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

КВАЛІФІКАЦІЙНА РОБОТА

Програмно-технічний засіб реалізації cloud-середовища на базі Raspberry Pi
Назва теми

Рівень вищої освіти *перший (бакалаврський)*

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

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

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

Шифр КвРКІ 22031.22.01.10 ПЗ

Виконав здобувач IV курсу, група КІ2-22-1 _____ Олена КОБЕЦЬ
Підпис Ініціали, прізвище

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

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

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

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

Рівень вищої освіти ПЕРШИЙ (БАКАЛАВРСЬКИЙ)

Галузь знань 12 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

Спеціальність 123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Освітня програма «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ ТА ПРОГРАМУВАННЯ»

ЗАТВЕРДЖУЮ
Завідувачка кафедри КІІС
Ольга ПАВЛОВА

" 10 " 01 2026 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Кобець Олені Дмитрівні
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Програмно-технічний засіб реалізації cloud-середовища на базі Raspberry Pi

Керівник проекту (роботи) Лисекно Сергій Миколайович, д.т.н., проф.
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 01.03.2026 р. № 5

2. Термін подання здобувачем роботи на кафедру 01.06.2026 р.

3. Вихідні дані до роботи Завдання на кваліфікаційну роботу

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

Теоретичні основи досліджуваної задачі

Проєктування програмно-технічного засобу реалізації cloud-середовища на базі raspberry PI

Програмно-апаратна реалізація cloud-середовища на базі raspberry PI

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

Узагальнена схема взаємодії компонентів засобу

Блок-схеми алгоритмів функціонування систем

Інтерфейс системи

6. Консультанти розділів кваліфікаційної роботи

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

7. Дата видачі завдання «_10_»_____01_____2026 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	10.01.2026	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.02.2026	виконано
3	Робота над розділом 1 – дослідження предметної області та постановка задачі	01.03.2026	виконано
4	Робота над розділом 2 – вибір компонентів для проєктування системи адаптивного застосування моніторингових елементів розвідувального БПЛА	01.04.2026	виконано
5	Робота над розділом 3 – проєктування системи адаптивного застосування моніторингових елементів розвідувального БПЛА	29.04.2026	виконано
6	Оформлення пояснювальної записки згідно вимог	25.05.2026	виконано
7	Попередній захист ВКР	26.05.2025	виконано
8	Захист ВКР на засіданні ЕК	Червень 2026 року	

Здобувач_____ Олна КОБЕЦЬ
Підпис Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи_____ Сергій ЛИСЕНКО
Підпис Ім'я, ПРІЗВИЩЕ

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 22031.22.01.10 ПЗ	Пояснювальна записка	65		
			Графічні матеріали			
2		КвРКІ 22031.22.01.10 Е8	Узагальнена схема взаємодії компонентів засобу	1		
3		КвРКІ 22031.22.01.10 Е8	Блок-схеми алгоритмів функціонування систем	1		
4		КвРКІ 22031.22.01.10 Е8	Інтерфейс системи	1		

Зм	Арк	№ докум	Підпис	Дата	КвРКІ 22031.22.01.10 ВП		
Розробив		Кобець			Відомість проекту	Літера	Аркуш 1 / Аркушів 1
Перевір.		Лисенко				У	
Н. контр.		Лисенко				ХНУ, КІ2-22-1	
Затв.		Павлова		01.06			

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Програмно-технічний засіб реалізації cloud-середовища на базі Raspberry Pi».

Автор роботи: Олена КОБЕЦЬ.

Керівник роботи: Сергій ЛИСЕНКО.

Пояснювальна записка: 65 с., 1 рис., 3 дод., 50 джерел.

Графічна частина: 3 креслення.

АРХІТЕКТУРА, БАЗА ДАНИХ, СЕРВЕР, СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, РОЗГОРТАННЯ, RASPBERRY PI.

Кваліфікаційна робота бакалавра присвячена розробці програмно-технічного засобу реалізації cloud-середовища на базі Raspberry Pi. Актуальність теми зумовлена зростанням вимог до надійності, енергоефективності та безперервності функціонування cloud-середовищ.

Метою роботи є проєктування, реалізація та тестування cloud-середовищ.

Підпис здобувача

30.05.2026

Дата

ЗМІСТ

Зміст 2
Вступ 4
1 Теоретичні основи досліджуваної задачі 6
1.1 Поняття та еволюція cloud-обчислень 6
1.2 Поняття та еволюція cloud-обчислень 8
1.3 Моделі розгортання 11
1.4 Типові сценарії використання 14
1.5 Висновки до першого розділу 18
2 Проєктування програмно-технічного засобу реалізації cloud-середовища на базі raspberry PI 21
2.1 Загальна архітектура персонального cloud-середовища на базі Raspberry Pi 21
2.2 Апаратна складова програмно-технічного засобу 28
2.3 Системне програмне забезпечення 31
2.4 Серверний компонент програмно-технічного засобу 35
2.5 Клієнтський компонент програмно-технічного засобу 39
2.4 Висновки до другого розділу 42
3 Програмно-апаратна реалізація cloud-середовища на базі raspberry PI 45
3.1 Підготовка апаратної платформи Raspberry Pi 45
3.2 Розгортання системного оточення 48
3.3 Розгортання HomeDock OS 50
3.4 Реалізація серверної частини 52
3.5 Налаштування підсистеми безпеки .....
3.6 Налаштування мережевої доступності 59
3.7 Тестування та верифікація працездатності 61
3.8 Результати реалізації 63

Висновки 68
Перелік джерел посилань 69
Додаток А Опис модулів пристрою 75
Додаток Б Копія креслення «Узагальнена схема взаємодії компонентів засобу» 76
Додаток В Копія креслення «Блок-схеми алгоритмів функціонування систем» 77
Додаток Г Копія креслення «Інтерфейс системи» 78

ВСТУП

Cloud-обчислення стали базовою технологічною основою сучасних інформаційних систем, оскільки дозволяють організовувати доступ до обчислювальних ресурсів і програмних сервісів як до мережевої послуги [1-3].

У межах цього підходу дані, прикладні сервіси та середовища виконання розміщуються централізовано й надаються користувачам через стандартні інтерфейси, що спрощує використання цифрових інструментів у навчанні, роботі та повсякденних задачах [4-6].

Практичний ефект cloud-парадигми проявляється у можливості швидко розгортати й конфігурувати інформаційні сервіси, підтримувати спільну роботу з даними та забезпечувати доступ до них із різних пристроїв і локацій у межах визначених політик безпеки [7-9].

Значущість cloud-обчислень зростає через те, що сучасні процеси все частіше спираються на дистанційну взаємодію та розподілені команди. Для освітніх установ це означає організацію доступу до навчальних матеріалів, лабораторних середовищ і проєктних ресурсів без прив’язки до конкретного комп’ютерного класу [10-12].

Для малого бізнесу cloud-сервіси стають основою спільного документообігу, керування робочими процесами, підтримки клієнтських систем і забезпечення віддаленої роботи співробітників. Для інженерних та домашніх лабораторій хмарні підходи використовуються як спосіб формувати кероване середовище для тестування технологій, розгортання сервісів і організації експериментів із повторюваними конфігураціями та контрольованим доступом [13].

Окремий інтерес становить використання cloud-концепції у форматі приватних інсталяцій на доступному апаратному забезпеченні, зокрема на одноплатних комп’ютерах Raspberry Pi [14]. Такий підхід дозволяє реалізувати cloud-середовище в межах локальної мережі або навчальної лабораторії,

забезпечуючи типовий набір мережевих сервісів для зберігання даних, синхронізації, спільного доступу та віддаленої роботи [15]. У цьому контексті хмарні обчислення виступають не лише як інфраструктурна технологія великих дата-центрів, а як прикладна модель організації сервісів, яку можна адаптувати під обмежені ресурси, конкретні потреби користувачів і вимоги до керування та безпеки [16].

1 ТЕОРЕТИЧНІ ОСНОВИ ДОСЛІДЖУВАНОЇ ЗАДАЧІ

1.1 Поняття та еволюція cloud-обчислень

Хмарні обчислення у сучасному розумінні описують модель, за якої обчислювальні ресурси та програмні сервіси надаються користувачеві як мережеві послуги, доступні через стандартні протоколи та інтерфейси [17-19]. У центрі цієї моделі знаходиться не конкретний фізичний сервер або локально встановлене програмне забезпечення, а абстрагований пул ресурсів, з яким користувач взаємодіє через каталог сервісів, вебінтерфейс або API. Ресурси можуть включати обчислювальну потужність, мережеві компоненти, сховище даних, платформи виконання застосунків і прикладні функції. Важливою рисою є те, що користувач оперує ресурсами як логічними сутностями, а фізична реалізація, розміщення та внутрішня структура інфраструктури залишаються поза межами його безпосереднього керування [20, 21]. Історично передумови хмарних обчислень формувалися поступово. На ранніх етапах комп'ютерні системи будувалися навколо централізованих обчислювальних машин і термінального доступу, де багато користувачів спільно використовували одну інфраструктуру. Пізніше, з поширенням персональних комп'ютерів, домінував підхід локального використання програм і даних, а потреба у спільному доступі та віддаленій роботі компенсувалася файловими серверами та корпоративними мережами [22-24]. Далі розвиток інтернет-технологій, вебсервісів та концепцій сервіс-орієнтованої архітектури підсилив тенденцію до перенесення функцій у мережеве середовище. Вирішальним етапом стала зрілість віртуалізації, яка дозволила розділяти один фізичний сервер на кілька ізольованих середовищ виконання, керувати ними централізовано та автоматизувати їх створення і видалення. Сукупність цих чинників складає розподілені дата-центри, віртуалізація, стандартизація мережевих протоколів і автоматизація керування – сформувала модель, яка нині визначається як "хмара" [25].

Для формалізації поняття хмарних обчислень використовують набір ключових характеристик, що описують їх поведінку як сервісної моделі. Однією з базових характеристик є самообслуговування за запитом. Вона означає, що користувач або адміністратор може самостійно ініціювати виділення ресурсів, зміну параметрів сервісу чи підключення додаткових компонентів у момент виникнення потреби, без необхідності ручної участі постачальника послуги. Практично це реалізується через панелі керування та програмні інтерфейси, що дозволяють автоматизовано створювати екземпляри сервісів, керувати обліковими записами, конфігураціями та політиками доступу [26]. Другою характеристикою виступає широкий мережевий доступ. Вона описує доступність сервісів через мережу за допомогою стандартних механізмів, які підтримують різні типи клієнтських пристроїв та програмних агентів. Йдеться не про конкретну технологію підключення, а про принцип сервіс має бути спроєктований як мережевий, з передбачуваною взаємодією через протоколи, що забезпечують сумісність і можливість інтеграції [27-29]. Наступною характеристикою є об’єднання ресурсів у пул. У хмарній моделі інфраструктурні компоненти розглядаються як спільний ресурсний фонд, з якого виділяються логічні ресурси відповідно до запитів користувачів [30]. Пул керується таким чином, щоб забезпечувати ізоляцію між різними користувачами та сервісами на рівні доступу й виконання, при цьому конкретне фізичне розміщення ресурсів не є фіксованим для користувача [31]. У межах цієї характеристики часто згадують багатокористувацьке середовище, коли одна фізична база використовується декількома незалежними споживачами, але їхні робочі контексти розмежовані політиками, віртуалізацією, контейнеризацією та іншими механізмами ізоляції [32-34]. Еластичність характеризує здатність системи змінювати обсяг виділених ресурсів у відповідь на зміну навантаження. У хмарному контексті масштабування розглядається як керована зміна параметрів сервісу, зокрема кількості екземплярів, обсягу пам’яті, ресурсів CPU, пропускної здатності або сховища. Еластичність часто проявляється у формі горизонтального

масштабування через додавання або вилучення екземплярів сервісу та у формі вертикального масштабування через зміну ресурсних лімітів конкретного екземпляра. Ключовим у визначенні є саме можливість керованого й повторюваного пристосування під навантаження, а не конкретний спосіб її реалізації [35, 36].

Останньою базовою характеристикою є вимірюваність сервісу (measured service). Вона означає, що використання ресурсів контролюється та реєструється за допомогою метрик, що відображають споживання обчислювальних потужностей, сховища, мережевого трафіку, кількості запитів або часу роботи сервісу [37]. Така вимірюваність забезпечується системами моніторингу та обліку, які збирають дані про стан і використання ресурсів, надаючи прозорість як для постачальника, так і для споживача. У рамках цієї характеристики важливо, що ресурс не існує як “безконтрольний”, а його використання формалізоване в показниках і подіях, які можуть бути використані для керування, аудитів або автоматизованих правил [38]. У підсумку, визначення хмарних обчислень опирається на поєднання абстрагованого надання ресурсів як послуг та зазначених характеристик, які задають їхню операційну модель. Саме ці характеристики дозволяють відмежувати “хмару” від просто віддаленого сервера або традиційного хостингу, оскільки описують спосіб надання, доступу, керування та обліку ресурсів як стандартизованого сервісу [39].

1.2 Поняття та еволюція cloud-обчислень

У межах хмарних обчислень модель надання послуг описує, який саме рівень абстракції отримує споживач і де проходить межа відповідальності між тим, хто надає сервіс, і тим, хто ним користується [40]. Ця класифікація сформувалась як реакція на ускладнення IT-інфраструктур і потребу формалізувати, що саме є “послугою” у хмарі: окремі інфраструктурні ресурси, середовище виконання застосунків або готовий прикладний продукт.

Найчастіше розрізняють три базові моделі: IaaS, PaaS і SaaS, які можна розглядати як послідовні рівні сервісної абстракції від ближчого до апаратури до ближчого до кінцевої функціональності [41].

Модель Infrastructure as a Service (IaaS) трактує “послугу” як надання обчислювальних і мережевих ресурсів, а також ресурсів зберігання даних у вигляді керованих об’єктів інфраструктури. У цьому випадку споживач взаємодіє з віртуальними машинами, віртуальними мережами, дисковими томами, адресацією, правилами доступу та іншими інфраструктурними примітивами. Ресурси описуються параметрами, що відповідають апаратному рівню, наприклад кількістю процесорних ядер, обсягом оперативної пам’яті, розміром сховища або конфігурацією мережевих інтерфейсів. Керування здійснюється через панелі адміністрування та API, а життєвий цикл інфраструктурних об’єктів формується операціями створення, запуску, зупинки, масштабування, копіювання образів і зміни конфігурацій. У контексті еволюції хмар IaaS пов’язана з розвитком віртуалізації та підходів до програмно визначуваної інфраструктури, коли фізичні ресурси дата-центру або локального кластера подаються користувачеві як логічні сутності, придатні до автоматизованого керування [42].

Модель Platform as a Service (PaaS) визначає “послугу” як кероване середовище для розгортання та виконання застосунків, у якому інфраструктурні деталі приховані за платформним рівнем. Тут фокус переноситься з управління окремими віртуальними машинами на управління застосунком і його конфігурацією, залежностями та політиками виконання. Платформа зазвичай включає запуск, веб-сервер або сервер застосунків, системи керування конфігурацією, механізми журналювання, маршрутизацію запитів, а також інтегровані сервіси даних, які споживаються через стандартизовані інтерфейси. Споживач працює з артефактами рівня застосунку: вихідним кодом, контейнерними образами, декларативними маніфестами, змінними оточення, параметрами підключення до сервісів і правилами доступу. PaaS як підхід

розвивався паралельно з поширенням вебзастосунків, DevOps-практик і контейнеризації, оскільки ці технології дозволили описувати процес розгортання у вигляді повторюваних сценаріїв і перетворювати “платформу” на набір керованих сервісів, що підтримують життєвий цикл застосунку [43].

Модель Software as a Service (SaaS) визначає “послугу” як готовий прикладний продукт, доступний через браузер, мобільний застосунок або програмний інтерфейс. У цій моделі користувач взаємодіє переважно з прикладною логікою, функціями та даними на рівні бізнес-процесів або користувацьких сценаріїв. Налаштування зводяться до параметрів предметної області, конфігурації облікових записів, політик доступу, інтеграцій та правил обробки даних. Технічні аспекти розгортання, оновлення, масштабування і підтримки середовища виконання віднесені до внутрішньої реалізації сервісу. Формування SaaS як домінантної моделі для багатьох класів програмного забезпечення стало наслідком розвитку вебтехнологій, стандартів безпечної автентифікації, моделей багатокористувацької ізоляції даних та загального переходу від ліцензування “коробкових” продуктів до доступу “як сервісу” [44].

Окрім трьох базових моделей, у літературі та практиці зустрічаються похідні або деталізовані підходи, які уточнюють рівень сервісу. Наприклад, Function as a Service (FaaS) розглядає як послугу виконання окремих функцій або обробників подій у керованому середовищі, де одиницею розгортання стає не застосунок у цілому, а функція з визначеним тригером запуску. Також часто виділяють Backend as a Service (BaaS), де споживач отримує готові бекенд-компоненти на кшталт автентифікації, баз даних, сховищ об’єктів чи push-повідомлень, звертаючись до них через API. Ці похідні моделі відображають тенденцію до подальшого дроблення функціональності на сервіси й мікросервіси та зростання ролі керованих компонентів, які вбудовуються у розробку та експлуатацію систем без необхідності проектувати інфраструктурний шар кожного разу заново.

Для опису IaaS, PaaS і SaaS принципово важливо, що різниця між ними визначається не конкретним брендом чи продуктом, а рівнем абстракції та тим, на якому шарі зосереджена взаємодія користувача із сервісом. У практичних реалізаціях межі між моделями можуть бути нечіткими, оскільки сучасні платформи часто поєднують декілька рівнів одночасно, наприклад надаючи інфраструктурні ресурси разом з керованими сервісами даних або пропонуючи прикладний продукт, який підтримує інтеграції та розширення на рівні платформи. Проте базова класифікація IaaS–PaaS–SaaS залишається корисною для формального опису складу сервісу, ролей користувача і структури хмарного рішення, зокрема коли потрібно обґрунтувати вибір підходу для реалізації приватного cloud-середовища на обмежених апаратних ресурсах, таких як Raspberry Pi.

1.3 Моделі розгортання

Модель розгортання у контексті cloud-обчислень описує, де фізично та організаційно розміщується хмарна інфраструктура, кому вона належить, хто має до неї адміністративний доступ і як визначаються межі середовища з точки зору контролю, ізоляції та політик [45-47]. На відміну від моделей надання послуг, які фіксують рівень абстракції сервісу (інфраструктура, платформа, програмний продукт), моделі розгортання уточнюють “простір існування” хмари та характер її спільного використання. У практиці найчастіше використовують чотири узагальнені моделі: private, public, hybrid і community, кожна з яких пов’язана з певним способом організації ресурсів, керуванням і режимом доступу.

Приватна хмара розуміється як хмарне середовище, призначене для використання однією організацією або однією визначеною адміністративною доменною групою, коли межі доступу та керування задаються внутрішніми політиками власника. Така хмара може бути фізично розміщена на власних

майданчиках організації або на орендованих ресурсах, однак ключовим є те, що інфраструктура та сервіси логічно виділені під одного споживача, а адміністрування виконується або власними фахівцями, або уповноваженим підрядником. Ідентичність визначається насамперед моделлю керування: хто встановлює правила автентифікації, авторизації, мережевої сегментації, політики зберігання даних, резервного копіювання, журналювання та оновлень. Для приватної хмари також характерне формування єдиного адресного та ідентифікаційного простору, де інтеграція з каталогами користувачів, корпоративними сертифікаційними центрами та внутрішніми мережевими зонами задається як частина базового дизайну [48].

Публічна хмара є моделлю, за якої хмарні ресурси та сервіси надаються широкому колу споживачів на інфраструктурі провайдера, що функціонує як спільна основа для багатьох незалежних клієнтів. Ключовою рисою тут виступає належність інфраструктури зовнішньому постачальнику та формалізована процедура доступу споживачів через публічні інтерфейси керування. Межі ізоляції між клієнтами реалізуються на рівні віртуалізації, мережевих політик і систем керування доступом, а сам споживач оперує ресурсами в межах виділеного йому логічного простору, який може бути оформлений як окремий обліковий запис, тенант або проєкт. Для публічнлї хмари характерним є те, що управління фізичною інфраструктурою, базовими сервісами, а часто й значною частиною платформного шару, залишається на стороні провайдера, тоді як споживач зосереджується на конфігураціях у межах наданих механізмів. Публічність у цьому випадку не означає “відкритість даних”, а означає публічну доступність сервісу як комерційної або загальнодоступної пропозиції для багатьох клієнтів.

Гібридна хмара описує організацію, у якій використовується комбінація двох або більше хмарних середовищ різних типів, що зберігають власну ідентичність, але зв’язані між собою технологічно та організаційно таким чином, щоб забезпечити переносимість даних або застосунків, або керований розподіл

компонентів між середовищами. Найтиповіший випадок є поєднання private та public хмар, коли частина систем працює у внутрішньому середовищі організації, а частина є на інфраструктурі провайдера. Гібридність у строгому сенсі передбачає не просто паралельне використання двох хмар, а наявність узгоджених механізмів взаємодії: зв'язності мереж (через VPN, приватні канали або тунелювання), синхронізації ідентичностей (федерація, єдині домени автентифікації), керування ключами та сертифікатами, інтегрованого моніторингу та політик доступу до даних. Архітектура часто будується так, щоб компоненти могли бути розподілені за вимогами до розміщення даних, потребами інтеграції з локальними ресурсами або залежностями між сервісами, при цьому відносини між середовищами регламентуються як частина загальної моделі експлуатації.

Спільнотна хмара визначає хмарне середовище, яке поділяється кількома організаціями або групами користувачів, що мають спільні вимоги до політик, стандартів, регуляторних умов або цілей використання. На відміну від public cloud, де провайдер надає універсальну платформу для довільних клієнтів із загальними правилами, хмара передбачає, що споживачі пов'язані певною спільною ознакою, яка впливає на правила експлуатації середовища.

Такими ознаками можуть виступати приналежність до однієї галузі, участь у спільній освітній чи науковій програмі, підпорядкування єдиному регуляторному режиму або необхідність дотримання однакових профілів безпеки та обміну даними. Спільнотна хмара може бути розгорнута та керована однією з організацій-учасників, групою учасників спільно або стороннім оператором, однак її відмінність полягає у визначеній "закритій множині" споживачів та узгодженому наборі політик, які обслуговують саме потреби цієї спільноти. У такому середовищі суттєвими є питання спільного адміністрування, розмежування прав керування між організаціями, а також формалізація правил доступу до спільних ресурсів і даних [49].

Еволюція моделей розгортання відображає загальний рух IT від локальних обчислень до мережевих сервісів і від індивідуальних серверів до керованих пулів ресурсів. Спочатку організації намагалися відтворити підхід «власного дата-центру» у вигляді приватної хмари, формалізуючи віртуалізовану інфраструктуру як сервіс усередині домену організації. Паралельно розвивалися публічні провайдери, які масштабували інфраструктуру та стандартизували доступ, зробивши хмарні сервіси загальнодоступними як ринкову пропозицію.

У міру зростання складності систем і появи потреби розподіляти компоненти за різними середовищами, сформувався підхід як архітектурна відповідь на поєднання внутрішніх ресурсів із зовнішніми. Для секторів, де важливими є спільні правила, сумісність і довіра всередині групи організацій, виникла як компроміс між приватністю “власного” середовища і спільним використанням ресурсів.

У контексті бакалаврської роботи про cloud-середовище на базі Raspberry Pi ці моделі розгортання доречно інтерпретувати як організаційні сценарії: від повністю локальної приватної інфраструктури в межах домашньої або навчальної мережі до інтегрованих схем, де локальний вузол або мінікластер взаємодіє з зовнішніми сервісами, залишаючись частиною гібридного середовища, або як спільнотна інфраструктура для визначеного кола користувачів, наприклад навчальної групи чи лабораторії.

1.4 Типові сценарії використання

Типові сценарії використання cloud-обчислень у прикладних сферах формуються навколо повторюваних потреб: забезпечення віддаленого доступу до даних і сервісів, централізоване розміщення навчальних або робочих інструментів, організація спільної роботи з контрольованими ролями, а також розгортання прикладних систем із передбачуваною процедурою керування. У цих сценаріях “хмара” розуміється не як абстрактне поняття, а як сукупність

сервісів, що надаються через мережу і підтримують конкретні процеси: навчальні, адміністративні, виробничі чи експериментальні. При цьому характер сценаріїв залежить від контексту використання: в освіті фокус часто зміщується на доступність навчальних середовищ і лабораторних робіт, у малому бізнесі на підтримку операційних процесів і обміну даними, а у домашній лабораторії на відтворюваність експериментів, тестування та інтеграцію різнорідних компонентів [50].

В освітньому середовищі cloud-сервіси зазвичай залучаються для підтримки навчального процесу як на рівні контенту, так і на рівні інструментів. Поширеним є сценарій організації єдиного простору для зберігання і поширення навчальних матеріалів, де студенти отримують доступ до лекцій, методичних вказівок, практичних завдань, наборів даних та шаблонів проєктів через вебінтерфейс або синхронізацію клієнтських застосунків. Такий простір часто прив’язаний до ролей і груп, що відображають академічну структуру, а доступ може регламентуватися семестровими рамками або навчальними курсами. Іншим типовим сценарієм є використання хмарних середовищ для практичних і лабораторних робіт, коли студенту або групі надається ізольоване середовище виконання для конкретної дисципліни: наприклад, віртуальна машина з налаштованим стеком, контейнерне середовище або вебплатформа для розробки та тестування. У такому підході викладач може стандартизувати конфігурацію середовища, а студенти працюють з однаковими інструментами та залежностями незалежно від власних пристроїв. У межах ІТ-спеціальностей також поширене застосування хмарних систем для навчання основам адміністрування, мереж, кібербезпеки і DevOps, де важливими є контроль доступу, журналювання дій, відтворюваність конфігурацій і можливість розгортати типові сервісні компоненти (вебсервери, бази даних, сховища, системи моніторингу) у вигляді керованої інфраструктури.

У малому бізнесі сценарії використання cloud-обчислень часто будуються навколо повсякденних інформаційних потоків і підтримки операційної

діяльності. Типовою задачею є організація спільного файлового простору для документів, договорів, бухгалтерських файлів, презентацій, медіаконтенту та внутрішніх регламентів з можливістю розмежування доступу між відділами або ролями співробітників. Окремою лінією використання є підтримка внутрішніх комунікацій і проєктної взаємодії, коли хмарні сервіси забезпечують зберігання артефактів проєктів, контроль версій документів, узгодження змін та роботу з шаблонними процесами. Для бізнесу з клієнтськими зверненнями характерним стає розміщення CRM/ERP-подібних систем або сервісів заявок, де доступ до прикладного інтерфейсу здійснюється через мережу, а дані зберігаються централізовано з визначеними політиками доступу. Також зустрічається сценарій публікації корпоративних вебресурсів, внутрішніх панелей (dashboards) або API для інтеграції з партнерськими системами. В окремих випадках хмара використовується як середовище для резервного копіювання й зберігання архівів з формалізованими правилами доступу, зокрема коли необхідно централізувати історичні дані та забезпечити контрольований доступ до них для обмеженого кола осіб.

У домашній лабораторії cloud-підхід зазвичай реалізується як керований набір сервісів на власному обладнанні, що відтворює принципи “приватної хмари” в мініатюрі. Типовий сценарій тут пов’язаний з організацією персонального сховища даних з доступом із різних пристроїв і з можливістю синхронізації, коли користувач інтегрує файлові сервіси, вебдоступ і клієнтські застосунки в єдину систему. Інший поширений сценарій є створення полігону для тестування серверних технологій, де на одному або кількох вузлах розгортаються контейнери або віртуальні середовища для перевірки конфігурацій, розробки та експериментів із мережевими сервісами. У цьому контексті cloud-логіка проявляється через каталог сервісів, декларативні конфігурації, автоматизоване розгортання і розмежування доступу до різних компонентів лабораторії. Також характерним є використання домашньої інфраструктури як платформи для медіасервісів, систем резервного копіювання,

домашньої автоматизації та моніторингу, де окремі компоненти взаємодіють як сервісна екосистема і можуть бути доступні локально або віддалено через контрольовані канали доступу.

У сфері наукових досліджень і інженерних експериментів cloud-середовища застосовуються як платформа для виконання обчислень, зберігання й обробки експериментальних даних, а також для розгортання інструментів аналізу, що мають доступ через вебінтерфейс. Типовим є сценарій, коли група дослідників формує спільний простір для наборів даних, результатів моделювань і скриптів, а обчислювальні задачі виконуються в ізольованих середовищах, що відтворюють однакові залежності та версії бібліотек. У навчально-наукових лабораторіях cloud-підхід часто використовується для забезпечення доступу студентів і співробітників до спеціалізованих інструментів, які складно або недоцільно розгортати на кожному робочому місці, а також для підтримки життєвого циклу проєктів, де потрібні повторювані середовища тестування та інтеграції.

У сфері розробки програмного забезпечення типовими є сценарії використання хмар для середовищ розгортання, тестування та інтеграції, коли інфраструктура керується як набір сервісів, а застосунки публікуються в керованих середовищах. Поширені практики включають розміщення репозиторіїв, систем безперервної інтеграції, артефактних сховищ, середовищ для staging/production, а також сервісів спостережуваності, що збирають логи та метрики. У таких сценаріях хмара виступає платформою для стандартизації процесів розгортання, керування конфігураціями та доступу до компонентів через формалізовані інтерфейси.

У державному та муніципальному секторі cloud-сервіси часто використовуються для надання електронних послуг, підтримки документообігу, організації робочих просторів для установ і забезпечення доступу співробітників до прикладних систем з різних локацій. Тут сценарії зазвичай включають рольові моделі доступу, інтеграцію з реєстрами та журналювання дій користувачів,

оскільки робота з даними регламентується процедурами та внутрішніми політиками. У медичній і соціальній сферах cloud-підхід використовується для розміщення інформаційних систем, реєстраційних платформ, порталів для запису та доступу до даних, де важливими є контроль доступу, ведення історії змін і взаємодія між різними підрозділами.

У виробничих і логістичних процесах хмарні системи залучаються як середовище для збору даних від обладнання, диспетчеризації, обліку ресурсів і підтримки аналітики. У таких випадках cloud-сценарій часто поєднує периферійні пристрої або локальні вузли збору даних із центральними сервісами, які зберігають, агрегують і надають доступ до даних через інтерфейси для операторів або інтеграцій. Подібні підходи використовуються і в IoT-проєктах, де частина функцій розташована ближче до джерела даних, а частина у централізованому середовищі, що забезпечує керування, облік і доступ. У кожній із зазначених сфер cloud-сценарії можна описувати через те, які сервіси використовуються як “базові компоненти” середовища та як організовано доступ до них: через вебінтерфейси, клієнтські програми, API або інтеграції між системами. Для тематики побудови cloud-середовища на базі Raspberry Pi ці сценарії є релевантними як приклади того, які функціональні ролі має виконувати локальна приватна хмара: надання доступу до файлів і даних, підтримка спільної роботи, розгортання сервісів для навчальних або інженерних задач, а також інтеграція з мережевими механізмами віддаленого доступу та контролю користувачів.

1.5 Висновки до першого розділу

Постановка задачі розроблення програмно-технічного засобу реалізації cloud-середовища на базі Raspberry Pi полягає у створенні керованого локального хмарного середовища, яке забезпечує надання мережевих сервісів зберігання та доступу до даних, а також організацію віддаленої взаємодії користувачів із цими

сервісами в межах визначених політик безпеки. Об’єктом розроблення виступає програмно-технічний комплекс, що поєднує апаратну платформу Raspberry Pi з операційною системою та набором системного і прикладного програмного забезпечення, налаштованого для роботи як приватна хмара у локальній мережі з можливістю контрольованого зовнішнього доступу.

Метою розроблення є проєктування та реалізація рішення, яке дозволяє розгорнути cloud-середовище на ресурсно обмеженій ARM-платформі з відтворюваною процедурою інсталяції та конфігурування. У межах цієї мети необхідно визначити цільову архітектуру середовища, обрати модель розгортання та технологічний стек, встановити вимоги до продуктивності й надійності, а також описати механізми адміністрування, моніторингу та захисту доступу. Для досягнення мети формулюється сукупність взаємопов’язаних задач. Першочергово необхідно проаналізувати типові підходи до побудови приватних хмарних середовищ і визначити, які з них є придатними для Raspberry Pi з урахуванням апаратних обмежень процесорної архітектури, обсягу оперативної пам’яті та характеристик підсистеми введення-виведення. На основі цього аналізу слід сформувати структурну схему програмно-технічного засобу, що описує склад компонентів, їх функціональні ролі та взаємодію на рівні мережевих протоколів і сервісів, включаючи компоненти зберігання даних, доступу користувачів, автентифікації, вебінтерфейсу, термінального або API-керування, а також підсистеми журналювання й моніторингу.

Далі необхідно визначити вимоги до організації мережевого доступу в локальному та віддаленому режимі. У межах постановки задачі мають бути описані сценарії доступу користувачів із внутрішньої мережі та сценарії доступу ззовні, де обов’язково враховується необхідність захищених каналів зв’язку, коректної роботи доменних і сертифікаційних механізмів, а також обмеження доступу на рівні мережевих правил. Паралельно формулюються вимоги до моделі користувачів і прав доступу, щоб система могла розмежовувати доступ

до ресурсів, забезпечувати керування обліковими записами й підтримувати аудит ключових дій.

Окремим блоком постановки задачі є визначення вимог до надійності зберігання даних і відновлюваності. Для цього потрібно задати політики резервного копіювання, описати базові процедури відновлення після збоїв і визначити, як саме у системі фіксуватимуться критичні події та стани. У межах розроблення також необхідно передбачити механізми контролю працездатності сервісів, збору метрик, аналізу журналів і формування повідомлень про відхилення в роботі, оскільки cloud-середовище передбачає постійну доступність сервісів у межах заданих умов експлуатації.

З огляду на те, що розроблюваний об’єкт є програмно-технічним засобом, у постановці задачі важливо визначити, яким способом буде забезпечено відтворюваність інсталяції та конфігурації. Це означає необхідність розробити документацію і, за можливості, сценарії автоматизованого розгортання, які дозволяють повторити встановлення системи на аналогічній платформі без ручного налаштування кожного компонента. Також слід визначити критерії приймання результатів, що дозволяють перевірити коректність роботи cloud-середовища в межах заданих сценаріїв: доступність сервісів у локальній мережі, автентифікація користувачів, операції з файлами/даними, стійкість до типових помилок конфігурації, збереження працездатності після перезавантаження та відновлення з резервної копії.

Задача бакалаврської роботи полягатиме у розробленні та експериментальній перевірці програмно-технічного засобу на базі Raspberry Pi, який реалізуватиме приватне cloud-середовище з визначеним складом сервісів, контрольованим мережевим доступом, системою керування користувачами, механізмами захисту, журналюванням і підтримкою процедур резервного копіювання та відновлення, а також матиме описану й повторювану методику розгортання та адміністрування.

2 ПРОЄКТУВАННЯ ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ РЕАЛІЗАЦІЇ CLOUD-СЕРЕДОВИЩА НА БАЗІ RASPBERRY PI

2.1 Загальна архітектура персонального cloud-середовища на базі Raspberry Pi

Розроблюваний програмно-технічний засіб реалізації cloud-середовища на базі Raspberry Pi являє собою комплексну систему, що об’єднує апаратну мікрокомп’ютерну платформу та багаторівневий стек програмного забезпечення для розгортання, керування та безпечного надання доступу до набору хмарних сервісів у межах домашньої або корпоративної мережі. У даному підрозділі визначено концептуальну модель розроблюваного засобу, описано його багаторівневу архітектуру, наведено схему взаємодії апаратних і програмних компонентів та обґрунтовано обрані архітектурні рішення.

Під хмарним середовищем у роботі розуміється сукупність обчислювальних, мережевих та сховищних ресурсів, що надають користувачеві набір сервісів (зберігання файлів, мультимедійні сервіси, системи резервного копіювання, застосунки продуктивності тощо) з можливістю доступу до них з довільного пристрою через веб-інтерфейс.

Концептуальна модель розроблюваного засобу складається з сутностей:

- фізичний обчислювальний вузол на базі Raspberry Pi, який виконує роль сервера;
- ядро операційної системи cloud-середовища, що здійснює керування життєвим циклом сервісів;
- ізольовані контейнеризовані застосунки, які реалізують конкретні користувацькі сервіси;
- веб-інтерфейс адміністрування, доступний з будь-якого пристрою;
- підсистеми безпеки, мережевої доступності та моніторингу.

Характерною рисою такої моделі є суміщення ролей кінцевого користувача й адміністратора в одній особі, що висуває підвищені вимоги до

зручності керування. У зв’язку з цим ядро cloud-середовища повинно забезпечувати автоматизацію типових адміністративних операцій: встановлення застосунків із готового каталогу, оновлення образів, перенаправлення портів, отримання й продовження SSL-сертифікатів, моніторинг використання ресурсів.

До функціональних вимог до розроблюваного програмно-технічного засобу належать: підтримка одночасного функціонування декількох ізольованих користувацьких сервісів на одному фізичному вузлі; керування сервісами через веб-інтерфейс без необхідності використання командного рядка; захищений віддалений доступ з Інтернету з використанням протоколу HTTPS; автоматичне виявлення пристрою в локальній мережі через multicast DNS; робота в умовах обмежених обчислювальних ресурсів, характерних для одноплатних комп’ютерів; підтримка архітектур ARM та ARM64, властивих платформі Raspberry Pi; модульність, що дозволяє нарощувати функціональність без модифікації ядра системи.

Переліченим вимогам відповідає відкрите програмне рішення HomeDock OS, яке обрано за основу розроблюваного програмно-технічного засобу. Архітектура розроблюваного програмно-технічного засобу побудована за класичним принципом багаторівневого розшарування, у якому кожний наступний рівень використовує абстракції, надані попереднім, і надає власний інтерфейс для вищого рівня. Такий підхід дозволяє локалізувати зміни всередині окремого рівня, спрощує тестування та підвищує переносимість рішення між різними моделями Raspberry Pi і навіть між різними апаратними платформами (x86, x86_64, ARM, ARM64).

У складі розроблюваного засобу виділено шість ієрархічних рівнів:

1. Апаратний, який представлений одноплатним комп’ютером Raspberry Pi (моделі 4B або 5), периферійними пристроями зберігання (microSD-картка, зовнішній SSD за інтерфейсом USB 3.0 або NVMe через PCIe), мережевими інтерфейсами Gigabit Ethernet і Wi-Fi, підсистемою живлення та активного

охолодження. Апаратний рівень забезпечує фізичне виконання коду, зберігання даних і мережеву комунікацію.

2. Операційна система, яка функціонує на базі дистрибутиву Raspberry Pi OS (64-bit), похідного від Debian GNU/Linux. Цей рівень надає драйвери апаратного забезпечення, файлову систему, мережевий стек, систему ініціалізації (systemd), механізми керування процесами та користувачами. Саме на цьому рівні здійснюється базове налаштування безпеки операційної системи (SSH, firewall, оновлення пакетів).

3. Платформа контейнеризації, представлена системами Docker Engine та Docker Compose. Цей рівень забезпечує ізоляцію користувацьких застосунків у легковагих контейнерах, керування їх мережами та томами, а також декларативний опис багатоконтейнерних розгортань через YAML-маніфести. Усі сервіси, що надаються користувачеві, функціонують саме на цьому рівні, що дозволяє уніфікувати їх встановлення, оновлення та видалення.

4. Серверне ядро cloud-середовища, реалізоване мовою Python 3 на базі веб-фреймворку Flask і ASGI-сервера Hypercorn із використанням проміжного шару AsyncioWSGIMiddleware. Ядро включає модульну бібліотеку pymodules, яка об’єднує функції конфігурування, маршрутизації HTTP-запитів, керування SSL-сертифікатами, взаємодії з Docker Engine, а також набір фонових потоків для моніторингу ресурсів контейнерів, автоматичного перенаправлення портів через UPnP, перевірки оновлень застосунків та отримання сповіщень. Цей рівень виконує роль контролера, що інтерпретує запити користувача та транслює їх у команди для платформи контейнеризації.

5. Клієнтський інтерфейс, реалізований як односторінковий веб-застосунок (Single Page Application) на фреймворку Vue 3 з використанням мови TypeScript. Збірку та інтеграцію модулів здійснює інструмент Vite; стилізація виконується засобами Tailwind CSS. Ключовим функціональним елементом клієнтського рівня є менеджер вікон Prism Window Manager, що забезпечує багатовіконний інтерфейс, наближений до традиційного робочого столу.

Клієнтський рівень взаємодіє із серверним ядром через HTTP/HTTPS API і не має прямого доступу до нижчих рівнів.

6. Прикладні сервіси користувача, які являють собою сукупність контейнеризованих застосунків, встановлених користувачем через вбудований App Store. Кожен застосунок описується docker-compose-маніфестом, який зберігається в каталозі app-store. Цей рівень є розширюваним: додавання нового сервісу не потребує модифікації ядра cloud-середовища.

Принцип суворого розшарування між рівнями дозволяє, по-перше, замінити будь-який рівень незалежно від інших (наприклад, перенести рішення на альтернативну апаратну платформу без зміни серверного ядра) і, по-друге, забезпечує чіткий розподіл відповідальності між підсистемами.

Узагальнена схема взаємодії компонентів програмно-технічного засобу наведена на рис. 2.1. Схема відображає як потоки керування (HTTP-запити користувача, команди до Docker Engine, сигнали фонових потоків), так і потоки даних (файли користувача, конфігураційні файли, журнали, відповіді API).

Типовий сценарій обробки користувацького запиту має такий вигляд. Користувач з довільного пристрою, підключеного до локальної мережі або до Інтернету, відкриває у веб-браузері адресу cloud-середовища (локальну IP, публічну IP, домен Dynamic DNS або мультикастове ім’я homedock.local). Браузер встановлює TLS-з’єднання з Hypercorn, який виконує розшифрування трафіку з використанням SSL-сертифікатів, завантажених із файлової системи. Далі запит передається через AsyncioWSGIMiddleware до Flask-застосунку, де відбувається перевірка автентифікації через cookie-сесію, додавання CSP-заголовків та CSRF-nonce, маршрутизація до відповідного обробника, зареєстрованого модулем маршрутизації. Обробник, за потреби, звертається до Docker Engine через UNIX-сокет для виконання дій над контейнерами (запуск, зупинка, читання журналів, отримання статистики). Результат обробки формується у вигляді відповіді та повертається клієнтові, де інтерпретується компонентами Vue 3 і відображається користувачеві.

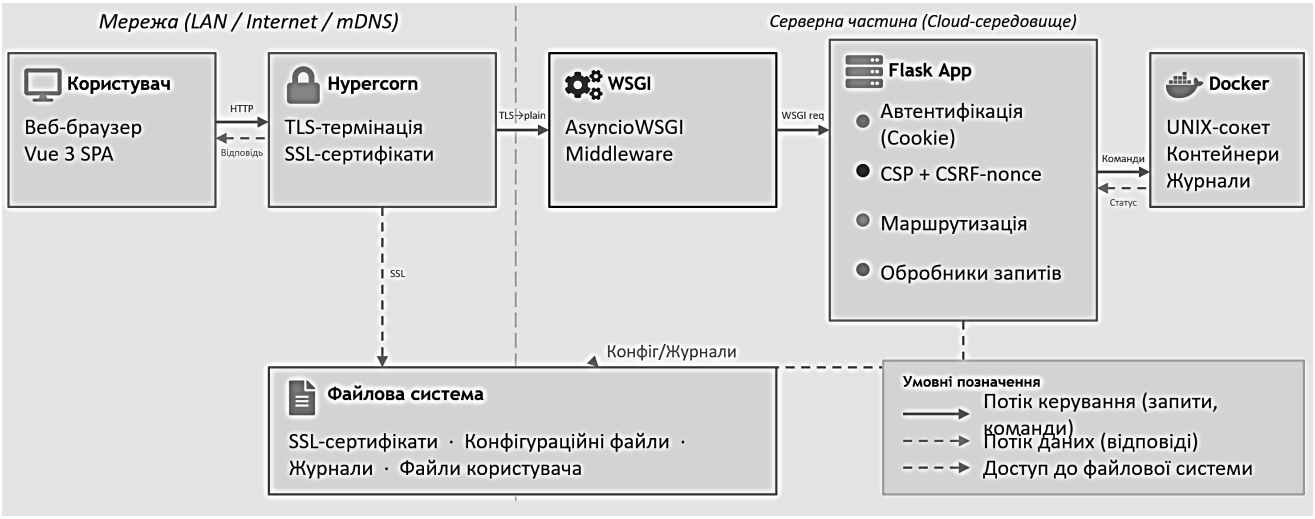


Рисунок 2.1 - Узагальнена схема взаємодії компонентів програмно-технічного засобу

Паралельно з обробкою синхронних запитів у серверному ядрі виконуються фонові потоки. Потік моніторингу ресурсів контейнерів періодично опитує Docker Engine і оновлює внутрішню статистику використання CPU, оперативної пам’яті та мережевого трафіку. Потік автоматичного перенаправлення портів взаємодіє із маршрутизатором локальної мережі за протоколом UPnP для прокидання портів, необхідних для публікації сервісів у Інтернет. Потік перевірки оновлень періодично порівнює локальні версії образів контейнерів з актуальними у віддалених реєстрах. Потік оголошення сервісу через Zeroconf/mDNS публікує ім’я homedock.local у локальній мережі, що дозволяє клієнтам виявляти сервер без використання DNS-серверів.

Взаємодія з апаратним рівнем здійснюється опосередковано через операційну систему: файлові операції переадресовуються драйверам блокових пристроїв, мережеві. Така опосередкована модель взаємодії забезпечує переносимість серверного ядра між різними моделями Raspberry Pi та між різними апаратними платформами загалом.

Прийняті архітектурні рішення обумовлені необхідністю забезпечити баланс між функціональністю, продуктивністю, безпекою та зручністю

використання в умовах обмежених ресурсів одноплатного комп’ютера Raspberry Pi.

Принцип самостійний хостинг обрано як концепцію, оскільки він забезпечує повний контроль користувача над персональними даними, виключає залежність від сторонніх провайдерів, зменшує сукупну вартість володіння порівняно з тривалою підпискою на публічні хмарні сервіси та узгоджується з принципом «privacy by design».

Рисунок 2.2. - Raspberry Pi 5 [2]

Контейнеризація застосунків на базі Docker обрана через низку переваг: ізоляція процесів та залежностей, що виключає конфлікти між застосунками; декларативний опис розгортання у вигляді docker-compose-маніфестів; низькі накладні витрати порівняно з повноцінною віртуалізацією, що є критично важливим для пристроїв з обмеженими ресурсами; існування готових мультиархітектурних образів для ARM/ARM64. Крім того, Docker надає єдиний інтерфейс керування для застосунків, реалізованих різними технологіями, що спрощує побудову каталогу App Store.

Веб-інтерфейс як основний канал взаємодії з користувачем обрано з міркувань кросплатформенності: клієнт не потребує встановлення, працює у будь-якому сучасному браузері на ПК, планшеті або смартфоні, підтримує віддалений доступ без додаткової конфігурації VPN і легко інтегрується з механізмами HTTPS та CSP для забезпечення безпеки.

Python у поєднанні з Flask і Hypercorn обрано для реалізації серверного ядра у зв'язку з широкою екосистемою бібліотек (у тому числі офіційним Docker SDK), високою швидкістю розробки, хорошою підтримкою на платформі ARM та можливістю гнучкої інтеграції з асинхронними ASGI-серверами. Flask забезпечує мінімальний базовий фреймворк, що легко розширюється, а Hypercorn додає асинхронну модель обробки запитів, необхідну для одночасного обслуговування великої кількості клієнтів та фонових задач.

Vue 3 з TypeScript і Vite обрано як сучасний стек розробки клієнтського інтерфейсу, що поєднує реактивну модель відображення даних, статичну типізацію для зменшення кількості помилок часу виконання та швидкий режим розробки із гарячою заміною модулів (Hot Module Replacement). Компонентний підхід Vue 3 добре узгоджується з багатовіконною моделлю Prism Window Manager.

Платформу Raspberry Pi обрано як апаратну основу з огляду на поєднання низької вартості, малих розмірів, помірного енергоспоживання, достатньої обчислювальної потужності сучасних моделей (Pi 4B, Pi 5) для розгортання декількох одночасно працюючих контейнерних сервісів, широкої підтримки з боку спільноти розробників, а також доступності GPIO-інтерфейсів для потенційної інтеграції з пристроями Internet of Things.

Сукупність наведених рішень забезпечує, з одного боку, сучасний користувацький досвід, притаманний комерційним хмарним платформам, а з іншого боку повний контроль користувача над власними даними та обчислювальною інфраструктурою. Деталізоване описання кожного з перелічених компонентів наведено у наступних підрозділах.

2.2 Апаратна складова програмно-технічного засобу

Апаратна складова розроблюваного програмно-технічного засобу є фізичною основою, на якій функціонують усі програмні рівні cloud-середовища. Від характеристик апаратного забезпечення безпосередньо залежить продуктивність системи, її енергоспоживання, надійність зберігання даних та можливість одночасного виконання декількох контейнеризованих сервісів. У цьому підрозділі детально розглянуто обчислювальну платформу Raspberry Pi, підсистему зберігання даних, мережеві інтерфейси, підсистему живлення й охолодження, а також супутню мережеву інфраструктуру.

Для реалізації розроблюваного програмно-технічного засобу у роботі застосовано Raspberry Pi 5, потужності якої достатні для задачі обчислювальні характеристики та повну підтримку 64-бітних операційних систем і Docker.

Raspberry Pi 5 побудована на SoC Broadcom BCM2712 з чотириядерним процесором ARM Cortex-A76 архітектури ARMv8.2-A з тактовою частотою 2,4 ГГц, що забезпечує приблизно у 2–3 рази вищу обчислювальну продуктивність у порівнянні з Pi 4B. Обсяг оперативної пам’яті LPDDR4X досягає 16 ГБ у старшій ревізії. Додатково плата містить роз’єм FPC для підключення PCI Express 2.0 x1, що дозволяє використовувати твердотільні накопичувачі формату M.2 NVMe через відповідні плати розширення. Графічне ядро VideoCore VII забезпечує апаратне декодування відео форматів HEVC.

З огляду на задачу розгортання персонального cloud-середовища, мінімально достатньою конфігурацією слід вважати Raspberry Pi 5 із 8 ГБ оперативної пам’яті, що забезпечує комфортну одночасну роботу щонайменше 8–12 контейнеризованих сервісів середньої ресурсоємності.

Підсистема зберігання даних є критично важливим компонентом cloud-середовища, оскільки від її швидкодії та надійності безпосередньо залежить продуктивність контейнеризованих сервісів, а також збереженість

користувацьких даних. Raspberry Pi підтримує кілька типів носіїв інформації, кожний з яких має власні переваги й обмеження.

Засобом первинного завантаження системи є картка пам'яті microSD, що підключається через вбудований слот. Незважаючи на зручність такого підходу, microSD-картки мають низку суттєвих обмежень для серверного використання: відносно невисока швидкість довільного читання й запису, обмежений ресурс записів, схильність до деградації у разі частих операцій запису. Ці фактори роблять microSD непридатною як основне сховище для cloud-середовища, у якому виконується безперервний запис журналів, бази даних, файлів користувачів та системи резервного копіювання.

Для серверного використання розділено функції завантаження системи та зберігання користувацьких даних. Raspberry Pi 5 додатково дозволяє підключати M.2 NVMe накопичувачі напряму через PCI Express шину з використанням офіційної плати M.2 HAT+ або подібних рішень від сторонніх виробників, що забезпечує максимальну швидкодію підсистеми зберігання.

Для типового домашнього cloud-середовища достатнім є накопичувач обсягом від 256 ГБ до 1 ТБ. При виборі слід надавати перевагу моделям з підтримкою команди TRIM, високим показником середнього часу напрацювання на відмову та достатнім значенням загального обсягу даних, що можуть бути записані протягом строку служби. Додатково, для зберігання великих обсягів мультимедійних даних, можуть використовуватися магнітні жорсткі диски, підключені через USB 3.0 з самостійним живленням.

Мережеві інтерфейси Raspberry Pi забезпечують взаємодію пристрою з локальною мережею та з Інтернетом і, відповідно, з клієнтськими пристроями, що звертаються до cloud-середовища. Обидві розглянуті моделі оснащені двома типами мережевих інтерфейсів: провідним Gigabit Ethernet та бездротовим модулем, що підтримує стандарти IEEE 802.11ac та Bluetooth 5.0.

У серверному сценарії використання перевагу слід віддавати провідному Ethernet-з'єднанню. Воно забезпечує вищу стабільну пропускну здатність (до 1

Гбіт/с теоретично, близько 940 Мбіт/с практично), нижчі затримки, відсутність впливу інтерференції радіосигналів та вищу енергоефективність у порівнянні з Wi-Fi. Для розгортання домашнього cloud-середовища це означає передбачувану продуктивність при передачі файлів великого обсягу, потокової передачі мультимедіа та одночасному обслуговуванні декількох клієнтів.

Бездротовий інтерфейс Wi-Fi 802.11ac (підтримка діапазонів 2,4 ГГц і 5 ГГц) доцільно розглядати лише як запасний або тимчасовий варіант підключення – наприклад, у разі неможливості прокласти кабель до місця розміщення пристрою. У такому разі слід використовувати виключно діапазон 5 ГГц, забезпечувати близьке розташування до точки доступу та розуміти, що фактична пропускна здатність буде суттєво нижчою за теоретичну (як правило, 100–300 Мбіт/с).

Стабільна робота Raspberry Pi в режимі цілодобового функціонування сервера висуває підвищені вимоги як до системи живлення, так і до відведення тепла. Недотримання цих вимог призводить до зниження тактової частоти процесора, непередбачуваних перезавантажень, пошкодження файлової системи та, у граничних випадках, до виходу компонентів з ладу.

Raspberry Pi 5 потребує джерела живлення 5 В/5 А (27 Вт) з підтримкою технології USB Power Delivery. Застосування блоків живлення з недостатнім струмом або неякісних кабелів є поширеною причиною нестабільної роботи, особливо у разі підключення периферійних пристроїв через USB.

Тепловий режим Raspberry Pi у разі тривалого навантаження вимагає використання активного або пасивного охолодження. Для Raspberry Pi 4B типовим рішенням є установка радіатора на SoC та корпус із вбудованим вентилятором; для Raspberry Pi 5 виробник пропонує офіційне активне охолодження, що встановлюється безпосередньо на плату. У серверному застосуванні, де пристрій функціонує в закритому шасі цілодобово, використання активного охолодження є обов’язковим.

Крім самого пристрою Raspberry Pi, повноцінне функціонування cloud-середовища потребувало відповідним чином налаштованої мережевої інфраструктури локальної мережі користувача. До ключових елементів цієї інфраструктури належать маршрутизатор з можливістю автоматичного прокидання портів за протоколом UPnP або ручного налаштування перенаправлень портів, можливість закріплення сталої IP-адреси за пристроєм Raspberry Pi через DHCP reservation або шляхом ручного призначення статичної IP-адреси, а також для забезпечення доступу з Інтернету сервіс Dynamic DNS у разі використання динамічної публічної IP-адреси, наданої провайдером.

Протокол Universal Plug and Play (UPnP) використовується серверним ядром cloud-середовища для автоматичного оголошення маршрутизатору необхідності відкриття визначених TCP-портів у напрямку з Інтернету в локальну мережу. Такий підхід спрощує первинне налаштування з боку користувача, однак у мережах з підвищеними вимогами до безпеки протокол UPnP доцільно вимкнути на рівні маршрутизатора та налаштовувати перенаправлення портів вручну.

Сервіс Dynamic DNS забезпечує відображення змінної публічної IP-адреси, призначеної провайдером Інтернету, на незмінне доменне ім’я (наприклад, у формі user.duckdns.org або user.no-ip.com). Клієнт DDNS періодично повідомляє сервісу поточне значення IP-адреси, у результаті чого користувач може звертатися до власного cloud-середовища за постійним URL, не залежно від зміни IP-адреси провайдером.

2.3 Системне програмне забезпечення

Системне програмне забезпечення розроблюваного програмно-технічного засобу виконує роль посередника між апаратною платформою Raspberry Pi та серверним ядром cloud-середовища. Воно забезпечує керування ресурсами апаратного забезпечення, надає уніфіковані API для роботи з файловою

системою, мережею та процесами, а також створює підґрунтя для ізольованого виконання контейнеризованих застосунків. У складі системного програмного забезпечення виділено операційну систему Raspberry Pi OS, ядро Linux з особливостями архітектури ARM64, платформу контейнеризації Docker Engine та інструмент оркестрації Docker Compose.

Raspberry Pi OS – офіційний дистрибутив операційної системи, що розробляється та підтримується Raspberry Pi Foundation спеціально для пристроїв серії Raspberry Pi. Дистрибутив базується на Debian GNU/Linux, що забезпечує використання зрілої екосистеми пакетів та перевірених засобів адміністрування, і водночас містить оптимізації під апаратні особливості одноплатних комп'ютерів, зокрема специфічні drivers для GPU VideoCore, камери, GPIO-інтерфейсу та інших вбудованих периферійних пристроїв.

У роботі використовується 64-бітна версія Raspberry Pi OS (варіант без графічного середовища – Lite). Відмова від графічного середовища обґрунтована тим, що взаємодія з cloud-середовищем повністю здійснюється через веб-інтерфейс, а локальне графічне середовище лише споживало б додаткові ресурси пам'яті та процесора без функціональної необхідності. 64-бітний режим забезпечує сумісність із сучасними версіями Docker Engine та мультиархітектурними контейнерними образами, а також повноцінне використання всього обсягу оперативної пам'яті.

Ключові компоненти Raspberry Pi OS, які використовуються у розроблюваному програмно-технічному засобі, включають: менеджер пакетів APT для встановлення та оновлення системного програмного забезпечення; систему ініціалізації systemd для керування системними сервісами, у тому числі автозапуском серверного ядра cloud-середовища; сервер OpenSSH для віддаленого адміністрування; стандартний набір утиліт GNU Coreutils. Пакетна база Debian гарантує доступність усіх необхідних для функціонування cloud-середовища системних бібліотек та інтерпретаторів.

Ядро Linux виконує функції низькорівневого керування апаратним забезпеченням і реалізує всі примітиви, на яких ґрунтується технологія контейнеризації. Для платформи Raspberry Pi використовується збірка ядра, адаптована до архітектури ARM64 та доповнена специфічними для Broadcom BCM2711/BCM2712 драйверами. Завантаження ядра здійснюється через двостадійну послідовність: спершу виконується firmware GPU, яка ініціалізує апаратне забезпечення, а потім передає керування власне ядру Linux, завантаженому з розділу /boot.

Ключовими підсистемами ядра Linux, на яких ґрунтується контейнеризація Docker, є namespaces і control groups. Механізм namespaces забезпечує ізоляцію окремих ресурсів (PID-простору, мережевих інтерфейсів, точок монтування файлових систем, UTS-ідентифікаторів, користувачів, IPC) так, що процеси всередині контейнера сприймають оточення як власну незалежну систему. Механізм cgroups забезпечує квотування ресурсів процесорного часу, оперативної пам’яті, пропускної здатності дискового та мережевого вводу-виводу на рівні груп процесів, що дозволяє запобігати деградації системи внаслідок некоректної поведінки окремого контейнера.

Архітектура ARM64, у порівнянні з x86_64, характеризується меншим енергоспоживанням, іншою моделлю пам’яті та відмінним набором інструкцій. Ці особливості впливають на сумісність програмного забезпечення: для запуску контейнерів на Raspberry Pi необхідно використовувати образи, скомпільовані саме під ARM64 (або мультиархітектурні образи, що автоматично обирають відповідний варіант). Більшість популярних відкритих проектів на сьогодні надають офіційні ARM64-образи через реєстри Docker Hub та GitHub Container Registry, що усуває перешкоди для впровадження cloud-середовища на Raspberry Pi.

Для реалізації інтерфейсу керування контейнерами поверх низькорівневих механізмів ядра Linux було застосовано платформу контейнеризації Docker Engine. На відміну від традиційної віртуалізації, контейнери не емулюють

апаратне забезпечення та не потребують власного ядра – вони розділяють ядро з хост-системою, що забезпечує істотно менші накладні витрати, швидший запуск та вищу щільність розміщення застосунків на одному фізичному вузлі. Ці властивості є визначальними для застосування у рамках обмежених ресурсів Raspberry Pi.

Архітектура Docker Engine включає кілька взаємопов'язаних компонентів: демон dockerd, що виконує запити керування та взаємодіє з ядром через системні виклики; низькорівневий час виконання containerd, що здійснює безпосереднє керування життєвим циклом контейнерів; runc – виконавче середовище, сумісне зі специфікацією Open Container Initiative (OCI). Взаємодія клієнтських застосунків з Docker Engine здійснюється через REST API, що зазвичай експонується на локальному UNIX-сокеті /var/run/docker.sock. Саме цей інтерфейс використовує серверне ядро розроблюваного cloud-середовища для виконання операцій над контейнерами.

Ключовими концепціями, з якими оперує Docker, є образ – незмінний шар, що містить файлову систему та метадані для створення контейнера; контейнер – екземпляр, запущений на основі образу; том – механізм постійного зберігання даних, незалежний від життєвого циклу контейнера; мережа – віртуальна мережа, що об'єднує групу контейнерів. Образи зберігаються у реєстрах, основним з яких є публічний Docker Hub. Сучасна практика публікації мультиархітектурних образів за допомогою списків маніфестів дозволяє одному тегу посилатися на набір варіантів образу для різних архітектур, завдяки чому Docker на Raspberry Pi автоматично завантажує саме ARM64-варіант без додаткових дій з боку користувача. Для розгортання самого cloud-середовища було використано Docker Compose.

2.4 Серверний компонент програмно-технічного засобу

Серверний компонент розроблюваного програмно-технічного засобу є центральною ланкою, що об’єднує клієнтський інтерфейс, платформу контейнеризації та операційну систему в єдину функціонально завершену систему. Саме він реалізує бізнес-логіку cloud-середовища: прийом і обробку HTTP-запитів від веб-інтерфейсу, взаємодію з Docker Engine через REST API, керування конфігурацією, ведення журналів, моніторинг ресурсів, а також координацію фонових задач. У цьому підрозділі детально розглянуто мову реалізації, веб-фреймворк, сервер застосунків, проміжні шари, модульну структуру коду та підсистему фонових потоків.

Як мова реалізації серверного ядра використано Python 3. Цей вибір зумовлений низкою об’єктивних переваг мови для задач системного адміністрування та розробки веб-застосунків: висока виразність синтаксису, що скорочує обсяг коду; наявність розвинутої стандартної бібліотеки; широка екосистема сторонніх бібліотек, доступних через індекс PyPI; офіційна бібліотека docker SDK for Python для взаємодії з Docker Engine; крос-платформена підтримка, зокрема повна підтримка архітектури ARM64.

Версія Python, що використовується у розроблюваному засобі, 3.11 або новіша, що забезпечує доступ до сучасних можливостей мови (структурне зіставлення за шаблоном, покращена типізація, пришвидшений інтерпретатор) та до найновіших версій ключових бібліотек. Залежності проекту зафіксовано у файлі requirements.txt і встановлюються через стандартний менеджер пакетів pip, що гарантує відтворюваність середовища на різних пристроях.

До ключових зовнішніх бібліотек Python, використаних у серверному компоненті, належать: Flask – веб-фреймворк; Flask-Compress – проміжний шар стиснення відповідей; Hypercorn – ASGI-сервер для продакшн-розгортання; Werkzeug – бібліотека WSGI-утиліт, зокрема проміжний шар ProxyFix; Jinja2 – шаблонізатор HTML; бібліотеки для інтеграції з апаратною та мережевою

інфраструктурою (обробка SSL-сертифікатів, відкриття портів через UPnP, оголошення сервісів через Zeroconf/mDNS).

У серверному ядрі cloud-середовища Flask виконує такі ключові функції: маршрутизація HTTP-запитів до відповідних обробників; інтеграція з механізмом сесій на основі cookie з властивостями HttpOnly, SameSite=Strict та Secure; обслуговування статичних ресурсів клієнтського SPA-застосунку; надання шаблонів для службових сторінок (сторінки помилок, сторінка входу до оновлення); підтримка Application Context та Request Context для зручної передачі даних між шарами обробки запиту.

Модульність Flask-застосунку забезпечується шляхом реєстрації маршрутів через окремий модуль RouteAllModules, що входить до складу бібліотеки pymodules. Такий підхід дозволяє додавати нові групи ендпойнтів без модифікації головного файлу homedock.py.

Стандартний механізм запуску Flask-застосунків – вбудований розробницький сервер – не призначений для продакшн-використання внаслідок відсутності асинхронної обробки запитів, обмеженої продуктивності та відсутності підтримки HTTP/2. У розроблюваному засобі для обслуговування HTTP-запитів у виробничому режимі використано Hypercorn – сучасний сервер застосунків, що підтримує специфікації ASGI (Asynchronous Server Gateway Interface) та WSGI, а також протоколи HTTP/1.1, HTTP/2 і WebSocket.

Оскільки Flask сам по собі є WSGI-застосунком (синхронним), для його інтеграції з асинхронним Hypercorn використовується проміжний шар AsyncioWSGIMiddleware, що є частиною бібліотеки Hypercorn. Цей шар обгортає WSGI-застосунок у ASGI-сумісний інтерфейс, забезпечуючи виконання синхронного коду Flask у пулі потоків asyncio без блокування основного циклу подій. Такий гібридний підхід дозволяє, з одного боку, зберегти звичну модель розробки Flask-застосунку, а з іншого – отримати переваги асинхронної обробки при обслуговуванні великої кількості одночасних з’єднань.

Конфігурація Hypercorn у розроблюваному засобі включає низку параметрів: прив'язку до всіх мережевих інтерфейсів на налаштованому користувачем порту; підключення файлів SSL-сертифікатів у разі активації HTTPS; обмеження максимального розміру тіла запиту (1 ГБ) для завантаження великих файлів; приховання заголовка Server для зменшення витоку інформації про серверне програмне забезпечення. Крім того, у разі активації HTTPS на порту 443 паралельно запускається окремий сервер перенаправлення з порту 80, що автоматично переадресовує клієнтів на захищену версію.

Для оптимізації мережевого трафіку та адаптації серверного ядра до різних сценаріїв розгортання використовується набір проміжних шарів (middleware). Flask-Compress автоматично стискає відповіді алгоритмами gzip або brotli, що суттєво зменшує обсяг трафіку для текстових форматів (HTML, JSON, JavaScript, CSS) і покращує час завантаження інтерфейсу для віддалених клієнтів.

Проміжний шар ProxyFix з бібліотеки Werkzeug активується у разі розгортання cloud-середовища за зворотним проксі-сервером. Шар коректно обробляє HTTP-заголовки X-Forwarded-For, X-Forwarded-Proto і X-Forwarded-Host, що дозволяє Flask-застосунку отримувати справжню IP-адресу клієнта та правильно формувати абсолютні URL у відповідях навіть у разі термінації TLS на рівні проксі.

Додатково, для запобігання зловмисному завантаженню файлів надмірного розміру, що могло б призвести до вичерпання оперативної пам'яті або дискового простору Raspberry Pi, застосовується ContentSizeLimitMiddleware, який перевіряє значення заголовка Content-Length до початку обробки тіла запиту та відхиляє запити, що перевищують заданий ліміт.

Уся функціональність серверного ядра, що виходить за рамки стандартних механізмів Flask, винесена у спеціалізовану внутрішню бібліотеку pymodules. Така організація забезпечує локалізацію пов'язаних функцій у межах окремих модулів, спрощує повторне використання коду та полегшує внесення змін.

Головний файл homedock.py виконує лише роль точки входу, імпортуючи потрібні модулі та викликаючи їх функції ініціалізації в правильному порядку.

Така модульна організація дозволяє, по-перше, розробнику швидко орієнтуватися у структурі проекту, а по-друге – ізольовано тестувати окремі модулі без запуску повного серверного ядра.

Значна частина функціональності cloud-середовища має виконуватися не у відповідь на HTTP-запит, а періодично або безперервно у фоновому режимі. Для таких задач у серверному ядрі реалізована підсистема фонових потоків на базі стандартного модуля threading мови Python. Кожен фоновий потік інкапсульований у власний модуль pymodules і запускається на етапі старту основного процесу після ініціалізації Flask-застосунку.

Потік моніторингу ресурсів контейнерів (hd_ThreadContainerResourceUsage) періодично опитує Docker Engine та накопичує агреговану статистику використання процесорного часу, оперативної пам’яті та мережевого вводу-виводу для кожного контейнера. Ця інформація кешується у пам’яті серверного ядра і подається клієнтові у відповідь на запити веб-інтерфейсу без необхідності повторного звернення до Docker Engine, що зменшує затримки та знижує навантаження на Docker API.

Потік автоматичного перенаправлення портів (hd_ThreadAutoPortRouting) через визначені проміжки часу встановлює або оновлює правила UPnP на маршрутизаторі локальної мережі, забезпечуючи публікацію cloud-середовища в Інтернет без необхідності ручного налаштування з боку користувача. Потік перевірки оновлень застосунків (hd_ThreadAppUpdatesChecker) періодично звертається до реєстрів Docker для порівняння локальних та віддалених версій образів і сигналізує користувачеві про доступні оновлення. Потік отримання сповіщень (hd_ThreadNotificationsFetcher) завантажує оголошення від розробника cloud-середовища, що відображаються у веб-інтерфейсі. Потік оголошення сервісу через Zeroconf (hd_ThreadZeroConf) публікує ім’я homedock.local у локальній мережі через mDNS.

Через особливості інтерпретатора CPython, а саме наявність глобального блокування інтерпретатора (Global Interpreter Lock, GIL), багатопотокова модель Python є ефективною для задач, пов’язаних переважно з очікуванням введення-виведення (I/O-bound), – саме такими є описані фонові задачі (мережеві запити, системні виклики). Це дозволяє досягти значної паралельності обробки без переходу на складніші моделі мультипроцесингу або асинхронної обробки в рамках окремого циклу подій.

2.5 Клієнтський компонент програмно-технічного засобу

Клієнтський компонент розроблюваного програмно-технічного засобу реалізує користувацький інтерфейс, через який здійснюється керування усіма функціями cloud-середовища. Він представлений односторінковим веб-застосунком, побудованим на базі сучасного стеку фронтенд-розробки, що включає фреймворк Vue 3, мову TypeScript, інструмент збірки Vite, фреймворк стилізації Tailwind CSS та фірмовий менеджер вікон Prism Window Manager. У цьому підрозділі послідовно розглянуто кожен з компонентів клієнтського стеку та його роль у забезпеченні функціональності cloud-середовища.

Для побудови користувацьких інтерфейсів було використано Vue – JavaScript-фреймворк, що поєднує декларативну шаблонну модель з реактивною системою відстеження залежностей. В основі Vue лежить модель реактивного програмування: компонент декларує свій стан та шаблон відображення, а фреймворк автоматично відстежує залежності шаблону від стану і виконує точкові оновлення реального DOM при зміні відповідних полів. Це звільняє розробника від ручного синхронізування даних та представлення, зменшує кількість потенційних помилок і спрощує підтримку складних інтерфейсів.

Composition API, який є основним стилем написання компонентів у Vue 3, пропонує групування функціональності за логічним призначенням (реактивний стан, обчислювані значення, побічні ефекти, звернення до зовнішніх сервісів) у

вигляді composable-функцій. Це дозволяє повторно використовувати складні патерни взаємодії (наприклад, логіку роботи з вікнами Prism Window Manager, керування авторизаційною сесією, обмін даними з серверним API) у вигляді окремих модулів, що імпортуються в компоненти за потреби. В ідіоматичному Vue 3 компонент зазвичай оформляється як однофайловий (.vue) з блоком <script setup lang="ts">, що поєднує переваги Composition API з лаконічним синтаксисом та автоматичним експортом символів для шаблону.

У розроблюваному пристрої було застосовано Vite, який виконує низку ключових задач: запускає локальний dev-сервер на порту 5173 з підтримкою гарячої заміни модулів; транспілює TypeScript та компілює шаблони Vue; обробляє CSS за допомогою PostCSS та Tailwind CSS; створює для продакшну оптимізовану збірку у каталозі homedock-ui/vue3/dist/ з маніфестом manifest.json, що містить відображення логічних імен файлів у їх хешовані імена.

Інтеграція між Vite dev-сервером та Flask-бекендом у режимі розробки реалізована через модуль vite-fusion, який зчитує manifest.json або підключається до dev-сервера Vite залежно від режиму запуску cloud-середовища. Такий підхід дозволяє, з одного боку, забезпечити звичний досвід розробки з гарячою заміною модулів, а з іншого – зберегти єдину точку розгортання у продакшні.

Стилізація клієнтського інтерфейсу cloud-середовища виконується за допомогою фреймворку Tailwind CSS, що реалізує парадигму utility-first. На відміну від класичних CSS-фреймворків (Bootstrap, Foundation), Tailwind не надає готових компонентів із зафіксованими стилями, а натомість пропонує набір атомарних класів-утиліт, які безпосередньо відповідають одній CSS-властивості (наприклад, flex, mt-4, text-lg, bg-blue-500). Комбінування таких класів безпосередньо у шаблонах дозволяє швидко створювати унікальний візуальний дизайн без необхідності писати користувацький CSS, а також уникати проблеми невикористаних стилів та каскадних конфліктів.

Tailwind інтегрується у процес збірки через PostCSS – інструмент трансформації CSS, що використовує плагіни для перетворення вихідного коду.

Налаштування Tailwind CSS задається у файлі tailwind.config.js та включає перелік файлів, що скануються для виявлення використаних утилітарних класів, налаштування теми (кольори, шрифти, розміри), а також активні плагіни. Завдяки JIT-компіляції результуючий CSS-файл містить лише ті класи, що реально використовуються у проекті, що забезпечує мінімальний розмір бандлу.

Інтеграція клієнтського SPA-застосунку з Flask-бекендом реалізована через бібліотеку vite-fusion. Ця бібліотека підтримує два режими роботи, що відповідають двом режимам функціонування cloud-середовища загалом. У режимі розробки vite-fusion звертається до dev-сервера Vite за адресою http://localhost:5173, що забезпечує підтримку гарячої заміни модулів – зміни у коді клієнтської частини автоматично відображаються у відкритому браузері без повного перезавантаження сторінки. У продакшн-режимі vite-fusion зчитує маніфест homedock-ui/vue3/dist/.vite/manifest.json та генерує коректні HTML-теги <script> і <link> для підключення оптимізованих ресурсів з хешованими іменами.

Додатково vite-fusion інтегрується з підсистемою безпеки cloud-середовища: через параметр nonce_provider бібліотеці передається функція, що повертає поточне значення CSP-nonce з контексту запиту Flask. Це значення автоматично додається до всіх згенерованих <script>-тегів, що забезпечує їхню відповідність суворій політиці Content Security Policy і одночасно дозволяє браузеру виконувати легітимний JavaScript-код клієнтського застосунку.

Маршрутизація у межах SPA реалізована на боці клієнта через Vue Router. Серверний Flask-застосунок при зверненні до всіх маршрутів, що не відповідають API-ендпойнтам, повертає кореневу HTML-сторінку з підключеним SPA-бандлом; подальша маршрутизація здійснюється вже у браузері. Обмін даними між клієнтом і сервером виконується виключно через JSON-повідомлення за HTTPS.

Важливою особливістю клієнтського інтерфейсу розроблюваного cloud-середовища є Prism Window Manager – вбудований менеджер вікон, що реалізує

метафору класичного настільного середовища операційної системи у веб-браузері. На відміну від типових веб-застосунків, що використовують лінійну модель сторінок, Prism Window Manager дозволяє користувачеві одночасно відкривати декілька незалежних вікон з різною функціональністю (керування контейнерами, перегляд журналів, файловий менеджер, налаштування), розташовувати їх довільним чином, змінювати розміри, згортати, розгортати та перемикатися між ними.

Менеджер вікон підтримує декілька візуальних тем (White, Aero+, Noir), що надають користувачеві можливість адаптувати зовнішній вигляд інтерфейсу до власних уподобань. Усі теми зберігають єдину функціональну модель взаємодії, змінюючи лише колірну палітру, прозорість та анімації. Реалізація менеджера вікон ґрунтується на Composition API Vue 3 та використовує власну систему керування z-index, фокусом та позиціями вікон.

Такий підхід до організації інтерфейсу забезпечує підвищену продуктивність роботи адміністратора cloud-середовища – можливість паралельної роботи з декількома задачами є суттєвою перевагою перед класичними single-page-панелями керування. Водночас на мобільних пристроях інтерфейс автоматично адаптується до вертикального екрану, переходячи до однопотокового режиму відображення, що зберігає зручність використання на будь-якому пристрої з сучасним веб-браузером.

2.4 Висновки до другого розділу

У другому розділі кваліфікаційної роботи виконано проєктування програмно-технічного засобу реалізації персонального cloud-середовища на базі платформи Raspberry Pi. У результаті проведеного аналізу визначено концептуальну модель системи, сформовано її багаторівневу архітектуру та обґрунтовано вибір апаратних і програмних компонентів, необхідних для побудови сучасного cloud-рішення.

У ході проєктування встановлено, що запропонована архітектура базується на принципі суворого розшарування, відповідно до якого система поділена на апаратний рівень, операційну систему, платформу контейнеризації, серверне ядро, клієнтський інтерфейс та прикладні сервіси користувача. Такий підхід забезпечує модульність, масштабованість, переносимість та спрощує супровід системи, дозволяючи незалежно модифікувати або замінювати окремі компоненти без впливу на інші рівні.

Обґрунтовано доцільність використання Raspberry Pi 5 як апаратної основи cloud-середовища завдяки поєднанню достатньої обчислювальної потужності, підтримки ARM64, низького енергоспоживання та широких можливостей інтеграції з периферійними й IoT-пристроями. Визначено оптимальну конфігурацію апаратної підсистеми, що включає використання SSD/NVMe-накопичувачів, провідного Gigabit Ethernet-з'єднання, стабільного джерела живлення та активного охолодження для забезпечення надійного цілодобового функціонування.

У межах дослідження системного програмного забезпечення встановлено, що використання Raspberry Pi OS у 64-бітному виконанні разом із Docker Engine та Docker Compose забезпечує ефективну реалізацію контейнеризованої інфраструктури з мінімальними накладними витратами. Використання механізмів Linux namespaces і cgroups дозволяє забезпечити ізоляцію сервісів, контроль використання ресурсів та стабільність роботи cloud-середовища в умовах обмежених апаратних ресурсів.

Спроєктовано серверний компонент системи на базі Python 3, Flask та Hypercorn, який реалізує бізнес-логіку cloud-середовища, взаємодію з Docker Engine, обробку HTTP/HTTPS-запитів, моніторинг ресурсів, автоматизацію мережевих налаштувань та виконання фонових задач. Використання модульної архітектури бібліотеки pymodules забезпечує гнучкість подальшого розширення функціональності та спрощує супровід програмного коду.

Також розроблено концепцію клієнтського компонента у вигляді односторінкового веб-застосунку на базі Vue 3, TypeScript, Vite та Tailwind CSS. Запропонований інтерфейс забезпечує кросплатформенність, адаптивність та зручність адміністрування cloud-середовища через браузер без необхідності використання командного рядка. Особливістю реалізації є використання Prism Window Manager, що реалізує багатовіконну модель взаємодії та підвищує ефективність роботи користувача.

Таким чином, у результаті виконаного проєктування сформовано цілісну архітектуру програмно-технічного засобу реалізації персонального cloud-середовища, яка поєднує сучасні технології контейнеризації, веб-розробки та self-hosting-підходу. Запропоновані рішення забезпечують баланс між продуктивністю, безпекою, функціональністю та зручністю використання, що створює підґрунтя для подальшої практичної реалізації та експериментального дослідження системи у наступних розділах роботи.

3 ПРОГРАМНО-АПАРАТНА РЕАЛІЗАЦІЯ CLOUD-СЕРЕДОВИЩА НА БАЗІ RASPBERRY PI

3.1 Підготовка апаратної платформи Raspberry Pi

Процес реалізації програмно-технічного засобу починається з підготовки апаратної платформи Raspberry Pi, яка виступає фізичною основою cloud-середовища. Підготовка охоплює комплектацію та фізичне складання пристрою, запис образу операційної системи на носій, первинне завантаження з базовим налаштуванням та оновлення системних пакетів. Коректне виконання цього етапу є передумовою стабільного функціонування всіх наступних програмних компонентів.

Для розгортання cloud-середовища було підготовлено апаратний комплект на базі Raspberry Pi 4 Model B з 8 ГБ оперативної пам'яті. До складу комплекту входять: безпосередньо плата Raspberry Pi 4B; офіційний блок живлення USB Type-C 5 В / 3 А; алюмінієвий корпус із вбудованим вентилятором для активного охолодження; microSD-картка ємністю 32 ГБ класу A2 для завантаження операційної системи; зовнішній SSD-накопичувач ємністю 256 ГБ, підключений через USB 3.0, що використовується як основне сховище даних cloud-середовища; кабель Ethernet категорії 5e для підключення до локальної мережі.

Складання пристрою виконується у такій послідовності: на SoC та мікросхему оперативної пам'яті встановлюються теплопровідні прокладки, що входять до комплекту корпусу; плата розміщується в корпусі та фіксується; вентилятор підключається до GPIO-контактів живлення (5 В та GND); SSD-накопичувач підключається до порту USB 3.0 (верхній ряд портів, синього кольору). Після складання пристрій підключається до маршрутизатора через Ethernet-кабель, але живлення на цьому етапі не подається – спочатку необхідно підготувати носій із операційною системою.

Запис образу операційної системи на microSD-картку виконується за допомогою офіційного інструменту Raspberry Pi Imager, доступного для

Windows, macOS та Linux. Цей інструмент автоматизує процес завантаження актуального образу, його верифікації та запису на носій.

У Raspberry Pi Imager було обрано образ Raspberry Pi OS Lite (64-bit) – варіант без графічного середовища робочого столу, оптимальний для серверного використання. Перед записом у розширених налаштуваннях Imager (доступних через комбінацію Ctrl+Shift+X або через меню налаштувань) задаються початкові параметри: ім'я хоста (hostname) – наприклад, raspberrypi; ім'я користувача та пароль для першого входу; увімкнення SSH-сервера з автентифікацією за паролем; налаштування часового поясу та розкладки клавіатури. Попереднє конфігурування через Imager усуває необхідність підключення монітора та клавіатури до Raspberry Pi для первинного налаштування, що є суттєвою перевагою для безголового (headless) серверного розгортання.

Після запису образу microSD-картка встановлюється у відповідний слот Raspberry Pi, після чого подається живлення. Перше завантаження триває 2–3 хвилини, протягом яких система розширює розділ файлової системи на весь доступний обсяг картки, генерує унікальні SSH-ключі та застосовує попередньо задані налаштування.

Після першого завантаження доступ до Raspberry Pi здійснюється віддалено через протокол SSH з робочої станції, що знаходиться у тій же локальній мережі. Для підключення використовується команда ssh <ім'я_користувача>@<IP-адреса>, де IP-адреса може бути визначена через інтерфейс маршрутизатора або за допомогою утиліти мережевого сканування.

Після успішного підключення виконується перевірка та, за необхідності, коригування регіональних налаштувань через утиліту raspi-config: встановлюється локаль uk_UA.UTF-8 або en_US.UTF-8, коректний часовий пояс (Europe/Kyiv), розкладка клавіатури. Перевіряється стан SSH-сервера командою sudo systemctl status sshd та, за необхідності, забезпечується його автозапуск командою sudo systemctl enable ssh.

Додатково на цьому етапі рекомендовано змінити початковий пароль користувача на складніший, вимкнути автентифікацію SSH за паролем із заміною на ключову автентифікацію (шляхом додавання публічного ключа до файлу ~/.ssh/authorized_keys та встановлення параметра PasswordAuthentication no у файлі /etc/ssh/sshd_config), а також налаштувати статичну IP-адресу або закріплення адреси через DHCP reservation на маршрутизаторі для забезпечення незмінної адреси пристрою у локальній мережі.

Перед встановленням будь-якого додаткового програмного забезпечення необхідно оновити всі системні пакети до актуальних версій. Це забезпечує отримання останніх виправлень безпеки та сумісність із сучасними версіями Docker та Python. Оновлення виконується стандартними командами менеджера пакетів APT: sudo apt update для оновлення індексу пакетів та sudo apt full-upgrade -y для встановлення оновлень.

Після оновлення системних пакетів виконується базове тюнінг-налаштування, спрямоване на оптимізацію Raspberry Pi для серверного використання. Через утиліту raspi-config вимикається функція підтримки робочого столу (якщо не було обрано Lite-версію), налаштовується розподіл пам'яті GPU (для серверного використання достатньо мінімального значення 16 МБ, що вивільняє додатковий обсяг оперативної пам'яті для контейнерів).

У файлі /boot/firmware/config.txt або /boot/config.txt (залежно від версії Raspberry Pi OS) за необхідності вносяться коригування параметрів системи: активація інтерфейсу USB для завантаження з зовнішнього SSD, налаштування частоти вентилятора, активація watchdog-таймера для автоматичного перезавантаження у разі зависання. На зовнішньому SSD створюється файлова система ext4 та точка монтування, що додається до /etc/fstab для автоматичного підключення при завантаженні. Саме на цьому накопичувачі надалі зберігатимуться дані Docker-контейнерів та користувацькі файли cloud-середовища.

3.2 Розгортання системного оточення

Після підготовки апаратної платформи та базового налаштування операційної системи наступним етапом було розгортання системного оточення – встановлення програмних компонентів, на яких безпосередньо функціонуватиме cloud-середовище. До таких компонентів належать Docker Engine, Docker Compose, інтерпретатор Python 3 з необхідними бібліотеками, а також Node.js для збірки клієнтської частини.

Docker Engine для Raspberry Pi встановлюється з офіційного репозиторію Docker за процедурою, рекомендованою документацією проекту. Спочатку встановлюються необхідні залежності для роботи з HTTPS-репозиторіями: пакети ca-certificates, curl та gnupg. Далі додається GPG-ключ офіційного репозиторію Docker та налаштовується джерело пакетів для архітектури arm64.

Встановлення виконується командою sudo apt install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin, що встановлює демон Docker (docker-ce), клієнт командного рядка (docker-ce-cli), низькорівневе середовище виконання контейнерів (containerd.io), плагін збірки образів (docker-buildx-plugin) та плагін Docker Compose (docker-compose-plugin). Після встановлення користувач додається до групи docker командою sudo usermod -aG docker $USER, що дозволяє виконувати команди Docker без використання sudo.

Верифікація встановлення виконується запуском тестового контейнера командою docker run hello-world, що завантажує мінімальний образ з Docker Hub та виводить підтверджувальне повідомлення. Додатково перевіряється підтримка архітектури ARM64 командою docker info, де у секції Architecture має відображатися значення aarch64. Для забезпечення автоматичного запуску Docker при завантаженні системи активується відповідний systemd-сервіс командою sudo systemctl enable docker.

У сучасних версіях Docker Engine плагін Docker Compose встановлюється автоматично разом із основним пакетом (docker-compose-plugin) і доступний

через команду docker compose (без дефіса). Для перевірки коректності встановлення виконується команда docker compose version, що має вивести поточну версію плагіна.

Працездатність Docker Compose перевіряється створенням тестового docker-compose.yml з описом сервісу nginx) та його запуском командою docker compose up -d. Після успішного запуску контейнер має бути доступний на зазначеному порту, а його стан – відображатися у виводі команди docker compose ps. Після завершення тестування сервіс зупиняється та видаляється командою docker compose down.

Серверне ядро cloud-середовища додатково виконує програмну верифікацію наявності Docker та Docker Compose при кожному запуску через модулі hd_FunctionsMain (функції validate_docker_installation та validate_docker_compose_installation), що забезпечує раннє виявлення проблем і формування зрозумілих повідомлень про помилки.

Raspberry Pi OS 64-bit включає попередньо встановлений інтерпретатор Python 3, однак для функціонування серверного ядра cloud-середовища необхідно встановити менеджер пакетів pip та набір залежностей, визначених у файлі requirements.txt проекту. Менеджер pip встановлюється командою sudo apt install python3-pip python3-venv.

Залежності проекту встановлюються командою pip3 install -r requirements.txt, що завантажує та інсталює усі необхідні Python-бібліотеки: Flask, Flask-Compress, Hypercorn, Werkzeug, Jinja2, бібліотеки для роботи з Docker API, SSL-сертифікатами, UPnP, Zeroconf та інші. Деякі бібліотеки містять C-розширення, компіляція яких вимагає наявності системних пакетів розробки (python3-dev, build-essential, libffi-dev, libssl-dev), що встановлюються попередньо через APT.

Для ізоляції залежностей cloud-середовища від системних бібліотек Python може використовуватися віртуальне середовище (venv), створене командою python3 -m venv /opt/homedock/venv. Такий підхід запобігає конфліктам версій

бібліотек і спрощує оновлення або видалення cloud-середовища без впливу на операційну систему.

Клієнтська частина cloud-середовища побудована на фреймворку Vue 3 з використанням інструменту збірки Vite та потребує Node.js для компіляції TypeScript-коду, обробки CSS через PostCSS і Tailwind, а також для створення оптимізованої продакшн-збірки. Встановлення Node.js на Raspberry Pi виконувілося через офіційний репозиторій NodeSource

Встановлення Node.js та npm було необхідним лише для збірки клієнтського застосунку або для роботи у режимі розробки з підтримкою HMR.

3.3 Розгортання HomeDock OS

Після підготовки системного оточення було виконано розгортання cloud-середовища HomeDock OS. Проект підтримує два варіанти розгортання – нативний запуск безпосередньо на хост-системі та контейнеризований запуск через Docker Compose. Кожний варіант має власні переваги і сценарії застосування.

Структура проекту організована наступним чином. У кореневому каталозі розміщено головний файл серверного ядра homedock.py, конфігурацію Docker-контейнеризації (Dockerfile, docker-compose.yml, entrypoint.sh), файли конфігурації фронтенд-збірки (package.json, vite.config.ts, tsconfig.json, tailwind.config.js, postcss.config.js), файл залежностей Python (requirements.txt) та файл з поточною версією (version.txt). Каталог pymodules містить усі модулі серверного ядра. Каталог homedock-ui містить вихідний код клієнтського інтерфейсу на Vue 3 із підкаталогом vue3 для компонентів, стилів та ресурсів. Каталог app-store містить маніфести та docker-compose-шаблони застосунків, доступних для встановлення.

Файл version.txt зберігає рядок поточної версії cloud-середовища, що використовується для відображення у веб-інтерфейсі та для перевірки оновлень.

Файл .gitignore налаштований на ігнорування згенерованих артефактів (каталогів node_modules, dist, __pycache__), файлів конфігурації з конфіденційними даними та журналів.

Нативний запуск є рекомендованим варіантом для Raspberry Pi, оскільки він забезпечує мінімальні накладні витрати і прямий доступ серверного ядра до Docker Engine хост-системи. Підготовка до нативного запуску включає встановлення всіх залежностей Python (pip3 install -r requirements.txt), встановлення залежностей фронтенду (npm install) та збірку клієнтського застосунку (npm run build), що створює оптимізовану збірку у каталозі homedock-ui/vue3/dist/.

Запуск серверного ядра виконувався командою python3 homedock.py. При цьому homedock.py є точкою входу, що виконує послідовність ініціалізації: зміна робочого каталогу, перевірка та генерація конфігурації, ініціалізація директорій, валідація Docker і Docker Compose, створення Flask-застосунку, реєстрація маршрутів, запуск фонових потоків та, нарешті, запуск HTTP/HTTPS-сервера.

Для забезпечення автоматичного запуску cloud-середовища при завантаженні Raspberry Pi створюється systemd-сервіс. Файл сервісу /etc/systemd/system/homedock.service визначає тип сервісу, робочий каталог, команду запуску, користувача, від імені якого виконується процес, та політику перезапуску (Restart=on-failure). Після створення файлу сервіс активується командами sudo systemctl daemon-reload та sudo systemctl enable homedock.

При першому запуску cloud-середовища серверне ядро виконує низку автоматичних ініціалізаційних операцій. Модуль hd_FunctionsConfig виявляє відсутність конфігураційного файлу homedock_server.conf і генерує його з параметрами за замовчуванням: стандартний порт (80 або 443 у разі наявності SSL-сертифікатів), ім'я користувача за замовчуванням, активація Zeroconf та DDNS за необхідності. Модуль hd_FunctionsInitUserFolders створює структуру каталогів для зберігання даних застосунків, журналів, SSL-сертифікатів та тимчасових файлів.

Модуль hd_FunctionsActiveInstance перевіряє, що жоден інший екземпляр серверного ядра не функціонує на тому ж порту, запобігаючи конфліктам прив'язки. Модуль hd_UpdateDeps перевіряє актуальність залежностей та за необхідності оновлює їх.

Після завершення ініціалізації серверне ядро виводить у консоль повний стартовий журнал, що включає: версію cloud-середовища та хеш версії; робочий каталог; мережевий порт; локальну та публічну IP-адреси; стан SSL та режим розробки; тип процесора та операційну систему; облікові дані для першого входу (ім'я користувача та пароль за замовчуванням – passwd); URL-адреси для доступу з локальної мережі, з Інтернету та через DDNS. Після ознайомлення з обліковими даними користувач відкриває веб-інтерфейс у браузері за будь-якою із запропонованих адрес та виконує вхід з використанням початкових облікових даних. Зміна пароля на безпечний є обов'язковою першочерговою дією.

3.4 Реалізація серверної частини

Серверна частина cloud-середовища реалізована у вигляді Flask-застосунку, що запускається головним файлом homedock.py. У цьому підрозділі детально описано послідовність ініціалізації серверного ядра, налаштування middleware, конфігурацію продакшн-сервера, реєстрацію маршрутів, запуск фонових потоків та механізми взаємодії з Docker Engine.

Flask-застосунок ініціалізується у модулі hd_HDOSWebServerInit, де створюється екземпляр класу Flask з налаштуванням каталогів шаблонів та статичних файлів. Після імпорту у homedock.py до застосунку послідовно підключаються компоненти.

Фільтр шаблонів b64encode_filter з модуля hd_AppFilters реєструється через метод add_template_filter і дозволяє кодувати довільні дані у формат Base64 безпосередньо у Jinja2-шаблонах, що використовується для вбудовування зображень та бінарних даних. Middleware Flask-Compress активується викликом

конструктора Compress і забезпечує автоматичне стиснення HTTP-відповідей алгоритмами gzip або brotli для текстових MIME-типів.

Інтеграція з фронтенд-збіркою виконується через бібліотеку vite-fusion, що реєструється функцією register_vite_assets. Їй передаються параметри режиму розробки (dev_mode), URL dev-сервера Vite, шлях до dist-каталогу збірки, шлях до manifest.json та функція-провайдер CSP-nonce. У залежності від режиму vite-fusion або проксіює запити до dev-сервера Vite, або генерує HTML-теги підключення оптимізованих ресурсів з manifest.json.

У продакшн-режимі (run_on_development = false) Flask-застосунок обслуговується асинхронним ASGI-сервером Hypercorn. Конфігурація Hypercorn створюється як об'єкт класу Config з наступними параметрами: bind встановлюється на 0.0.0.0 із зазначеним портом, що забезпечує приймання з'єднань з усіх мережевих інтерфейсів; loglevel встановлюється у DEBUG для детального журналювання; include_server_header вимикається для приховування ідентифікації серверного програмного забезпечення.

У разі активації SSL до конфігурації Hypercorn додаються шляхи до файлів сертифікатів: certfile вказує на fullchain.pem, keyfile – на privkey.pem, ca_certs – на chain.pem. Каталог розташування сертифікатів визначається функцією get_ssl_cert_directory модуля hd_FunctionsNativeSSL. Якщо SSL увімкнено і порт дорівнює 443, додатково запускається HTTP-сервер перенаправлення на порту 80 через модуль hd_HTTPRedirector.

WSGI-застосунок Flask загортається у ASGI-сумісну обгортку через функцію homedock_www_asgi, яка застосовує ProxyFix (у разі активації reverse proxy), AsyncioWSGIMiddleware для асинхронного виконання та ContentSizeLimitMiddleware для обмеження розміру запитів. Запуск серверів виконується через asyncio.run(run_all_servers()), де run_all_servers використовує asyncio.gather для паралельного запуску основного HTTPS-сервера та допоміжного HTTP-перенаправлення, з коректною обробкою сигналів SIGINT та SIGTERM для плавного завершення роботи.

Маршрутизація HTTP-запитів у серверному ядрі реалізована модульно через функцію RouteAllModules з однойменного модуля pymodules. Ця функція приймає екземпляр Flask-застосунку та функцію send_public_key і виконує реєстрацію всіх груп маршрутів (endpoints), що забезпечують API cloud-середовища.

До основних груп маршрутів належать: маршрути автентифікації (вхід, вихід, перевірка сесії); маршрути керування застосунками App Store (перегляд каталогу, встановлення, видалення, оновлення); маршрути керування контейнерами (запуск, зупинка, перезапуск, отримання статистики); маршрути доступу до журналів контейнерів; маршрути керування конфігурацією cloud-середовища; маршрути завантаження та керування файлами; маршрути системного моніторингу (CPU, RAM, диск, мережа); маршрут передачі публічного ключа через функцію send_public_key.

Модульний підхід до реєстрації маршрутів забезпечує, по-перше, чітку організацію коду – кожна група маршрутів може бути реалізована в окремому файлі модуля; по-друге, можливість додавання нових API-ендпойнтів без модифікації головного файлу homedock.py; по-третє, можливість умовної реєстрації маршрутів у залежності від конфігурації (наприклад, Enterprise-функції, що завантажуються модулем hd_EnterpriseLoader).

Після реєстрації маршрутів і безпосередньо перед запуском HTTP-сервера виконується ініціалізація фонових потоків, що забезпечують періодичне виконання сервісних функцій cloud-середовища. Запуск потоків здійснюється викликом відповідних функцій-стартерів.

Функція start_resource_usage_thread запускає потік моніторингу ресурсів контейнерів, що періодично опитує Docker Engine та кешує статистику використання CPU, пам'яті та мережі. Функція start_auto_port_routing_thread запускає потік UPnP-клієнта, що забезпечує автоматичне перенаправлення портів на маршрутизаторі. Функція start_app_updates_checker_thread запускає потік перевірки оновлень, що порівнює хеші локальних образів із віддаленими.

Функція start_notifications_fetcher_thread запускає потік отримання сповіщень від розробника cloud-середовища.

Усі фонові потоки запускаються як daemon-потоки (daemon=True), що забезпечує їх автоматичне завершення при завершенні основного процесу без необхідності явної синхронізації. Кожний потік містить внутрішній цикл із заданим інтервалом між ітераціями та обробку виняткових ситуацій для запобігання аварійному завершенню потоку через помилки окремих операцій.

Взаємодія серверного ядра з Docker Engine здійснюється через REST API, що експонується на локальному UNIX-сокеті /var/run/docker.sock. Серверне ядро використовує цей інтерфейс для виконання повного спектру операцій над контейнерами: створення контейнера на основі зазначеного образу з визначеними параметрами (порти, томи, змінні середовища, мережі); запуск, зупинка та перезапуск контейнерів; видалення контейнерів та асоційованих томів; отримання переліку запущених контейнерів з їх статусом; отримання потокової статистики використання ресурсів; отримання журналів контейнера; завантаження образів з Docker Hub.

Для розгортання багатоконтейнерних застосунків із каталогу App Store серверне ядро генерує docker-compose-маніфест на основі шаблону, записує його у робочу директорію застосунку та ініціює розгортання через виклик Docker Compose. Цей підхід дозволяє використовувати декларативний опис сервісів з усіма перевагами Compose (управління залежностями, іменовані мережі, узгоджений запуск) і водночас зберігає контроль серверного ядра над процесом розгортання.

Монтування Docker-сокету у контейнер cloud-середовища (у разі Docker-in-Docker розгортання) є стандартним підходом, що дозволяє контейнеру керувати контейнерами хост-системи. Слід зазначити, що такий підхід вимагає обережності з точки зору безпеки, оскільки доступ до Docker-сокету фактично еквівалентний root-правам на хост-системі.

Конфігурація Flask-сесій виконується безпосередньо у homedock.py шляхом встановлення набору параметрів об'єкта homedock_www.config. Час дії сесії було встановлено на 24 години (PERMANENT_SESSION_LIFETIME). Секретний ключ для підпису cookie генерується як 32 байти випадкових даних (SECRET_KEY = os.urandom(32)). Параметр SESSION_REFRESH_EACH_REQUEST встановлюється у False, що означає оновлення часу закінчення сесії лише при її явній модифікації, а не при кожному запиті, що зменшує навантаження на механізм сесій. Параметри безпеки cookie (HttpOnly, SameSite=Strict, Secure, кастомне ім'я homedock_session) описані у підрозділі 2.6.4.

Обробка HTTP-помилок реалізована модулем hd_HTMLErrorCodeHandler, який реєструє через функцію setup_error_handlers обробники для стандартних кодів помилок (400, 403, 404, 405, 500 та інших). Кожний обробник генерує стилізовану HTML-сторінку з інформативним повідомленням про помилку, хешем версії cloud-середовища та зворотним посиланням на головну сторінку. Використання кастомних сторінок помилок замість стандартних відповідей Flask покращує користувацький досвід та не розкриває технічну інформацію про серверний стек потенційному зловмиснику.

Клієнтська частина cloud-середовища реалізована як односторінковий веб-застосунок (SPA) на базі Vue 3 та TypeScript. У цьому підрозділі описано процес збірки клієнтського застосунку, механізм інтеграції з серверним ядром, реалізацію маршрутизації та компонентної моделі, функціональність менеджера вікон Prism та адаптивну верстку.

Збірка клієнтського застосунку виконується інструментом Vite, конфігурація якого визначена у файлі vite.config.ts. Процес збірки ініціюється командою npm run build, що запускає Vite у режимі продакшн-збірки. Vite виконує наступну послідовність операцій: транспіляцію TypeScript-коду через компілятор tsc; компіляцію шаблонів однофайлових Vue-компонентів (.vue) через плагін @vitejs/plugin-vue; обробку CSS через PostCSS з плагіном Tailwind

CSS; розділення коду на чанки (code splitting) для оптимізації завантаження; мінімізацію JavaScript та CSS; генерацію хешованих імен файлів для ефективного кешування; створення файлу маніфесту .vite/manifest.json.

Результатом збірки є каталог homedock-ui/vue3/dist/, що містить оптимізовані JavaScript-бандли, CSS-файли, статичні ресурси (іконки, зображення, шрифти) та файл маніфесту. Розмір результуючих бандлів мінімізується завдяки JIT-компіляції Tailwind CSS (включаються лише використані класи), tree-shaking (видалення невикористаного коду) та мінімізації (видалення пробілів, коментарів, скорочення ідентифікаторів).

Після продакшн-збірки клієнтський застосунок інтегрується з Flask-бекендом через бібліотеку vite-fusion. У продакшн-режимі vite-fusion зчитує файл manifest.json, що містить відображення логічних імен файлів вихідного коду (наприклад, src/main.ts) у хешовані імена продакшн-бандлів (наприклад, assets/main-a1b2c3d4.js). На основі цього відображення vite-fusion генерує HTML-теги <script type="module"> та <link rel="stylesheet">, які вставляються у базовий HTML-шаблон Flask.

Кожний згенерований <script>-тег автоматично отримує атрибут nonce з поточним значенням CSP-nonce, отриманим через функцію-провайдер nonce_provider. Це забезпечує сумісність із суворою Content Security Policy, що забороняє виконання скриптів без валідного nonce. Статичні ресурси (зображення, шрифти) обслуговуються Flask як звичайні статичні файли з каталогу dist.

Маршрутизація у межах клієнтського SPA реалізована через Vue Router у режимі history mode. Сервер Flask налаштований таким чином, що будь-який GET-запит, який не відповідає відомому API-ендпойнту або статичному файлу, повертає кореневу HTML-сторінку із завантаженим SPA-бандлом. Подальша маршрутизація виконується на стороні клієнта: Vue Router аналізує URL та відображає відповідний компонент без повного перезавантаження сторінки.

Компонентна модель UI побудована на Composition API Vue 3 із використанням конвенції <script setup lang="ts">. Компоненти організовані ієрархічно: кореневий компонент App.vue містить менеджер вікон Prism, який, у свою чергу, містить компоненти окремих вікон (Dashboard, AppStore, ContainerManager, LogViewer, Settings та інші). Кожний компонент інкапсулює свій стан, обчислювані властивості та методи взаємодії з серверним API. Спільна логіка (наприклад, робота з API, керування сесією, обробка помилок) виділена у composable-функції, що імпортуються за потреби.

Обмін даними між клієнтом та сервером виконується через HTTP-запити (fetch API або бібліотека axios) до API-ендпойнтів серверного ядра. Відповіді повертаються у форматі JSON і типізуються через TypeScript-інтерфейси, що забезпечує перевірку типів на етапі компіляції.

Prism Window Manager є ключовим елементом користувацького інтерфейсу, що реалізує метафору робочого столу операційної системи у веб-браузері. Технічно він представлений Vue-компонентом, що керує колекцією дочірніх компонентів-вікон, забезпечуючи для кожного з них незалежне позиціонування, зміну розмірів, переміщення перетягуванням, згортання, розгортання на повний екран та переключення фокусу.

Керування z-index реалізовано через реактивний масив, що відстежує порядок накладення вікон: при натисканні на вікно воно отримує максимальне значення z-index у колекції, що виносить його на передній план. Позиція та розміри кожного вікна зберігаються у реактивному стані та оновлюються обробниками подій миші (mousedown, mousemove, mouseup) або дотику (touchstart, touchmove, touchend) для мобільних пристроїв.

Візуальні теми Prism (White, Aero+, Noir) реалізовані через набір CSS-змінних, що перевизначаються на рівні кореневого елемента менеджера вікон. Зміна теми не потребує перезавантаження компонентів – достатньо оновити набір CSS-змінних, після чого всі вкладені елементи автоматично адаптують кольори, прозорість, тіні та анімації.

Cloud-середовище має бути доступним з пристроїв із різними розмірами екрану – від великих моніторів до смартфонів. Адаптивна верстка реалізована за допомогою responsive-утиліт Tailwind CSS, що застосовують CSS-стилі умовно залежно від ширини вікна перегляду (viewport) через медіа-запити.

На великих екранах (ширина понад 1024 пікселі) інтерфейс Prism Window Manager працює у повнофункціональному багатовіконному режимі з підтримкою перетягування, зміни розмірів та довільного позиціонування вікон. На планшетних пристроях (768–1024 пікселі) вікна автоматично збільшуються у розмірі та обмежуються у кількості одночасно відображуваних. На мобільних пристроях (менше 768 пікселів) інтерфейс переходить у однопотоковий режим, де кожне вікно займає повну ширину екрану, а навігація між функціональними розділами здійснюється через бокове меню або панель задач.

Тестування адаптивної верстки виконується за допомогою інструментів розробника у браузері (Chrome DevTools, Firefox Responsive Design Mode), що дозволяють емулювати різні розміри екранів та пристрої, а також безпосередньо на фізичних мобільних пристроях.

3.5 Налаштування мережевої доступності

Для повноцінного функціонування cloud-середовища необхідно забезпечити його доступність як у локальній мережі, так і з Інтернету. У цьому підрозділі описано налаштування Dynamic DNS, оголошення сервісу через Zeroconf, автоматичне перенаправлення портів та опціональне налаштування зовнішнього зворотного проксі.

Налаштування Dynamic DNS починається з реєстрації безкоштовного облікового запису на одному з DDNS-сервісів та створення доменного імені, наприклад myhome.duckdns.org. Далі на Raspberry Pi встановлюється DDNS-клієнт, що періодично оновлює запис A у DNS-зоні DDNS-сервісу, встановлюючи його значення рівним поточній публічній IP-адресі. Для

DuckDNS це може бути реалізовано через простий cron-завдання, що виконує HTTP-запит до API сервісу кожні 5 хвилин.

Після налаштування DDNS-клієнта отримане доменне ім'я вказується у параметрі dynamic_dns конфігураційного файлу homedock_server.conf. Серверне ядро використовує це значення для формування URL зовнішнього доступу, що відображається у стартовому журналі та може використовуватися користувачем для віддаленого доступу до cloud-середовища з будь-якого місця.

Оголошення сервісу cloud-середовища у локальній мережі через Zeroconf активується параметром local_dns у конфігураційному файлі. При запуску серверного ядра модуль hd_ThreadZeroConf створює фоновий потік, що за допомогою бібліотеки zeroconf для Python публікує mDNS-запис типу A, що пов'язує ім'я homedock.local з локальною IP-адресою Raspberry Pi.

Процес публікації виконується у окремому потоці (threading.Thread), результат якого перевіряється через спільну змінну thread_result. У разі успіху серверне ядро додає URL http://homedock.local (або https://homedock.local) до переліку адрес доступу у стартовому журналі. У разі невдачі (наприклад, якщо порт mDNS зайнятий іншим сервісом) виводиться відповідне попередження.

Користувач може звертатися до cloud-середовища за адресою homedock.local з будь-якого пристрою у тій самій локальній мережі, що підтримує mDNS. Підтримка mDNS є вбудованою у macOS, iOS, Windows 10 і новіші, Android та більшість дистрибутивів Linux (через демон avahi-daemon).

Автоматичне відкриття портів реалізується фоновим потоком hd_ThreadAutoPortRouting, що взаємодіє з маршрутизатором через протокол UPnP. При старті потік виконує SSDP-discover для пошуку маршрутизатора, що підтримує профіль Internet Gateway Device. Після виявлення маршрутизатора потік надсилає SOAP-запити на створення правил перенаправлення портів, що встановлюють відповідність між зовнішнім портом маршрутизатора та внутрішнім портом Raspberry Pi.

Правила перенаправлення створюються для портів, на яких функціонує cloud-середовище (зазвичай 80 та 443 для HTTP/HTTPS, а також порти встановлених контейнерних застосунків). Потік працює циклічно з визначеним інтервалом, оновлюючи правила для компенсації їх можливого видалення маршрутизатором після закінчення lease-часу або перезавантаження.

У деяких сценаріях розгортання (наприклад, при використанні провайдера з CGNAT, що унеможливлює прямий доступ з Інтернету, або при потребі у додатковому рівні захисту) доцільно використовувати зовнішній зворотний проксі-сервер. Серверне ядро cloud-середовища підтримує встановлення Nginx як зворотного проксі на тому ж Raspberry Pi або на окремому сервері. Nginx налаштовується для прийому TLS-з'єднань на портах 80 та 443, термінації SSL та перенаправлення розшифрованого трафіку до cloud-середовища на внутрішньому порту. Конфігурація включає директиви proxy_pass, proxy_set_header для передачі заголовків X-Forwarded-For, X-Forwarded-Proto та X-Forwarded-Host, а також параметри для підтримки WebSocket-з'єднань.

3.6 Тестування та верифікація працездатності

Після завершення розгортання та налаштування всіх компонентів cloud-середовища виконується комплексне тестування, спрямоване на верифікацію коректності функціонування, оцінку продуктивності, перевірку безпеки та стабільності. Тестування проводиться за п'ятьма напрямами.

Функціональне тестування охоплює перевірку коректності роботи всіх елементів веб-інтерфейсу cloud-середовища. Послідовно перевіряються: процес автентифікації (вхід з коректними та некоректними обліковими даними, завершення сесії, перевірка терміну дії сесії); навігація між розділами (Dashboard, App Store, Control Hub, Settings); коректність відображення менеджера вікон Prism (відкриття, закриття, переміщення, зміна розмірів вікон, перемикання фокусу, зміна тем); повний цикл керування застосунками

(встановлення, запуск, зупинка, перезапуск, оновлення, видалення); перегляд журналів контейнерів; зміна параметрів конфігурації через інтерфейс Settings.

Тестування виконується вручну у декількох браузерах (Google Chrome, Mozilla Firefox, Safari) для перевірки крос-браузерної сумісності. Для кожного тестового сценарію фіксується очікуваний та фактичний результат.

Тестування продуктивності спрямоване на оцінку поведінки cloud-середовища під навантаженням в умовах обмежених ресурсів Raspberry Pi. Вимірюються наступні показники: відсоток використання процесора (CPU usage) при різній кількості запущених контейнерів (1, 3, 5, 8, 12); обсяг зайнятої оперативної пам'яті, включаючи серверне ядро, Docker Engine та контейнери; температура SoC під навантаженням з використанням команди vcgencmd measure_temp; час відклику веб-інтерфейсу вимірюється через інструменти розробника браузера як TTFB – Time to First Byte.

Тестування проводилося у два етапи: у стані спокою із запущеним серверним ядром та без користувацьких контейнерів, а потім під робочим навантаженням із запущеними типовими сервісами. Результати фіксувалися у таблицях та візуалізуються у вигляді графіків для наочного представлення залежності споживання ресурсів від кількості контейнерів.

Особлива увага приділялася виявленню порогу, при якому температура SoC досягає критичних значень (85 °C для Raspberry Pi 4B), що спричинює автоматичне зниження тактової частоти процесора.

Тестування мережевого доступу перевіряє доступність cloud-середовища з різних точок підключення та за різними протоколами. Перевіряється доступ із локальної мережі за IP-адресою (http://<IP> або https://<IP>), за мультикастовим ім'ям (http://homedock.local), а також з Інтернету за публічною IP-адресою та за доменним ім'ям DDNS.

Для кожної точки доступу перевіряється коректність TLS-з'єднання (відсутність помилок сертифіката), працездатність перенаправлення з HTTP на HTTPS, коректність передачі заголовків X-Forwarded, а також працездатність

WebSocket-з'єднань. Тестування з Інтернету виконується з використанням мобільного пристрою, підключеного через мобільну мережу (4G/5G), що гарантує відсутність прямого мережевого з'єднання з локальною мережею Raspberry Pi.

Тестування стабільності спрямоване на перевірку здатності cloud-середовища безперебійно функціонувати протягом тривалого часу. Для цього cloud-середовище з декількома запущеними контейнерними застосунками залишається у безперервній роботі протягом щонайменше 72 годин (3 доби). Протягом цього періоду періодично перевіряються: доступність веб-інтерфейсу; коректність роботи запущених контейнерів; відсутність витоків пам'яті у серверному ядрі; стабільність температури SoC; коректність роботи фонових потоків (оновлення правил UPnP, моніторинг ресурсів).

По завершенні періоду тестування аналізуються журнали error.log серверного ядра на наявність критичних помилок та виняткових ситуацій. Також перевіряється коректність відновлення після примусового перезавантаження Raspberry Pi після перезапуску cloud-середовище повинно автоматично запуститися через systemd, а всі контейнери необхідно відновити роботу відповідно до їх політики перезапуску.

3.7 Результати реалізації

За результатами виконаних етапів реалізації отримано повнофункціональне персональне cloud-середовище на базі Raspberry Pi 4B з 8 ГБ оперативної пам'яті та зовнішнім SSD-накопичувачем. Cloud-середовище забезпечує веб-інтерфейс з менеджером вікон Prism, що доступний з будь-якого пристрою за протоколом HTTPS. Через вбудований App Store встановлено та успішно функціонують три контейнерних застосунки (Nextcloud, Jellyfin, Adminer), що демонструють основні сценарії використання – файлове сховище, медіасервер та адміністрування баз даних.

Доступ до cloud-середовища верифіковано з локальної мережі (за IP-адресою та за ім'ям homedock.local через mDNS), а також з Інтернету (за доменним ім'ям DDNS через UPnP-перенаправлення портів). Скріншоти веб-інтерфейсу, панелі Dashboard та Control Hub наведені у Додатку Г.

Для оцінки ефективності використання апаратних ресурсів виконано порівняння показників продуктивності cloud-середовища на Raspberry Pi 4B (4 ГБ) та Raspberry Pi 4B (8 ГБ). Вимірювання проведено у двох режимах: у стані спокою та під робочим навантаженням з п'ятьма запущеними контейнерами.

У стані спокою серверне ядро cloud-середовища разом із Docker Engine споживає приблизно 350–450 МБ оперативної пам'яті та 3–5% процесорного часу. Під робочим навантаженням з п'ятьма контейнерами споживання зростає до 1,5–2,5 ГБ оперативної пам'яті та 15–30% процесорного часу. Температура SoC з активним охолодженням стабілізується на рівні 45–55 °C, що є далеким від порогу (85 °C).

Результати порівняння демонструють, що конфігурація з 4 ГБ оперативної пам'яті є достатньою для одночасної роботи 3–5 контейнерних застосунків, тоді як конфігурація з 8 ГБ дозволяє комфортно утримувати 8–12 контейнерів. Raspberry Pi 5 з процесором Cortex-A76 (2,4 ГГц) забезпечить додатково приблизно 2–3-кратний приріст обчислювальної продуктивності та підтримку до 16 ГБ оперативної пам'яті, що дозволить масштабувати cloud-середовище до 15–20 одночасних контейнерів.

Для забезпечення доступності cloud-середовища для користувачів без технічного досвіду складено коротку інструкцію, що охоплює основні сценарії використання. Інструкція містить наступні кроки.

Для першого входу: відкрити веб-браузер та ввести адресу, вказану у стартовому журналі; ввести ім'я користувача та пароль за замовчуванням (passwd); змінити пароль на безпечний у розділі Settings.

Для встановлення застосунку: відкрити розділ App Store; знайти потрібний застосунок; натиснути кнопку Install; дочекатися завершення завантаження та запуску.

Для керування застосунками: перейти на Dashboard; використати кнопки Start, Stop, Restart для керування станом; натиснути на назву застосунку для відкриття його веб-інтерфейсу; використати кнопку Logs для перегляду журналів.

Для моніторингу: відкрити Control Hub для перегляду статистики використання CPU, пам'яті та мережі кожним контейнером.

У процесі реалізації та тестування виявлено низку обмежень розроблюваного програмно-технічного засобу. По-перше, обчислювальні ресурси Raspberry Pi є обмеженими у порівнянні з повноцінними серверами, що накладає обмеження на кількість одночасно запущених контейнерів та їх ресурсоємність. Застосунки з інтенсивним використанням процесора (наприклад, транскодування відео у реальному часі) можуть спричинювати помітну деградацію продуктивності.

По-друге, підтримка ARM64 з боку Docker-образів, хоча й постійно розширюється, все ще не охоплює деякі спеціалізовані застосунки, образи яких доступні лише для архітектури amd64. По-третє, мережева доступність з Інтернету залежить від конфігурації маршрутизатора та провайдера (наявність CGNAT, можливість UPnP або ручного перенаправлення портів), що у деяких випадках вимагає використання тунельних рішень.

До перспективних напрямів подальшого розвитку належать: реалізація автоматичного резервного копіювання даних контейнерів за розкладом; інтеграція з системами моніторингу та оповіщення; реалізація автоматичного оновлення SSL-сертифікатів через вбудований ACME-клієнт; розширення App Store додатковими категоріями застосунків (IoT, домашня автоматизація, засоби розробки); підтримка кластеризації декількох Raspberry Pi для горизонтального

масштабування; реалізація механізму аутентифікації з підтримкою двофакторної верифікації.

3.8 Висновки до розділу 3

У третьому розділі виконано програмно-апаратну реалізацію cloud-середовища HomeDock OS на базі одноплатного комп'ютера Raspberry Pi. Послідовно розглянуто всі етапи розгортання системи: підготовку апаратної платформи, встановлення та налаштування системного оточення, розгортання серверного ядра і клієнтської частини, забезпечення мережевої доступності, а також проведення комплексного тестування та оцінювання результатів реалізації.

У процесі підготовки апаратної платформи сформовано оптимальну конфігурацію на базі Raspberry Pi 4B з 8 ГБ оперативної пам'яті, зовнішнім SSD-накопичувачем та активним охолодженням, що забезпечило стабільне функціонування cloud-середовища в умовах тривалого навантаження. Реалізовано розгортання Raspberry Pi OS Lite з попереднім налаштуванням SSH-доступу, параметрів мережі та базових механізмів безпеки.

У ході розгортання системного оточення встановлено та налаштовано Docker Engine, Docker Compose, Python 3 і Node.js, що сформувало повноцінну платформу для контейнеризованого виконання застосунків та функціонування серверного ядра HomeDock OS. Використання Docker забезпечило ізоляцію сервісів, спрощення керування застосунками та можливість масштабування cloud-середовища.

Під час реалізації HomeDock OS виконано інтеграцію Flask-бекенду, Vue 3-клієнта та Docker-інфраструктури у єдину програмну систему. Реалізовано модульну серверну архітектуру з підтримкою API-маршрутів, фонового моніторингу, автоматичного керування контейнерами, підтримки SSL, Zeroconf,

Dynamic DNS та UPnP. Клієнтська частина забезпечує сучасний веб-інтерфейс із менеджером вікон Prism, адаптивною верстkою та підтримкою роботи на різних типах пристроїв.

Проведене тестування підтвердило коректність функціонування cloud-середовища у локальній мережі та через Інтернет, стабільність роботи контейнерів, працездатність механізмів автоматичного запуску та прийнятний рівень продуктивності навіть за обмежених апаратних ресурсів Raspberry Pi. Отримані результати демонструють можливість ефективного використання одноплатних комп’ютерів як основи персональних cloud-систем для домашнього або малого офісного використання.

За результатами реалізації встановлено, що Raspberry Pi 4B з 8 ГБ оперативної пам’яті забезпечує стабільне функціонування cloud-середовища з одночасною роботою декількох контейнеризованих сервісів, а використання сучасних веб- та контейнерних технологій дозволяє створити доступне, масштабоване та енергоефективне рішення персонального cloud-середовища.

ВИСНОВКИ

У першому розділі проведено аналіз зроблено постановку задачі розроблення програмно-технічного засобу реалізації cloud-середовища на базі Raspberry Pi.

У другому розділі роботи виконано проєктування програмно-технічного засобу реалізації персонального cloud-середовища на базі платформи Raspberry Pi. У результаті проведеного аналізу визначено концептуальну модель системи, сформовано її багаторівневу архітектуру та обґрунтовано вибір апаратних і програмних компонентів, необхідних для побудови сучасного cloud-рішення.

У третьому розділі виконано програмно-апаратну реалізацію cloud-середовища HomeDock OS на базі одноплатного комп’ютера Raspberry Pi. Послідовно розглянуто всі етапи розгортання системи: підготовку апаратної платформи, встановлення та налаштування системного оточення, розгортання серверного ядра і клієнтської частини, забезпечення мережевої доступності, а також проведення комплексного тестування та оцінювання результатів реалізації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Butschan J., Heidenreich S., Weber B., Kraemer T. Tackling hurdles to digital transformation – The role of competencies for successful industrial internet of things (IIoT) implementation. *International Journal of Innovation Management*. 2019. Vol. 23, No. 04. P. 1950036.

2. Bhandari B. A hybrid IIoT-based system for real-time monitoring and control of industrial processes. *J. Algebr. Stat*. 2020. Vol. 11, No. 1. P. 82–93.

3. On the performance of cloud services and databases for industrial IoT scalable applications / P. Ferrari, E. Sisinni, A. Depari et al. *Electronics*. 2020. Vol. 9, No. 9. P. 1435.

4. Jamil M. N., Schelén O., Monrat A. A., Andersson K. Enabling industrial internet of things by leveraging distributed edge-to-cloud computing: Challenges and opportunities. *IEEE Access*. 2024. Vol. 12. P. 127294–127308.

5. Haghnegahdar L., Joshi S. S., Dahotre N. B. From IoT-based cloud manufacturing approach to intelligent additive manufacturing: Industrial Internet of Things – An overview. *The International Journal of Advanced Manufacturing Technology*. 2022. Vol. 119, No. 3. P. 1461–1478.

6. IoT embedded cloud-based intelligent power quality monitoring system for industrial drive application / R. R. Singh, S. M. Yash, S. C. Shubham et al. *Future Generation Computer Systems*. 2020. Vol. 112. P. 884–898.

7. Ounifi H. A., Gherbi A., Kara N. Deep machine learning-based power usage effectiveness prediction for sustainable cloud infrastructures. *Sustainable Energy Technologies and Assessments*. 2022. Vol. 52. P. 101967.

8. Implementation of cloud based IoT technology in manufacturing industry for smart control of manufacturing process / S. I. Khan, C. Kaur, M. S. Al Ansari et al. *International Journal on Interactive Design and Manufacturing (IJIDeM)*. 2025. Vol. 19, No. 2. P. 773–785.

9. Varghese B., Wang N., Li J., Nikolopoulos D. S. Edge-as-a-service: Towards distributed cloud architectures. *arXiv preprint*. 2017. arXiv:1710.10090.

10. Khethavath P., Thomas J. P., Chan-Tin E. Towards an efficient distributed cloud computing architecture. *Peer-to-Peer Networking and Applications*. 2017. Vol. 10, No. 5. P. 1152–1168.

11. Nookala G., Gade K. R., Dulam N., Thumburu S. K. R. The Shift Towards Distributed Data Architectures in Cloud Environments. *Innov. Comput. Sci. J*. 2022. Vol. 11. P. 1695–1703.

12. Rehman A. U., Aguiar R. L., Barraca J. P. Fault-tolerance in the scope of cloud computing. *IEEE Access*. 2022. Vol. 10. P. 63422–63441.

13. Distributed cloud computing: architecture, enabling technologies, and open challenges / X. Q. Pham, T. D. Nguyen, T. Huynh-The et al. *IEEE Consumer Electronics Magazine*. 2022. Vol. 12, No. 3. P. 98–106.

14. Distributed cloud computing: Applications, status quo, and challenges / Y. Coady, O. Hohlfeld, J. Kempf et al. *ACM SIGCOMM Computer Communication Review*. 2015. Vol. 45, No. 2. P. 38–43.

15. Chekired D. A., Togou M. A., Khoukhi L. HybCon: A scalable SDN-based distributed cloud architecture for 5G networks. *IEEE Transactions on Cloud Computing*. 2021. Vol. 11, No. 1. P. 550–563.

16. Yangui S. A panorama of cloud platforms for IoT applications across industries. *Sensors*. 2020. Vol. 20, No. 9. P. 2701.

17. Saraswat M., Tripathi R. C. Cloud computing: Analysis of top 5 CSPs in SaaS, PaaS and IaaS platforms. *2020 9th International Conference System Modeling and Advancement in Research Trends (SMART)*. IEEE, 2020. P. 300–305.

18. Bhardwaj S., Jain L., Jain S. Cloud computing: A study of infrastructure as a service (IAAS). *International Journal of Engineering and Information Technology*. 2010. Vol. 2, No. 1. P. 60–63.

19. Suliman M. E., Madinah K. A brief analysis of cloud computing Infrastructure as a Service (IaaS). *International Journal of Innovative Science and Research Technology – IJISRT*. 2021. Vol. 6, No. 1. P. 1409–1412.

20. Samha A. K. Strategies for efficient resource management in federated cloud environments supporting Infrastructure as a Service (IaaS). *Journal of Engineering Research*. 2024. Vol. 12, No. 2. P. 101–114.

21. Isharufe W., Jaafar F., Butakov S. Study of security issues in platform-as-a-service (PaaS) cloud model. *2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*. IEEE, 2020. P. 1–6.

22. Fernandez J. Cloud Computing Revolution: Beyond Virtualization Scalability, Flexibility, and Efficiency in the Cloud Era. *MZ Comput. J*. 2023. Vol. 4. P. 1–5.

23. Mathur P. Cloud computing infrastructure, platforms, and software for scientific research. *High Performance Computing in Biomimetics: Modeling, Architecture and Applications*. 2024. P. 89–127.

24. Sukiasyan A., Badikyan H., Pedrosa T., Leitao P. Secure data exchange in Industrial Internet of Things. *Neurocomputing*. 2022. Vol. 484. P. 183–195.

25. Energy-efficient industrial internet of things: Overview and open issues / W. Mao, Z. Zhao, Z. Chang et al. *IEEE Transactions on Industrial Informatics*. 2021. Vol. 17, No. 11. P. 7225–7237.

26. An efficient algorithm for data transmission certainty in IIoT sensing network: A priority-based approach / K. G. Nalbant, S. Almutairi, A. H. Alshehri et al. *PLoS ONE*. 2024. Vol. 19, No. 7. P. e0305092.

27. Emerging network technologies for digital transformation: 5G/6G, IoT, SDN/IBN, cloud computing, and blockchain / M. Beshley, M. Klymash, I. Scherm et al. *IEEE International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering*. Cham : Springer Nature Switzerland, 2022. P. 1–20.

28. Public key encryption with equality test for Industrial Internet of Things system in cloud computing / G. G. Deverajan, V. Muthukumaran, C. H. Hsu et al. *Transactions on Emerging Telecommunications Technologies*. 2022. Vol. 33, No. 4. P. e4202.

29. Neural networks for cloud-based industrial internet of things / S. Kumar, Y. Lalitha Kameswari, S. Koteswara Rao et al. *Smart Computing Techniques in Industrial IoT*. Singapore : Springer Nature Singapore, 2024. P. 41–60.

30. A scalable two-layer blockchain system for distributed multicloud storage in IIoT / T. Xu, T. Qiu, D. Hu et al. *IEEE Transactions on Industrial Informatics*. 2022. Vol. 18, No. 12. P. 9173–9183.

31. Optimizing Latency Sensitive Applications for Amazon's Public Cloud Platform / J. Czentye, I. Pelle, A. Kern et al. *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2019. P. 1–7.

32. Kayaduman Ü., Kale T., Sözen N. Transitioning IIoT data processing: An experience with InfluxDB on Kubernetes. *2024 11th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, 2024. P. 247–252.

33. Mustyala A., Allam K. Automated Scaling and Load Balancing in Kubernetes for High-Volume Data Processing. *ESP Journal of Engineering and Technology Advancements*. 2023. Vol. 2, No. 1. P. 23–38.

34. Kasula V. K., Yadulla A. R., Konda B., Yenugula M. Fortifying cloud environments against data breaches: A novel AI-driven security framework. *World J. Adv. Res. Rev*. 2024. Vol. 24. P. 1613–1626.

35. Kumar R., Agrawal N. Analysis of multi-dimensional Industrial IoT (IIoT) data in Edge–Fog–Cloud based architectural frameworks: A survey on current state and research challenges. *Journal of Industrial Information Integration*. 2023. Vol. 35. P. 100504.

36. Ahmadi S. Zero trust architecture in cloud networks: Application, challenges and future opportunities. *Journal of Engineering Research and Reports*. 2024. Vol. 26, No. 2. P. 215–228.

37. A blockchain-enabled privacy-preserving verifiable query framework for securing cloud-assisted industrial internet of things systems / M. S. Rahman, I. Khalil, N. Moustafa et al. *IEEE Transactions on Industrial Informatics*. 2021. Vol. 18, No. 7. P. 5007–5017.

38. Halder S., Newe T. Enabling secure time-series data sharing via homomorphic encryption in cloud-assisted IIoT. *Future Generation Computer Systems*. 2022. Vol. 133. P. 351–363.

39. An Efficient IIoT Gateway for Cloud–Edge Collaboration in Cloud Manufacturing / Y. Zhang, D. Tang, H. Zhu et al. *Machines*. 2022. Vol. 10, No. 10. P. 850. DOI: 10.3390/machines10100850.

40. Secure Cloud-based Remote Monitoring of Environmental Factors using Mobile and Web Apps for Industry Automation / S. R., S. C. Thomas, A. Mathews et al. *2021 International Conference on Computational Performance Evaluation (ComPE)*, Shillong, India. IEEE, 2021. P. 282–287. DOI: 10.1109/ComPE53109.2021.9752238.

41. Jun C., Lee J. Y., Kim B. H. Cloud-based big data analytics platform using algorithm templates for the manufacturing industry. *International Journal of Computer Integrated Manufacturing*. 2019. Vol. 32, No. 8. P. 723–738. DOI: 10.1080/0951192X.2019.1610578.

42. Patel P. K., Dave D. P., Solanki K., Modi K. Improving Cloud Integrated Sensor Network Architecture, Applications & Challenges. *2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*, Erode, India. IEEE, 2021. P. 102–108. DOI: 10.1109/ICCMC51019.2021.9418370.

43. Towards Distributed IoT/Cloud based Fault Detection and Maintenance in Industrial Automation / A. Xenakis, A. Karageorgos, E. Lallas et al. *Procedia Computer Science*. 2019. Vol. 151. P. 683–690. DOI: 10.1016/j.procs.2019.04.091.

44. Multi-objective resource allocation for Edge Cloud based robotic workflow in smart factory / M. Afrin, J. Jin, A. Rahman et al. *Future Generation Computer Systems*. 2019. Vol. 97. P. 119–130. DOI: 10.1016/j.future.2019.02.062.

45. Park Y., Woo J., Choi S. A Cloud-based Digital Twin Manufacturing System based on an Interoperable Data Schema for Smart Manufacturing. *International Journal of Computer Integrated Manufacturing*. 2020. Vol. 33, No. 12. P. 1259–1276. DOI: 10.1080/0951192X.2020.1815850.

46. Chen Y. Intelligent algorithms for cold chain logistics distribution optimization based on big data cloud computing analysis. *Journal of Cloud Computing*. 2020. Vol. 9. P. 37. DOI: 10.1186/s13677-020-00174-x.

47. Gammelgaard B., Nowicka K. Next generation supply chain management: the impact of cloud computing. *Journal of Enterprise Information Management*. 2024. Vol. 37, No. 4. P. 1140–1160. DOI: 10.1108/JEIM-09-2022-0317.

48. Kholidy H. A. An Intelligent Swarm Based Prediction Approach For Predicting Cloud Computing User Resource Needs. *Computer Communications*. 2020. Vol. 151. P. 133–144. DOI: 10.1016/j.comcom.2019.12.028.

49. AI-Based Sustainable and Intelligent Offloading Framework for IIoT in Collaborative Cloud-Fog Environments / M. Kumar, G. K. Walia, H. Shingare et al. *IEEE Transactions on Consumer Electronics*. 2024. Vol. 70, No. 1. P. 1414–1422. DOI: 10.1109/TCE.2023.3320673.

50. Alzubi J. A., Alzubi O. A., Singh A., Ramachandran M. Cloud-IIoT-Based Electronic Health Record Privacy-Preserving by CNN and Blockchain-Enabled Federated Learning. *IEEE Transactions on Industrial Informatics*. 2023. Vol. 19, No. 1. P. 1080–1087. DOI: 10.1109/TII.2022.3189170.

ДОДАТОК А

Опис модулів пристрою

До складу pymodules входить кілька десятків модулів, які можна згрупувати за функціональним призначенням. Група керування конфігурацією: hd_FunctionsGlobals, hd_FunctionsConfig, hd_FunctionsInitUserFolders, hd_FunctionsActiveInstance – відповідають за завантаження та валідацію конфігурації, ініціалізацію користувацьких директорій та забезпечення запуску лише одного екземпляра серверного ядра.

Група веб-сервера та маршрутизації hd_HDOSWebServerInit, hd_RouteModules, hd_AppFilters – ініціалізують Flask-застосунок, реєструють обробники маршрутів і додають користувацькі фільтри шаблонів.

Група безпеки hd_NonceGenerator, hd_CSPMaxed, hd_FunctionsNativeSSL, hd_HTTPRedirector, hd_ApplyUploadLimits – забезпечують генерацію CSP-nonce, встановлення заголовків безпеки, роботу з SSL-сертифікатами, перенаправлення з HTTP на HTTPS та обмеження розміру завантажень.

Група мережевих функцій hd_FunctionsNetwork, hd_ThreadAutoPortRouting, hd_ThreadZeroConf – виявляють локальну та публічну IP-адреси, здійснюють автоматичне прокидання портів через UPnP та оголошення сервісу через mDNS.

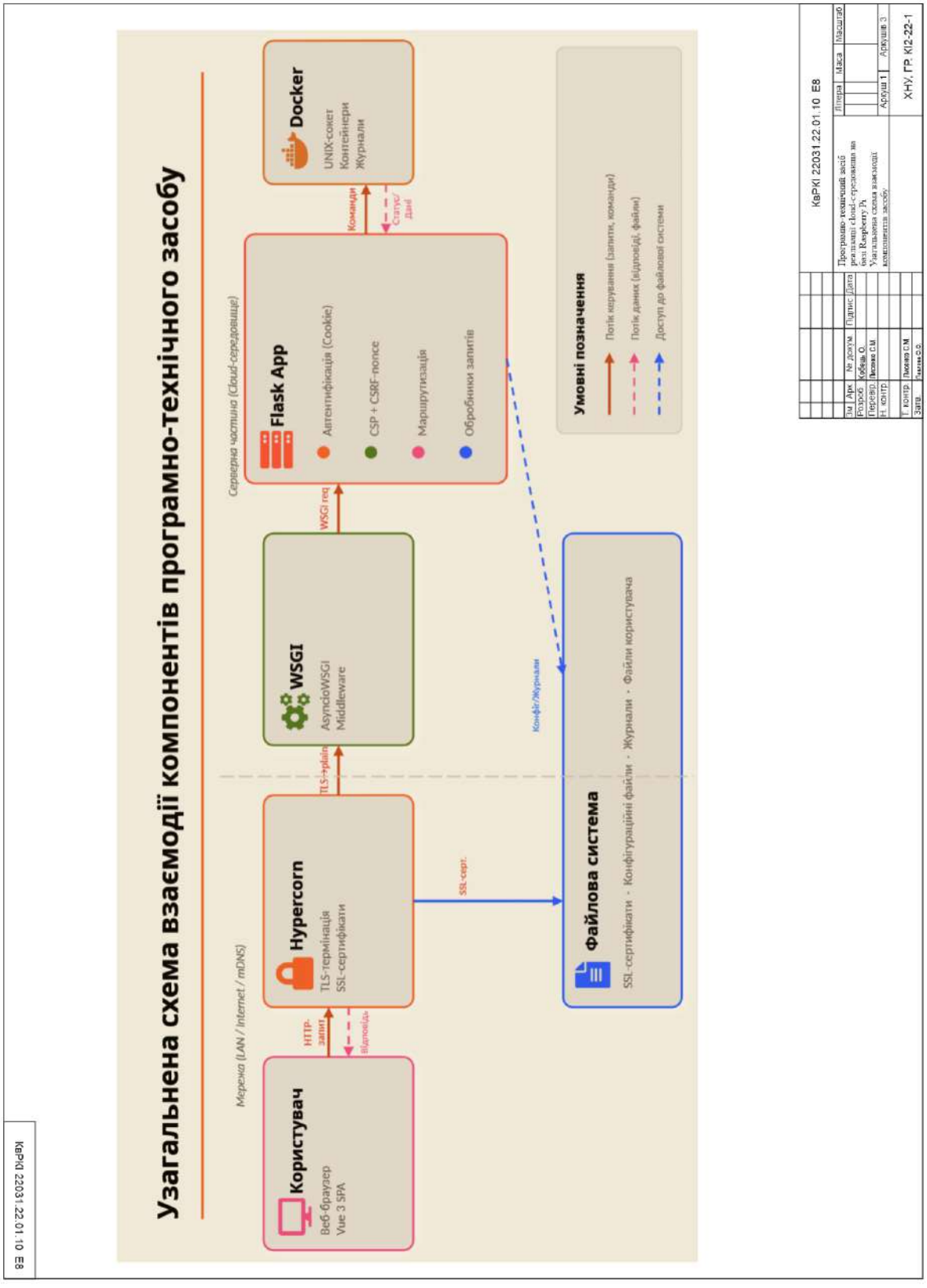
Група взаємодії з Docker та застосунками: hd_ThreadContainerResourceUsage, hd_ThreadAppUpdatesChecker, hd_UpdateDeps – моніторять використання ресурсів контейнерами, перевіряють наявність оновлень та оновлюють залежності самого серверного ядра.

Допоміжна група hd_HTMLErrorCodeHandler, hd_HMRUpdate, hd_FunctionsHostSelector, hd_FunctionsMain, hd_PublicKeySender, hd_EnterpriseLoader – реалізують обробку HTTP-помилок, підтримку гарячої заміни модулів у режимі розробки, виявлення середовища виконання, валідацію встановлення Docker та додаткові функції.

ДОДАТОК Б

(обов’язковий)

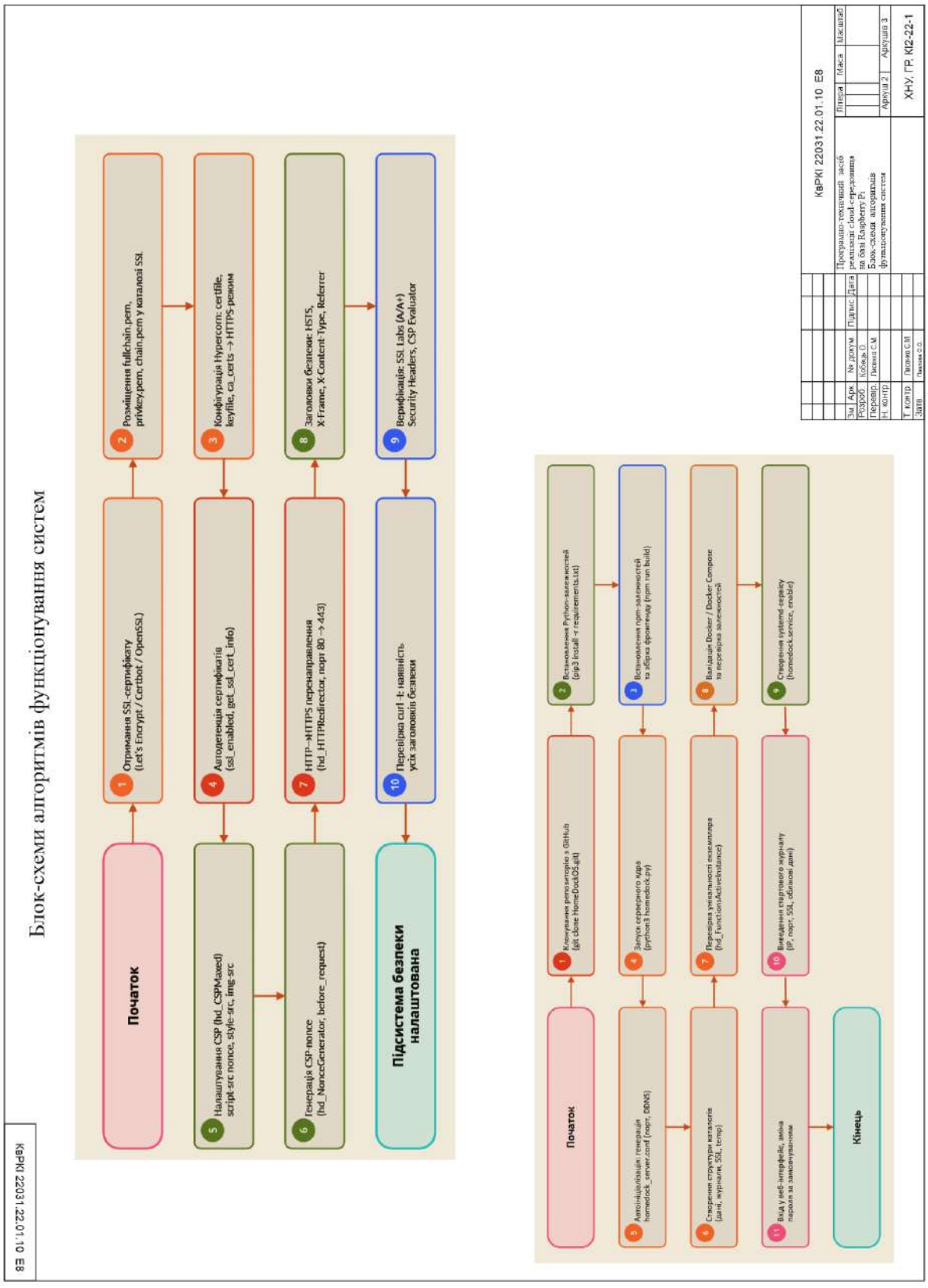
Копія креслення «Узагальнена схема взаємодії компонентів засобу»



ДОДАТОК В

(обов’язковий)

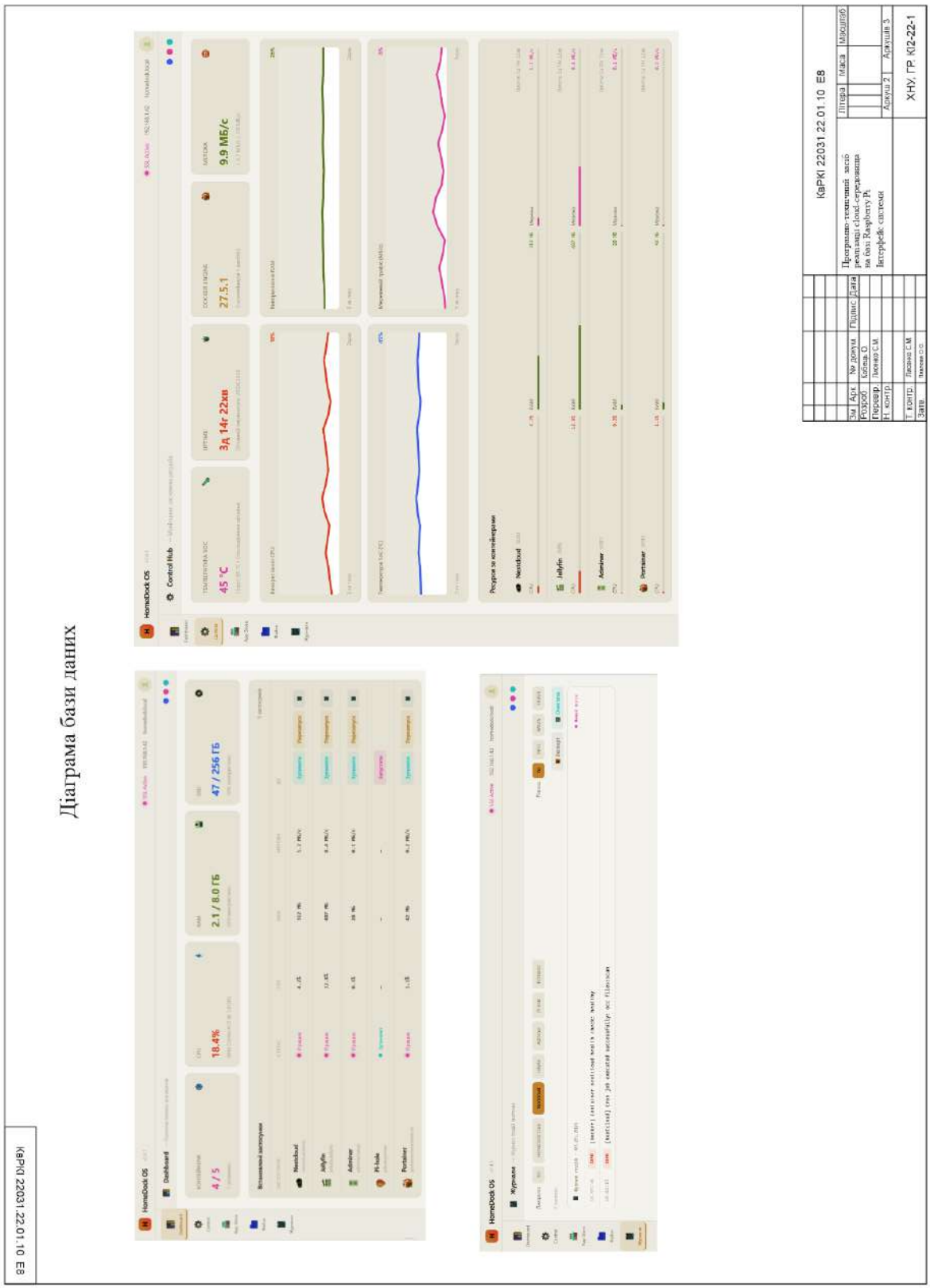
Копія креслення «Блок-схеми алгоритмів функціонування систем»



ДОДАТОК Г

(обов’язковий)

Копія креслення «Інтерфейс системи»



Tue May 26 08:47:39 EEST 2026, Медзатий Дмитро Миколайович, Хмельницький національний університет, ХНУ

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

Максимальне співпадіння з одним документом 26.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 14%**

	Документ		Сумарний збіг по Базі Даних	
ID: 272266 Назва: БКР Програмно-технічний засіб реалізації cloud?середовища на базі Raspberry?Pi Додано в БД: 2026-05-26 Автора: Олена КОБЕЦЬ Керівники: Сергій ЛИСЕНКО Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	124507	708	33965 (27%)	202 (29%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
269574	Назва: Звіт з ПДП Програмно-технічний засіб реалізації cloud?середовища на базі Raspberry?Pi. Додано в БД: 2026-02-27 Автора: Нічепорук А.О. Керівники: Кобець О.Д Консультанти: Опоненти:	32652 (26.0%)	187 (26.0%)

Протокол аналізу звіту подібності експертом

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

Автор: Олена КОБЕЦЬ

Співавтор:

Назва: Програмно-технічний засіб реалізації cloud-середовища на базі Raspberry Pi

Експерт: Сергій ЛИСЕНКО

Підрозділ: Кафедра комп'ютерної інженерії та інформаційних систем

Коефіцісит подібності 1:3.86%

Коефіцісит подібності 2:1.3%

Мікропробіли: 10

Заміна букв: 0

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2026-05-26 08:03:36.0

Після аналізу Звіту подібності констатую наступне:

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

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

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

Обґрунтування:

Дата експерт

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Кобець Олена Дмитрівна

Тема: Програмно-технічний засіб реалізації cloud-середовища на базі Raspberry Pi

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень ___3____ Кількість сторінок записки ___65___

1. Короткий зміст роботи та прийнятих рішень: Метою кваліфікаційної роботи є роботи є проєктування, реалізація та тестування cloud-середовищ на базі Raspberry Pi.

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

3. У першому розділі проведено аналіз зроблено постановку задачі розроблення програмно-технічного засобу реалізації cloud-середовища на базі Raspberry Pi. У другому розділі роботи виконано проєктування програмно-технічного засобу реалізації персонального cloud-середовища на базі платформи Raspberry Pi. У результаті проведеного аналізу визначено концептуальну модель системи, сформовано її багаторівневу архітектуру та обґрунтовано вибір апаратних і програмних компонентів, необхідних для побудови сучасного cloud-рішення. У третьому розділі виконано програмно-апаратну реалізацію cloud-середовища HomeDock OS на базі одноплатного комп'ютера Raspberry Pi. Послідовно розглянуто всі етапи розгортання системи: підготовку апаратної платформи, встановлення та налаштування системного оточення, розгортання серверного ядра і клієнтської частини, забезпечення мережевої доступності, а також проведення комплексного тестування та оцінювання результатів реалізації.

4. Позитивні сторони роботи: висока практична цінність роботи.

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

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

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

8. Інші зауваження: _____

9. Оцінка дипломної роботи: задовільно (С / 70)

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

Бедрамович Ліна Петрівна, зав.каф. ІПЗ, ХНУ

"___"_____2026 р. _____(підпис)

Зав. кафедри КІІС
д-р. філософії Ользі ПАВЛОВІЙ

Олена КОБЕЦЬ

ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи КІ2-22-1

ЗАЯВА

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

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

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

1 травня 2026 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

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

Назва кваліфікаційної роботи Операційна система реального часу для пристроїв IoT
Автор Олена КОБЕЦЬ
Освітня програма Комп'ютерна інженерія та програмування
Рівень вищої освіти перший (бакалаврський)
Спеціальність 123 Комп'ютерна інженерія
Науковий керівник: д.т.н., професор Сергій ЛИСЕНКО

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

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

Підтвердження:

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

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

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

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

4) збіг зі звітом з ПДП Програмно-технічний засіб реалізації cloud середовища на базі Raspberry Pi.Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає 3.86%; та системою Anti-Plagiarism складає 26%, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

01.06.2026

Завідувач кафедри _____ Підпис — Ольга ПАВЛОВА (Ім'я, ПРІЗВИЩЕ)

Гарант освітньої програми _____ Підпис — Андрій ШЧЕПОРУК (Ім'я, ПРІЗВИЩЕ)

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