference-proceedings/issue/view/inter-12.06.2026

. . . . . .

Електронні сертифікати учасників конференції та подяки науковим керівникам будуть доступні за посиланням вище з 12 червня. Розсилка замовлених друкованих примірників, сертифікатів та подяк відбудеться до 26 червня.

Конференцію зареєстровано Державною науковою установою «УкрІНТЕІ» в базі даних науково-технічних заходів України та інформаційному бюлетені «План проведення наукових, науково-технічних заходів в Україні» (Посвідчення № 123 від 26.01.2026).

З повагою,

Директор Молодіжної наукової ліги
Голова оргкомітету конференції
ІГОР КОРЕНЮК

КОНТЕЙНЕРНИЙ СТЕК ДЛЯ ARM-ВУЗЛІВ З БЕЗПЕРЕРВНИМ РОЗГОРТАННЯМ ДЛЯ IOT-ПРИСТРОЇВ

Шлапак Марта Сергіївна
Бакалавр спеціальності 122 – Комп'ютерні науки
Західноукраїнський національний університет, Україна
Науковий керівник: Осолінський Олександр Романович
к.т.н., доцент кафедри інформаційно-обчислювальних систем і управління
Західноукраїнський національний університет, Україна

Вступ

IoT-пристрої та периферійні вузли все частіше використовуються в системах моніторингу, розумного дому, промислових датчиках і локальних сервісах. Значна частина таких вузлів працює на ARM-платформах. Водночас ці пристрої мають обмежений обсяг оперативної пам'яті, менш продуктивний процесор, microSD-носій замість повноцінного диска, а також можуть працювати в умовах нестабільного живлення або інтернет-з'єднання.

За таких умов ручне оновлення програмного забезпечення стає незручним і ризикованим. Тому актуальною є побудова контейнерного стеку, який забезпечує відтворюване розгортання та віддалене оновлення працездатності після оновлення та автоматичне повернення до попередньої стабільної версії в разі збою.

Архітектура системи

Запропонована система поєднує середовище розробки, автоматизовану збірку, реєстр контейнерних образів та ARM-вузол, на якому безпосередньо виконуються сервіси. Основою практичної реалізації обрано Raspberry Pi, а як програмні засоби використано Docker Engine, Docker Compose, systemd, SQLite та віддалений реєстр контейнерних образів.

Загальний цикл роботи починається з Git-репозиторію, де зберігаються вихідний код, Dockerfile та файл docker-compose.yml. Після внесення змін запускається процес збирання контейнерного образу. Для підтримки різних апаратних платформ формується багатоплатформний образ, сумісний з amd64 та arm64. Готовий образ публікується у Docker Hub або іншому реєстрі, звідки ARM-вузол може отримати нову версію через стандартну операцію pull.

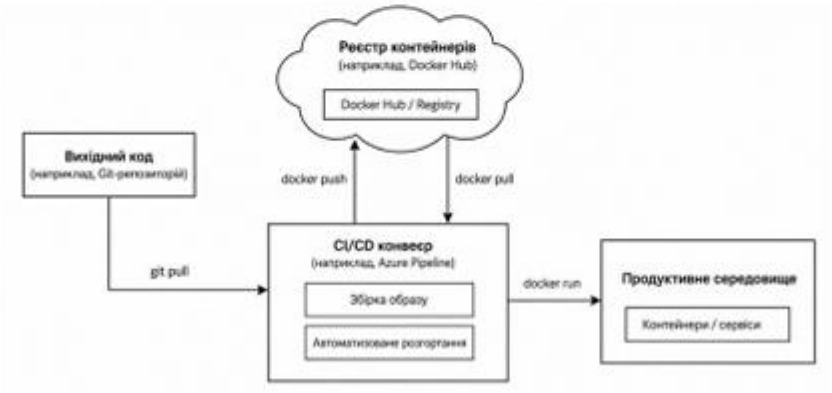


Рис. 1. **Загальна архітектура компонентів системи**

На ARM-вузлі сервіси запускаються, як єдиний стек через Docker Compose. Окремим елементом виступає модуль оновлення, реалізований у вигляді сценарію update.sh, який запускається вручну або автоматично через systemd timer.

Архітектура включає такі основні компоненти:

– репозиторій проекту, у якому зберігаються вихідний код, Dockerfile та конфігурація Docker Compose;

– середовище автоматизованого збирання, яке формує multi-arch образ і публікує його у віддалений реєстр;

– реєстр контейнерних образів, що зберігає доступні версії сервісу та використовується ARM-вузлом для оновлення;

– Raspberry Pi з Docker Engine і Docker Compose, на якому виконується прикладний сервіс;

– модуль OTA-оновлення з перевіркою /health, журналюванням і механізмом rollback.

Алгоритм OTA-оновлення

Алгоритм роботи модуля оновлення орієнтований на безпечне застосування нової версії без втрати працездатності вузла. Спочатку сценарій перевіряє доступність мережі та реєстру контейнерних образів. Якщо з'єднання відсутнє, спроба оновлення не виконується, а подія фіксується в журналі. Такий підхід не змінює поточну конфігурацію за умов нестабільного зв'язку.

Після підтвердження доступності реєстру система порівнює поточну версію сервісу з цільовою. Якщо нової версії немає, фіксується результат SKIP. Якщо ж виявлено новий образ, виконується завантаження через docker compose pull і повторний запуск стеку командою docker compose up -d. Після оновлення виконується прикладна перевірка через HTTP-інтерфейс /health.

Якщо сервіс відповідає коректно, оновлення завершується зі статусом OK. Якщо перевірка не проходить, сценарій автоматично повертає попередню стабільну версію контейнерного образу.

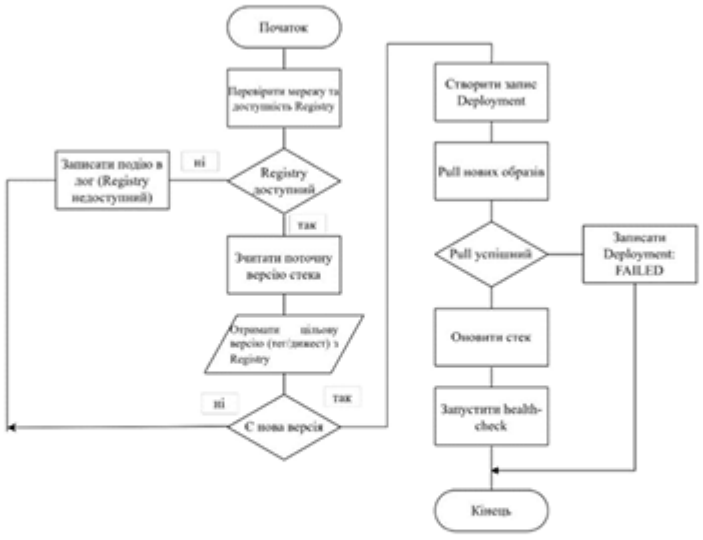


Рис. 2. **Схема алгоритму оновлення на Raspberry Pi**

Після відновлення виконується повторна перевірка стану сервісу. В разі успішного повернення фіксується статус ROLLBACK_OK, а якщо попередня версія також не відновлює працездатність, система позначає стан як критичний.

Висновки. було розроблено контейнерний стек для ARM-вузла з підтримкою безперервного розгортання та оновлення контейнеризованого сервісу. Запропонований підхід поєднує Docker Compose для керування стеком, multi-arch образи для сумісності з ARM-платформами, systemd timer для періодичного запуску оновлення, HTTP-перевірку працездатності та SQLite для збереження історії розгортань. Така збірка дозволяє оновлювати IoT-вузол без фізичного доступу до пристрою, зменшує ризик простою після невдалого оновлення та забезпечує зрозумілий журнал подій для подальшого аналізу адміністратором.

Список використаних джерел:

1. Kong L., Tan J., Huang J., Chen G., Wang S., Jin X., Zeng P., Khan M. K., Das S. K. Edge-computing-driven Internet of Things: A Survey // ACM Computing Surveys. – 2022. – Vol. 55, No. 8. – Article 174.
2. Urblik L., Kajati E., Papcun P., Zolotová I. Containerization in Edge Intelligence: A Review // Electronics. – 2024. – Vol. 13, No. 7. – Article 1335.
3. Docker Engine Documentation. Docker Docs. URL: https://docs.docker.com/engine/ (дата звернення: 07.04.2026).
4. Docker Compose Documentation. Docker Docs. URL: https://docs.docker.com/compose/ (дата звернення: 07.04.2026).
5. Multi-platform builds. Docker Docs. URL: https://docs.docker.com/build/building/multi-platform/ (дата звернення: 07.04.2026).
6. systemd.timer — Timer unit configuration. Ubuntu Manpages. URL: https://manpages.ubuntu.com/manpages/questing/man5/systemd.timer.5.html (дата звернення: 07.04.2026).