Міністерство освіти і науки України

Луцький національний технічний університет

(повне найменування закладу вищої освіти)

Факультет комп'ютерних та інформаційних технологій

(повне найменування факультету)

Кафедра комп'ютерної інженерії та безпеки

(повне найменування кафедри)

КВАЛІФІКАЦІЙНА РОБОТА ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «МАГІСТР»

АНАЛІЗ ВПЛИВУ АПАРАТНИХ РЕСУРСІВ НА ПРОДУКТИВНІСТЬ CI/CD СИСТЕМИ З ВИКОРИСТАННЯМ SERVERLESS АРХІТЕКТУРИ

ANALYSIS OF THE HARDVARE RESOURCES IMPACT ON THE CI/CD SYSTEM PERFORMANCE BY SERVERLESS ARCHITECTURE

спеціальність 123 Комп'ютерна інженерія
(шифр і назва спеціальності)
освітня програма «Комп'ютерна інженерія»
(назва освітньої програми)

Виконав: здобувач вищої освіти
групи КІм-21
Войтович Микола Володимирович

_____
(підпис)

Керівник: к.т.н., доцент
Мельник Катерина Вікторівна

_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«___» грудня 2025 р.
Гарант освітньої програми:
к.т.н., доцент Гринюк Сергій Васильович

_____
(підпис)

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

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

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Войтовичу Миколі Володимировичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Аналіз впливу апаратних ресурсів на продуктивність CI/CD системи з використанням serverless архітектури*

Керівник роботи *к.т.н., доцент Мельник Катерина Вікторівна*

затверджені наказом закладу вищої освіти від «17» червня 2025 року № 290/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи 09.12.2025р.

3. Вихідні дані до роботи *Джерелом розробки є науково-технічна література та публікації в періодичних виданнях з даного питання, опубліковані зарубіжні та вітчизняні роботи в даній області, різні інтернет-ресурси технічного спрямування*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Вступ
Аналіз існуючих рішень для CI/CD систем
Опис розробленої CI/CD системи
Експериментальна частина
Висновки

5. Перелік графічного (ілюстративного) матеріалу:

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз існуючих рішень для CI/CD систем	*Мельник К.В., доцент*		
Теоретичне дослідження та реалізація нової CI/CD системи	*Мельник К.В., доцент*		
Аналіз впливу апаратних ресурсів на продуктивність serverless у CI/CD системі	*Мельник К.В., доцент*		
Нормоконтроль	*Багнюк Н.В., доцент*		
Гарант ОП	*Гринюк С.В., доцент*		
Показник запозичень тексту	***___%***		
Академічна доброчесність	*Міскевич О.І., ст.викладач*		

7. Дата видачі завдання 18.06.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	*Огляд літератури із досліджуваної проблеми*	До 01.08.2025 р.	
2.	*Аналіз існуючих рішень для CI/CD систем*	До 20.08.2025 р.	
3.	*Теоретичне дослідження та реалізація нової CI/CD системи*	До 25.09.2025 р.	
4.	*Аналіз впливу апаратних ресурсів на продуктивність serverless у CI/CD системі*	До 20.10.2025 р.	
5.	*Висновки та пропозиції*	До 25.10.2025 р	
6.	*Формування списку використаних джерел*	До 27.10.2025 р.	
7.	*Формування додатків*	До 30.10.2025 р.	
8.	*Оформлення ілюстративного матеріалу*	До 05.11.2025 р.	
9.	*Представлення остаточного варіанту кваліфікаційної роботи керівникові*	До 11.11.2025 р.	
10.	*Нормоконтроль*	До 29.11.2025 р.	
11.	*Інструментальна перевірка на академічний плагіат*	До 02.12.2025 р.	
12.	*Здача кваліфікаційної роботи та всіх супровідних документів на кафедру*	До 09.12.2025 р.	

Здобувач вищої освіти _____ Войтович М.В.
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ Мельник К.В.
(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Войтович М. В. Аналіз впливу апаратних ресурсів на продуктивність CI/CD системи з використанням serverless архітектури.

Кваліфікаційна робота магістра ОП «Комп'ютерна інженерія» спеціальності 123 Комп'ютерна інженерія. Луцький національний технічний університет. Луцьк, 2025.

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

Перший розділ присвячено огляду сучасних підходів до побудови CI/CD систем та аналізу існуючих архітектур для проміжних середовищ розробки. Розглянуто переваги та недоліки традиційних серверних рішень на базі віртуальних машин, а також особливості serverless-архітектур, що забезпечують автоматичне масштабування та оплату за фактичне використання ресурсів.

В другому розділі здійснено вибір та обґрунтування технологій, необхідних для реалізації serverless CI/CD системи. Обрано стек інструментів: GitHub Actions, Laravel Vapor, AWS Lambda та пов'язані хмарні сервіси для виконання застосунку. Описано принципи інтеграції з Git, підходи до автоматичного створення окремих середовищ для кожної гілки та керування тимчасовими ресурсами.

Третій розділ присвячено порівняльному аналізу роботи CI/CD систем на базі serverless та EC2 інфраструктури. Описано модель тестування, моделі виконання навантаження та критерії оцінки. Досліджено час розгортання середовищ, стабільність роботи, продуктивність та фінансові витрати при різних рівнях навантаження. Наведено графічні та табличні результати, продемонстровано відмінності у масштабованості, автоматизації та потребі у людських ресурсах.

Ключові слова: CI/CD, serverless, AWS Lambda, Laravel Vapor, GitHub Actions, EC2, автоматизація розгортання, проміжні середовища.

ANNOTATION

Voitovych M. Analysis of the hardware resources impact on the CI/CD system performance by serverless architecture.

Qualifying work of a Master's of EP «Computer Engineering» specialty 123 Computer Engineering. Lutsk National Technical University. Lutsk, 2025.

Qualification work consists of an introduction, three sections, a conclusion, a references, and three appendices.

The first section is devoted to reviewing modern approaches to building CI/CD systems and analyzing existing architectures for intermediate development environments. The advantages and disadvantages of traditional server solutions based on virtual machines are considered, as well as the features of serverless architectures that provide automatic scaling and payment for actual resource usage.

The second section selects and justifies the technologies necessary for implementing a serverless CI/CD system. A stack of tools has been selected: GitHub Actions, Laravel Vapor, AWS Lambda, and related cloud services for application execution. The principles of Git integration, approaches to automatically creating separate environments for each branch, and the management of temporary resources are described.

The third section is devoted to a comparative analysis of the operation of CI/CD systems based on serverless and EC2 infrastructure. The testing model, load execution scenarios, and evaluation criteria are described. The deployment time of environments, stability, performance, and financial costs at different load levels are investigated. Graphical and tabular results are presented, demonstrating differences in scalability, automation, and human resource requirements.

Keywords: CI/CD, serverless, AWS Lambda, Laravel Vapor, GitHub Actions, EC2, deployment automation, staging environments.

ЗМІСТ

ВСТУП

Актуальність теми. З розвитком DevOps-практик виникає потреба у швидкому та економічному розгортанні проміжних середовищ. Віртуальні сервери мають обмежену масштабованість і потребують ручного управління ресурсами, тоді як serverless-архітектура дозволяє автоматично отримувати рівно стільки ресурсів, скільки необхідно. Тому дослідження продуктивності та впливу апаратних обчислень у CI/CD системах на базі serverless залишається актуальним та затребуваним.

Метою роботи є розробка та експериментальне дослідження CI/CD системи, що використовує serverless-підхід для автоматичного створення проміжних середовищ, та аналіз того, як зміна апаратних ресурсів впливає на час розгортання, масштабованість і фінансові витрати.

Об’єкт дослідження – сучасні підходи до побудови CI/CD систем та інфраструктур для проміжних середовищ розробки.

Предмет дослідження – порівняння впливу виділених віртуальних серверів та serverless-обчислень на швидкість, масштабованість та вартість роботи CI/CD системи.

Завдання, які необхідно виконати:

– проаналізувати існуючі підходи до побудови CI/CD систем та оцінити їх переваги та недоліки щодо використання обчислювальних ресурсів;

– реалізувати прототип CI/CD системи, що використовує безсерверну архітектуру для автоматичного створення проміжних тестових середовищ;

– дослідити вплив різних типів апаратних ресурсів (CPU, RAM, дискова підсистема, мережеві затримки) на швидкість та стабільність роботи CI/CD процесів.

Апробація результатів. Результати роботи представлені на Міжнародній науково-практичній конференції «Цифрова трансформація: виклики та стратегії», яка проводилася 25 лютого 2025 р., м . Луцьк [1].

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ CI/CD СИСТЕМ

1.1 Опис методології Git Flow та інших підходів

Сучасна розробка програмного забезпечення не можлива без використання системи контролю версій. Її використання забезпечує структуровану історію змін та можливість паралельної роботи декількох розробників. Git дає можливість організовувати командні процеси, забезпечує ізольованість змін і підтримує стабільність основних гілок проєкту. Система Git стала основою для різних методології розробки, які визначають правила створення гілок, інтеграції змін, управління релізами та виправленням помилок. Найбільш відомою та однією з перших структурованих систем стала методологія Git Flow, запропонована Vincent Driessen [2]. Вона була створена для проєктів, де релізи проходять певний життєвий цикл і потребують ретельної підготовки.

Загальна логіка Git Flow полягає в існуванні двох основних гілок main та develop, які визначають стан розробки. Main містить перевірений і стабільний код, що готовий до розгортання на робочих серверах. Develop виступає проміжною гілкою, де об’єднуються всі нові зміни перед їхнім потраплянням у робоче середовище. Будь-яка нова зміна починається зі створення окремої гілки feature, для ізольованої роботи над кодом, не впливаючи на інші частини коду. Це дозволяє відслідкувати усі зміни пов’язані з відповідним завданням. Коли зміни готові, вони відправляються у develop, а для оформлення релізу створюється гілка release, у якій проводиться фінальна підготовка, тестування та стабілізація. Якщо ж виникає необхідність оперативно виправити помилку у робочій версії, застосовується спеціальна гілка hotfix, яка обходить проміжні етапи та швидко потрапляє до main (рисунок 1.1). Завдяки такій структурі методологія забезпечує передбачуваність змін, чітке розмежування відповідальності і зручність підтримки масштабних проєктів. Така структура детально описана у книзі Pro Git [3].

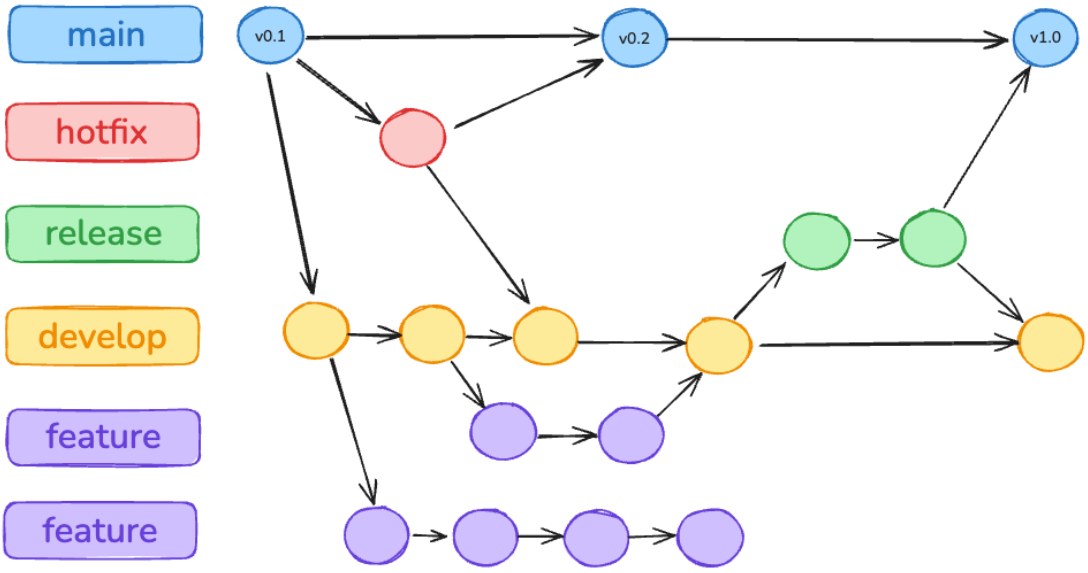

Рисунок 1.1 – Схема роботи Git Flow

Git Flow орієнтована на команди, де релізи мають етапність, а процеси розробки не вимагають миттєвої доставки функціональності. Поява DevOps-підходів і потреба швидко реагувати на бізнес-вимоги, сприяла появі простіших моделей. Одним із найвідоміших варіантів стала методологія GitHub Flow, що орієнтується лише на одну центральну гілку main. Будь-яка зміна починається зі створення короткотривалої гілки, яка після завершення роботи обов'язково проходить автоматичне тестування та перевірку коду [4]. Завдяки мінімалізму й простоті GitHub Flow підходить для веб-сервісів, що часто оновлюються, розгортаються декілька разів на день і не потребують складної підготовки релізів. Підхід передбачає миттєву інтеграцію змін, а отже, підвищує гнучкість команди та дозволяє прискорювати інновації.

Існує також методологія GitLab Flow (рисунок 1.2), яка поєднує гнучкість GitHub Flow із можливістю вести розробку з урахуванням окремих середовищ, таких як staging або pre-production. Вона орієнтована на ситуації, коли продукт повинен пройти проміжні рівні перевірок перед потраплянням у production, але водночас не потребує розгалуженої схеми гілок, як у Git Flow. GitLab Flow дозволяє адаптувати процес під потреби конкретної команди, додавати інтеграцію з відслідковуванням помилок та будувати гілки не лише від main, а й

від стабільних release міток [5]. Цей підхід часто використовується у DevOps-компаніях, де важливо синхронізувати зміни з реальними середовищами та автоматичним тестуванням.

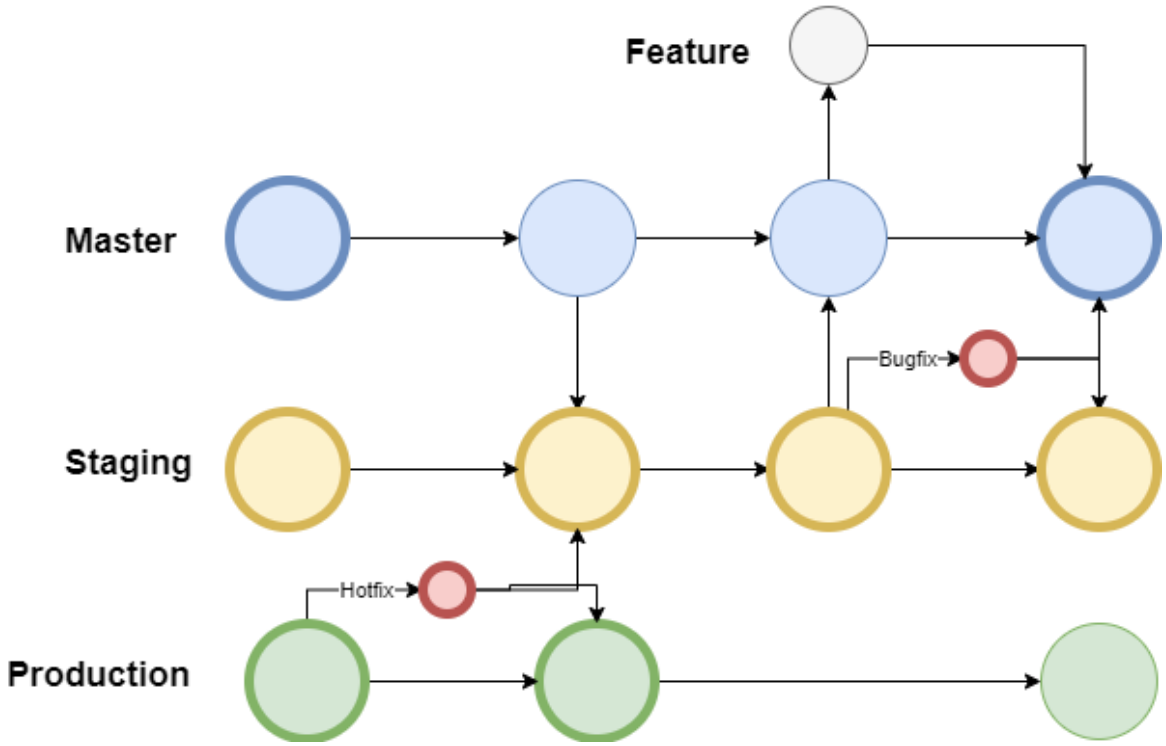

Рисунок 1.2 – Схема роботи Gitlab Flow

Поширення практик CI/CD та поява нових інстументів таких як terraform змінило роль моделей роботи з Git. Якщо раніше процес інтеграції був ручним і залежав від розробника, то зараз інструменти автоматизації відслідковують будь які зміни у гілках, запускають тести, виконують статичний аналіз і можуть створювати окремі середовища для кожного набору змін [6]. Основи автоматизованого розгортання та безперервної доставки сформульовані у фундаментальній праці Хамбла й Фарлі [7, с. 108]. Такий підхід отримав назву preview environments. Він передбачає, що кожна гілка або кожен pull-request має власне ізольоване середовище, доступне для тестування, перегляду інтерфейсу або демонстрації. Preview environments (рисунок 1.3) суттєво зменшують кількість помилок, які виникають після об’єднання коду, і дозволяють команді тестувальників бачити реальний стан продукту ще до релізу. Саме тому моделі

Git не розглядаються відокремлено від CI/CD, а навпаки, виступають однією з фундаментальних складових процесу автоматизації [8, с. 32].

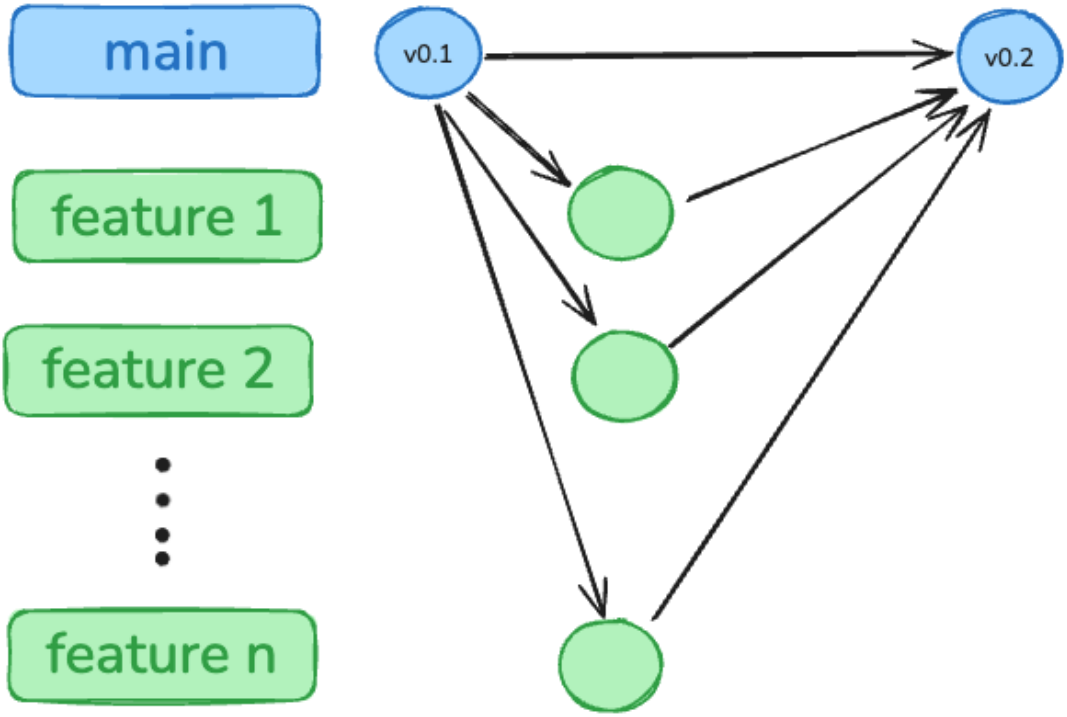

Рисунок 1.3 – Схема роботи проміжних середовищ

Створення проміжних середовищ з використанням традиційної серверної інфраструктури потребує значних ресурсів. Для розгортання окремого середовища необхідно володіти віртуальними серверами, базами даних, налаштуваннями мережі та засобами моніторингу. Кожне додаткове тестове середовище збільшує навантаження на апаратні ресурси, а у випадках, коли гілки створюються часто, резерви інфраструктури можуть вичерпуватися. Більш того, команди часто зустрічаються з ситуаціями, коли проміжні середовища працюють постійно, навіть тоді, коли в них уже немає потреби. Для прикладу, команда працює 8 годин, а середовища на сервері працюють безперервно. Це призводить до зайвих витрат і зменшення ефективності роботи інженерів.

На відміну від класичної моделі, serverless-сервіси не потребують постійно виділених серверів. Вони автоматично створюються за запитом, масштабуються відповідно до навантаження і видаляються, коли перестають бути потрібними. Такий підхід дозволяє створювати проміжні середовища практично миттєво, не утримуючи інфраструктуру в простої. У результаті управління гілками в Git напряму впливає на створення та видалення середовищ, формуючи повністю автоматизовану CI/CD систему, де ресурси витрачаються тільки тоді, коли вони реально потрібні.

Класичні методології, такі як Git Flow, залишаються актуальними для великих корпоративних проєктів, тоді як GitHub Flow та GitLab Flow набули популярності завдяки простоті та адаптивності. Усі вони є фундаментом для побудови систем, що забезпечують створення проміжних середовищ, прискорюють тестування й знижують ймовірність потрапляння помилок у робоче середовище. Саме тому аналіз підходів до організації гілок у Git є невід'ємною частиною дослідження автоматизованих CI/CD процесів та основою для переходу до serverless-рішень, у яких інфраструктура перестає бути статичною та перетворюється на керовану подіями, автоматичну і високоефективну.

1.2 Порівняння традиційних CI/CD систем зі serverless підходами

Розвиток технологій, хмарних сервісів та потреби бізнесу вимагають зменшення проміжку між розробкою нового функціоналу та впровадженням його для користувачів. Традиційні методи розробки програмного забезпечення такі як моделі Waterfall або V-Model, передбачають тривалі цикли створення продукту, складну комунікацію між розробниками та тестувальниками, та майже повну відсутність автоматизації. Вони виявилися неефективними у середовищах, де продукт потребує постійного оновлення [1, 2]. Поява методології DevOps змінила цю парадигму, поєднавши розробку і експлуатацію в єдиний комплекс процесів, які забезпечують безперервне вдосконалення продукту.

У центрі DevOps знаходиться CI/CD – набір практик, який охоплює два критично важливі напрямки: безперервну інтеграцію (Continuous Integration) та безперервне розгортання (Continuous Deployment). Поняття безперервної інтеграції означає, що всі зміни у вихідному коді регулярно об’єднуються у центральний репозиторій, після чого автоматично виконується збірка та тестування. Метою цього підходу є раннє виявлення помилок, усунення конфліктів при злитті гілок та забезпечення стабільної роботи продукту на усіх етапах розробки. Безперервне розгортання відповідає за автоматичне переміщення програмного продукту у тестові та робочі середовища, гарантуючи, що успішно перевірений код стане доступним користувачам без ручних втручань [4, 5].

CI/CD система складається з кількох компонентів. Першим є система контролю версій – найчастіше Git або один зі сервісів, що працюють на його основі: GitHub, GitLab або Bitbucket. Далі працює інструмент автоматизації, який реагує на події в репозиторії. Наприклад, створення нового коміту або pull-request викликає збірку застосунку, запуск тестів, статичного аналізу, збірку артефактів, міграцію бази даних, деплой у середовище або навіть створення нової інфраструктури [1, 4, 7]. Саме тому CI/CD розглядають як конвеєр – послідовний набір етапів, через які проходить кожна зміна у коді.

Раніше CI/CD реалізовували на фізичних чи віртуальних серверах. Компанії змушені були підтримувати виділені машини, на яких працювали build-системи, черги, тестові контейнери та середовища. Це створювало низку проблем: високу вартість інфраструктури, необхідність постійного адміністрування, обмеження масштабування та ризики простою під час пікового навантаження. Хоча гарантувало безпеку та надійність. Якщо розробники паралельно працювали над десятками функціональних гілок, для кожної з них потрібно було створювати окреме середовище або чергу на спільний сервер, що суттєво уповільнювало розробку, та вимагало більших затрат на апаратні ресурси [5, 6, с. 61].

Хмарні обчислення стали вирішенням цих обмеженнь. Компанії почали переносити CI/CD у AWS, Google Cloud, Azure, використовуючи кластери

контейнерів, керовані екземпляри баз даних, сховища артефактів та автоматизовані системи оркестрації. Згодом із розвитком хмар з'явилася нова парадигма – інфраструктура описана кодом. Його головна ідея полягає в тому, що розробник не керує серверами, операційними системами, оновленнями або масштабуванням. Він пише лише прикладний код, а всі ресурси виділяються автоматично, працюють стільки, скільки потрібно, і зникають після завершення виконання [7].

AWS Lambda є однією з найвідоміших serverless-платформ у світі. Вона дозволяє запускати окремі функції у відповідь на події: push у Git-репозиторій, виклик API, зміну у базі даних, подію з S3 або надходження повідомлення у чергу. Lambda не потребує налаштування серверів, а система автоматично масштабує функції при збільшенні навантаження. Це робить її особливо ефективною у контексті CI/CD, де більшість завдань виконуються короткочасно, з високою частотою викликів [9, 10].

Serverless у CI/CD можна застосовувати на різних етапах. Під час процесу інтеграції Lambda може запускати статичний аналізатор, перевіряти лінтинг, виконувати unit-тести або аналіз безпеки. Під час розгортання вона може формувати збірку, переміщувати файли в сховище, виконувати міграцію бази даних або створювати інфраструктуру для конкретної гілки. Для середовищ тестування це особливо важливо: система може автоматично створювати нову копію застосунку разом із базою даних та API, коли розробник відкриває pull-request. Після злиття гілки з основною цей ресурс видаляється, не залишаючи зайвих витрат [8, 9].

Економічна ефективність є однією з головних причин зростання популярності serverless-підходів. Традиційні сервери оплачуються навіть у періоди простою, тоді як Lambda, Cloud Run або Azure Functions тарифікуються за час виконання. Компанії з десятками розробників, які постійно створюють тестові гілки, заощаджують багато коштів, оскільки не утримують десятки екземплярів середовищ, які більшість часу не використовуються. Крім цього процеси не потрібно вручну запускати, очищати або конфігурувати.

Аналітика Google Cloud і DORA демонструє, що організації, які впровадили DevOps та автоматизовані конвеєри розгортання, демонструють у десятки разів швидший час виходу оновлень та різко нижчу кількість інцидентів після релізу [11]. Саме тому використання serverless у CI/CD стало фундаментальним у сучасній розробці, забезпечуючи стабільність, високу швидкість та передбачуваність життєвого циклу процесу розгортання.

1.3 Проблеми використання традиційної серверної архітектури для проміжних середовищ

Традиційна модель розгортання програмного забезпечення базувалася на використанні фізичних або віртуальних серверів, на яких встановлювалися необхідні компоненти для розміщення додатку, бази даних та системи керування процесами розробки [9]. Такий підхід довгий час не мав альтернатив та застосовувався в компаніях незалежно від масштабу та типу програмного продукту. Поява автоматизацій CI/CD-систем виявила, що традиційна серверна інфраструктура має значну кількість недоліків, які ускладнюють побудову сучасного гнучкого процесу доставки оновлень. Проблеми стають особливо відчутними під час розгортання проміжних середовищ, що використовуються для тестування та перевірки окремих функціональних гілок або pull-requests.

Важливим недоліком є суттєві затримки, пов’язані з часом розгортання середовищ. Віртуальний або виділений сервер потребує повноцінного циклу конфігурації: підняття операційної системи, встановлення залежностей, налаштування мережевих параметрів, конфігурування служби розгортання, бази даних та сховищ. У випадку, коли кожна нова гілка вимагає окремого середовища, затримка у кілька хвилин перетворюється на години. У великих проектах з десятками паралельних задач це призводить до серйозних затримок [11]. Розробники змушені чекати, поки середовище буде готове, або користуватися спільним середовищем, де зміни одного учасника впливають на

роботу інших. Таким чином, швидкість розробки знижується, що суперечить самій ідеї безперервного інтегрування.

Проблемою є також неминучий простій серверів. Традиційна інфраструктура передбачає, що навіть якщо середовище не використовується, сервер продовжує працювати й споживати ресурси. У типовому випадку проміжні середовища завантажені лише протягом короткого періоду активного тестування. Решту часу вони залишаються фактично неактивними, однак компанія все одно сплачує за апаратні ресурси або оренду обладнання. У разі постійного виділення серверів для кожної гілки кількість таких «зайвих» середовищ може досягати десятків. Це є прямим фінансовим збитком, особливо для стартапів або продуктів із високою частотою інтеграцій. При цьому адміністрування таких ресурсів також вимагає витрат часу: їх потрібно обслуговувати, оновлювати, контролювати їх стан та безпеку [11].

Навіть у випадку використання віртуальних серверів у хмарних провайдерів проблема залишається актуальною. Після створення середовища, воно функціонує постійно, незалежно від фактичного навантаження. Таким чином, модель «оплати за час існування серверу» стає неефективною у CI/CD, де відбуваються короткі цикли тестування та швидке злиття гілок. Якщо ж середовище видаляється вручну після завершення роботи, виникає інша складність – людський фактор. Розробники або DevOps-інженери можуть забути видалити непотрібні ресурси, що призводить до накопичення застарілих середовищ та додаткових витрат. У великих організаціях такі ситуації поширені та обліковуються як реальна фінансова проблема.

Ще одним недоліком є обмежена масштабованість традиційних серверів. Навіть якщо сервер орендований у хмарного провайдера та має гнучкі налаштування, його ресурси залишаються фіксованими. Якщо навантаження збільшується, система втрачає стабільність або потребує ручного масштабування, переналаштування та перезапуску процесів. У середовищах CI/CD навантаження різко змінюється: у періоди активної розробки build-скрипти, тести та розгортання запускаються десятки разів на годину, тоді

як у інший час навантаження мінімальне. Віртуальні сервери не здатні миттєво адаптуватися під такі коливання. Це призводить або до нестачі ресурсів у пікові моменти, або до перевитрати потужностей під час простою. В обох випадках компанія отримує неефективну інфраструктуру.

Традиційна серверна архітектура також створює ризики, пов’язані з підтримкою сумісності середовищ. Якщо кілька розробників працюють на різних гілках, а всі зміни тестуються на спільному сервері, можуть виникнути конфлікти версій, несумісні залежності або змішування даних. Інколи окремі тестувальники не здатні відтворити помилку, тому що середовище вже оновлене або змінене іншою командою. Це ускладнює налагодження та знижує достовірність результатів тестування. Для усунення таких проблем потрібні додаткові механізми ізоляції, такі як віртуальні машини чи контейнери, але їхнє використання знову призводить до збільшення вартості та ускладнює адміністрування.

Крім технічних та фінансових аспектів, традиційні сервери створюють додаткове навантаження на підтримку та безпеку. Сервери потребують оновлення, моніторингу, резервного копіювання, налаштування мережевих правил та контролю доступу. При відсутності автоматизації усі ці операції виконуються вручну, що збільшує ризик помилок та недоліків у безпеці. У контексті CI/CD, де середовища часто створюються і видаляються, постійне адміністрування стає особливо складним. Якщо команда не має достатнього досвіду у DevOps або системному адмініструванні, традиційна архітектура стає джерелом додаткових проблем [11].

Важливою проблемою є також обмеження швидкості інновацій. Коли інфраструктура залежить від серверів, частина часу розробників витрачається не на створення продукту, а на обслуговування середовищ, налаштування конфігурацій або усунення технічних збоїв. Це сповільнює цикл розробки. Синхронізація розробників, тестувальників та DevOps-інженерів у таких умовах ускладнюється. Будь-яка помилка на проміжному сервері блокує роботу всієї

команди. Особливо ці проблеми помітні для проектів із мікросервісною архітектурою, де необхідно координувати десятки незалежних компонентів.

На практиці традиційна серверна інфраструктура перестає масштабуватися разом із вимогами ринку. Компанії, які намагаються інтегрувати CI/CD у такий підхід, стикаються з тим, що сервери не відповідають гнучкості сучасної розробки. Якщо в проекті виникає потреба у швидкому створенні великої кількості середовищ, серверна архітектура виявляється надто повільною та дорогою. Багато компаній, усвідомлюючи ці обмеження, переходять до контейнеризації, але й вона не усуває основної проблеми – необхідності постійних обчислювальних ресурсів [10].

Таким чином, головними недоліками традиційної серверної архітектури для CI/CD є повільне розгортання середовищ, висока вартість підтримки, постійні простої, фіксовані ресурси, складнощі масштабування, високий ризик помилок конфігурації, труднощі ізоляції, людський фактор та зниження ефективності праці розробників. Це робить традиційні інфраструктурні рішення неефективними для сучасного процесу розробки, де швидкість, автоматизація та оптимізація витрат мають пріорітетне значення. Саме тому організації почали активно переходити до хмарних технологій та serverless-архітектури, що дозволяє виділяти ресурси лише у момент виконання завдань, автоматично масштабувати системи та будувати повністю самокеровані CI/CD процеси. У результаті компанії отримують динамічні проміжні середовища, які створюються і видаляються автоматично, знижують операційні витрати та підвищують якість продукту.

РОЗДІЛ 2
ОПИС РОЗРОБЛЕНОЇ CI/CD СИСТЕМИ

2.1 Архітектура системи

Для розробки CI/CD системи з розгортанням проміжних середовищ тестування було використано Laravel Vapor, AWS Lambda, Github Actions та відповідно система була протестована на розгортаннях Laravel застосунків.

2.1.1 Використання Laravel Vapor для автоматизації

Laravel Vapor – це платформа для розгортання Laravel-застосунків у безсерверній інфраструктурі AWS. Vapor формує архітектуру навколо AWS Lambda, API Gateway, S3, RDS та інших сервісів AWS, що дозволяє запускати код лише в моменти фактичного виконання. Це забезпечує еластичність масштабування, відсутність потреби в ручному адмініструванні серверів та суттєве зниження витрат на апаратні ресурси [12].

У межах розробленої CI/CD системи Laravel Vapor виконує роль ключового компонента автоматизації розгортання проміжних тестових середовищ. Основне завдання полягає в тому, щоб при створенні нової гілки у репозиторії GitHub автоматично створювалось окреме ізольоване середовище – з власною доменною адресою, конфігурацією, доступом до бази даних та виконанням усіх необхідних міграцій, та окремим змінними середовища. Vapor забезпечує цей процес за рахунок інтеграції з AWS Lambda та можливості створення окремих стеків розгортання, що існують незалежно один від одного.

Процес автоматизації складається з кількох етапів. На першому з них Laravel Vapor отримує конфігурацію проєкту vapor.yml, де визначається набір ресурсів, параметри середовища, назви сервісів і правила розгортання. При надходженні запиту на створення нового середовища CI/CD система викликає CLI-інструменти Vapor або API-драйвери й ініціює створення інфраструктури у хмарі. Далі формуються Lambda-функції, система маршрутизації, S3-сховище, налаштовуються змінні середовища та підключаються залежні сервіси. Фактично кожна гілка репозиторію отримує власний serverless-додаток, який

працює автономно й не потребує постійно запущених EC2-серверів. Зразок файлу vapor.yml наведений у лістингу 2.1.

Лістинг 2.1 – Зразок файлу vapor.yaml

```
id: 123
name: example
environments:
    production:
        memory: 3072
        cli-memory: 3072
        queue-memory: 3072
        queues:
          - main-production
        runtime: docker
        timeout: 600
        cli-timeout: 600
        queue-timeout: 600
        storage: prod
        database: prod
        domain: example.com
        build:
            - 'COMPOSER_MIRROR_PATH_REPOS=1 composer install --no-dev'
            - 'npm ci && npm run prod && rm -rf node_modules'
            - 'php artisan event:cache'
        deploy:
            - 'php artisan migrate --force'
        layers:
            - 'vapor:php-7.4'
            - 'vapor:php-7.4:imagick'
```

кінець лістингу 2.1

Важливою складовою є автоматизація міграцій, синхронізації статичних файлів та синхронізація бази даних. Vapor самостійно викликає необхідні команди Laravel, включаючи php artisan migrate та php artisan config:cache, передаючи їх у середовище Lambda. Також можна додати будь які всласні команди для додаткових налаштувань після того як середовище буде розгорнуте. Завдяки цьому розгортання виконується без участі розробника та не потребує додаткових систем керування конфігурацією. Для кешування та обробки статичних ресурсів Vapor використовує CDN CloudFront [13], що дозволяє отримати високу швидкість завантаження інтерфейсу навіть для тимчасових тестових середовищ.

Перевагою застосування Laravel Vapor у CI/CD є автоматичне видалення середовищ. Після інтеграції гілки в основну (наприклад, після pull request merge) CI/CD система ініціює команду видалення, і Vapor очищає всі відповідні Lambda-функції, DNS-записи, сховища та інші ресурси. Це усуває проблему накопичення застарілих середовищ, оптимізує фінансові витрати й забезпечує акуратність інфраструктури.

Таким чином, використання Laravel Vapor дозволяє автоматизувати процес створення й видалення проміжних середовищ, усунути потребу у постійних серверних ресурсах, забезпечити моментальне масштабування, високу продуктивність і відсутність операційних витрат на адміністрування. Завдяки інтеграції з AWS та підтримці архітектури serverless, рішення суттєво спрощує організацію CI/CD процесів і робить систему незалежною від традиційної серверної інфраструктури.

2.1.2 AWS Lambda для створення тестових середовищ

У розробленій CI/CD системі створення тимчасових тестових середовищ реалізовано на базі безсерверних обчислень AWS Lambda. Використання Lambda дозволяє виконувати автоматизовані операції з розгортання інфраструктури без утримання постійних серверів, оскільки виконання функції відбувається виключно в момент виникнення події [14]. Це мінімізує витрати, спрощує масштабування та усуває необхідність ручного втручання під час створення нових середовищ для перевірки працездатності коду.

Процес розгортання стартує в момент створення нової гілки у репозиторії. GitHub Actions формує запит до API Gateway AWS, який передає в Lambda службову інформацію, що містить назву гілки, унікальний ідентифікатор та параметри конфігурації проєкту. Lambda-функція обробляє отримані дані та ініціює створення ізольованого середовища, використовуючи інтеграцію з Laravel Vapor. У процесі виконання генерується окрема інфраструктура, яка включає домен, змінні середовища, власні ресурси S3 і базу даних. Після завершення розгортання функція викликає GitHub API та автоматично додає

посилання на створене середовище у pull request, що дозволяє команді розробників та тестувальнику одразу розпочати перевірку змін [15].

Конфігурація Lambda передбачає використання підтримуваного середовища виконання php, визначення обсягу оперативної пам'яті та обмеження часу виконання, що зазвичай встановлюється на рівні до 60 секунд. Доступ до керування ресурсами забезпечується IAM-політиками, що дозволяють взаємодіяти з Laravel Vapor, сервісом CloudFormation, S3, Secrets Manager і іншими компонентами AWS, необхідними для створення середовища. Змінні середовища Lambda зберігають параметри проєкту, шаблон домену та авторизаційні токени для взаємодії з GitHub [11]. На рисунку 2.1 зображена схема aws сервісів які використовуються для Laravel Vapor.

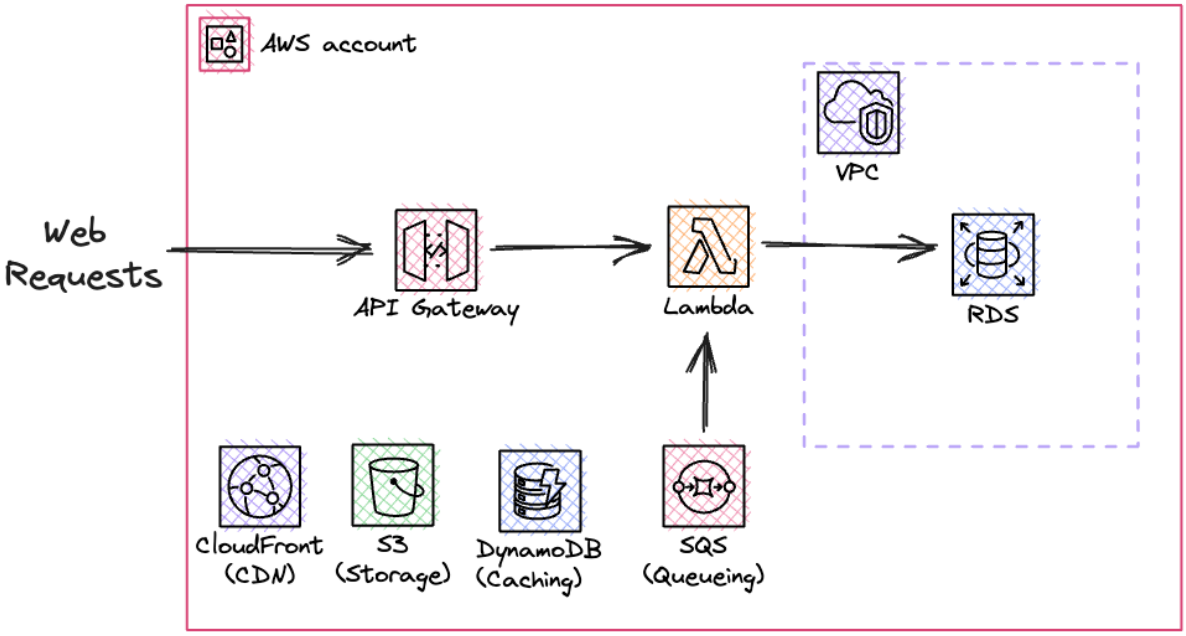

Рисунок 2.1 – Схема AWS сервісів використовуваних для Laravel Vapor [11]

Увесь перебіг виконання фіксується у CloudWatch Logs, що дає змогу аналізувати помилки та контролювати якість роботи механізму автоматичного розгортання [16]. Після завершення роботи над гілкою Lambda використовується також для зворотної операції – видалення середовища. При видаленні або злитті гілки CI-система повторно викликає функцію, яка очищує ресурси S3, видаляє

базу даних і скасовує зареєстровані домени. Таким чином, система не накопичує невикористані середовища і не витрачає додаткові кошти.

Застосування AWS Lambda дає можливість створювати середовище на вимогу без додаткових серверів, забезпечує масштабованість і автоматизацію процесу, що позитивно впливає на швидкість тестування, ізоляцію змін та загальну ефективність циклу розробки.

2.2 Алгоритм роботи системи: інтеграція з Git, створення і видалення середовищ

Алгоритм роботи розробленої CI/CD системи базується на повній інтеграції із системою керування версіями GitHub. Це дозволило автоматизувати життєвий цикл проміжних тестових середовищ: вони створюються у момент появи нової гілки та знищуються після завершення роботи над нею. Таким чином, інфраструктура існує лише тоді, коли вона потрібна, що усуває проблему постійного утримання окремих серверів або віртуальних машин.

Розгортання середовищ реалізовано за допомогою GitHub Actions – сервісу, здатного виконувати сценарії CI/CD після настання подій у репозиторії. Для цього було створено два окремих workflow-файли. Перший (deploy-preview) відповідає за автоматичне створення середовища при появі нової гілки з префіксом feature/**. Другий (delete-preview) – за його автоматичне видалення після того, як гілку видалено з репозиторію.

2.2.1 Автоматичне створення середовища

Коли розробник створює нову гілку для розробки функціоналу, у репозиторії активується подія push. GitHub Actions завантажує проєкт і розгортає тестове середовище, яке є точною копією робочого середовища [17]. У рамках першого етапу система перевіряє коректність змін. Для цього виконується встановлення залежностей PHP та JavaScript, розгортання локальної бази MySQL у Docker-контейнері та запуск міграцій. Це забезпечує можливість повного запуску застосунку у тестовому режимі.

Після цього запускається повний набір автоматизованих тестів: модульні тести Laravel, клієнтські тести, інтеграційне тестування і функціональні UI-тести через Cypress. Завдяки цьому система дає гарантію, що код не порушує роботу застосунку, перш ніж розгорнути його на реальній інфраструктурі. У випадку помилки збираються логи, журнали помилок та скріншоти тестів, а GitHub автоматично публікує їх як артефакти [17]. Це позбавляє потреби вручну відтворювати помилку на власному комп'ютері і значно пришвидшує процес виправлення.

Після успішного проходження тестів починається другий етап – розгортання середовища у хмарі AWS за допомогою Laravel Vapor. Використовуючи API-токен, система авторизується у Vapor, після чого назва гілки автоматично приводиться до нижнього регістру. Це робиться для уникнення конфліктів, оскільки імена середовищ у Vapor мають бути у форматі, придатному для використання в URL та доменних іменах. Далі сценарій створює або оновлює середовище: генерує конфігураційний файл, формує окрему базу даних, налаштовує змінні середовища, налаштовує S3 bucket та готує Docker-контейнер застосунку. Завершальний етап – виконання команди vapor deploy, яка завантажує програму, формує AWS Lambda-функції, налаштовує API Gateway та створює URL доступу до цього середовища. У результаті команда розробників отримує ізольоване середовище, доступне через унікальну адресу, де можна виконувати тестування, демонстрацію функціоналу або перевірку виправлених помилок [18].

Однією з ключових переваг такого підходу є відсутність ручного втручання. Створення середовища не потребує участі DevOps-інженера, не вимагає конфігурації серверів, підняття віртуальних машин або ручного розподілу ресурсів. Весь процес є повністю автоматизованим і відбувається незалежно від кількості гілок та частоти їх створення.

2.2.2 Автоматичне видалення середовища

Зворотна частина алгоритму активується тоді, коли розробник видаляє гілку з репозиторію. У традиційній серверній архітектурі це часто призводить до

накопичення невикористовуваних серверів або баз, які продовжують споживати ресурси, незважаючи на те, що вони більше нікому не потрібні. Завдяки окремому workflow delete-preview система контролює інфраструктуру і забезпечує її повне очищення.

Як тільки відбувається подія delete з гілкою формату feature/**, GitHub Actions запускає сценарій, який повторно інсталює Vapor CLI та авторизується у Vapor. Далі визначається назва середовища, що відповідає назві гілки. Система викликає команди vapor env:delete і vapor-cli database:delete, які видаляють Lambda-функції, інстанси бази даних і пов’язані ресурси. Після цього відбувається очищення S3 bucket, у якому могли зберігатися файли або артефакти цього середовища. Очищення проводиться примусово, що гарантує відсутність залишкових даних. У випадку будь-яких помилок застосунок вивантажує журнали, що дає можливість швидко діагностувати, чому саме певний ресурс не був видалений.

Завдяки такому підходу жодне тимчасове середовище не залишається «зайвим». AWS не продовжує нараховувати оплату за збереження Lambda-функцій, контейнерів, бази даних або сховищ, які більше не використовуються. Це особливо важливо для великих команд, де розробка може вестись у десятках паралельних гілок і кожна з них створювала б окрему інфраструктуру для тестувальників.

На всіх етапах роботи системи можна інтегрувати будь яку систему сповіщення, що дозволить розробникам отримувати повідомлення про результат виконання кожного сценарію. Навіть у випадку відмови CI або помилки під час розгортання команда отримує повідомлення з посиланням на журнали, що істотно прискорює реагування. Це позбавляє потреби вручну переглядати історію виконання workflow та підвищує прозорість процесу.

Таким чином, побудований алгоритм забезпечує повний життєвий цикл розгортання проміжних середовищ. Єдина дія, яку виконує розробник – створення гілки. Усі подальші процеси, включаючи розгортання серверів, підготовку середовища, запуск застосунку, формування доменного імені,

очищення ресурсів і моніторинг стану – виконуються автоматично без участі людини. Це дозволяє зменшити час на тестування, мінімізувати людські помилки, знизити витрати і забезпечити масштабованість, практично недоступну у випадку ручної підтримки серверів.

2.3 Особливості апаратної частини у serverless і EC2

Апаратна складова визначає реальну продуктивність CI/CD систем, оскільки саме вона обмежує або прискорює процеси компіляції, тестування, деплою та завантаження залежностей. Ключова відмінність між традиційною серверною інфраструктурою (наприклад, EC2) та serverless-платформами (AWS Lambda) полягає у способі виділення та використання апаратних ресурсів: статично-резервовані проти динамічно-керованих.

2.3.1 Апаратні ресурси в EC2

У випадку EC2 користувач отримує віртуальну машину, яка прив’язана до фізичного хосту AWS. Виділення CPU, RAM і дискової підсистеми відбувається за принципом hardware partitioning або time-sharing залежно від типу середовища. Наприклад, лінійки t3, t4g використовують механізм CPU кредитів – ядра надаються не постійно, а на основі кредитної моделі, що впливає на пікову продуктивність CI/CD завдань. Високопродуктивні лінійки c5, c6 працюють на базі Intel Xeon Scalable або AWS Graviton (ARM-архітектура), що забезпечує високу частоту та розширені інструкції SIMD/AVX, які критичні для компіляторів та інструментів статичного аналізу [19].

Пам’ять у EC2 ізольована та доступна постійно, не підлягає швидкому очищенню між процесами, що спрощує кешування залежностей (npm, composer, docker layers). Дискова підсистема представлена EBS або instance-store: EBS працює як мережевий блоковий пристрій із гарантовною пропускною здатністю (IOPS), а instance-store має низькі затримки, оскільки працює на локальних NVMe-дисках [18]. Для CI/CD це означає стабільний та передбачуваний час читання-запису.

Однак проблема EC2 у тому, що апаратні ресурси залишаються зарезервованими незалежно від інтенсивності використання. Навіть коли сервер просто очікує нового commit або pull request, його процесорні ядра, оперативна пам’ять та дискова підсистема залишаються активними. З інженерної точки зору це означає неефективне використання апаратного фонду:

– ядра простоюють, але виділені;

– оперативна пам’ять зарезервована, навіть якщо не використовується;

– мережеві та дискові ресурси не вимикаються.

Для CI/CD системи, яка створює десятки проміжних середовищ, це призводить до зростання витрат та перевитрати фізичних ресурсів датацентру.

2.3.2 Апаратні ресурси у serverless (AWS Lambda)

Serverless підхід використовує протилежну модель – ресурси виділяються динамічно, тільки на час виконання коду. AWS Lambda працює поверх контейнеризованої інфраструктури AWS і використовує набір ізольованих середовищ. З погляду комп’ютерної інженерії це означає:

– процесорні ядра не закріплені за користувачем;

– функції виконуються на спільній хмарній інфраструктурі;

– фізичний сервер невідомий та недоступний для прямої конфігурації.

Коли Lambda-функція запускається, AWS віртуально виділяє певний обсяг оперативної пам’яті (від 128 МБ до 10 ГБ), і відповідно до цього обсягу автоматично призначає частку процесорного часу. Чим більше оперативної пам’яті, тим більше CPU циклів – ця залежність є ключовою, оскільки CI/CD операції (компіляція, тести, архівація) часто CPU-інтенсивні. Таким чином, масштабування виконується не за кількістю ядер, а за розподілом процесора з обмеженою пам’яттю [14].

З інженерної точки зору, Lambda накладає обмеження на роботу з файловою системою. Функція отримує тимчасовий /tmp сховище (до 10-512 ГБ на сучасних конфігураціях), яке фізично розміщується на локальних NVMe-дисках, але очищається після завершення виконання [14]. Це означає, що CI/CD система повинна використовувати зовнішні сервіси:

– Amazon S3 для артефактів і логів;

– EFS для постійного файлового простору;

– DynamoDB / RDS Proxy для збереження статусів.

У порівнянні з EC2, де файлові дескриптори постійні, у Lambda операційна система й файловий стек ініціалізуються щоразу, що створює додаткові накладні витрати у вигляді холодного старту – часу, необхідного для створення контейнера, завантаження середовища виконання, мережевих підключень і функціонального коду. Під час CI/CD це може проявлятися у збільшенні затримок для перших запусків.

Ще одна особливість – мережеві ресурси Lambda. Функції виконуються у загальній мережевій інфраструктурі AWS, тому throughput і latency залежать від навантаження хмарного сегменту. При роботі у VPC Lambda використовує віртуальні інтерфейси та ENI, що додає кілька мілісекунд при ініціалізації, але дозволяє безпечно підключати приватні ресурси.

У контексті I/O операцій Lambda використовує віртуалізовану дискову систему з обмеженою кількістю файлових дескрипторів, що впливає на паралельне читання, логування і роботу з великими архівами. Для прискорення інтенсивних CI-процесів AWS пропонує механізм Provisioned Concurrency, який утримує функції в попередньо прогрітих контейнерах, зменшуючи холодний старт і покращуючи стабільність I/O [14].

Таким чином, EC2 забезпечує стабільний доступ до апаратних ресурсів із постійною виділеною потужністю процесора, пам’яті та дисків, що робить його передбачуваним, але малоефективним у сценаріях зі змінним навантаженням. Serverless-архітектура навпаки використовує динамічне виділення ресурсів, мінімізує простій і автоматично масштабується, але вимагає адаптації CI/CD систем до обмежень cold start, зовнішніх файлових систем та віртуалізованих мережевих операцій [18].

РОЗДІЛ 3
ЕКСПЕРИМЕНТАЛЬНА ЧАСТИНА

3.1 Методологія порівняння

Порівняння двох підходів до розгортання проміжних середовищ – serverless та EC2 – проводилося на основі практичного експерименту, виконаного в умовах реального командного процесу розробки. У межах цієї роботи було створено повноцінний проєкт – комплексну CI/CD систему, спеціально розроблену для автоматичного формування проміжних середовищ у хмарній інфраструктурі. Детальний процес створення описаний у попередньому розділі. Додатки А, Б, В містять готові лістинги проєкту, який був впроваджений у командні процеси розробки. Ця система не прив’язана до конкретного застосунку та може бути інтегрована у будь-яке програмне забезпечення, незалежно від його складності, оскільки вона використовує універсальні механізми роботи з Git, GitHub Actions, Laravel Vapor та AWS. Створений прототип поєднує у собі ключові складові комп’ютерної інженерії – хмарні технології, системне адміністрування, програмування, інфраструктурну автоматизацію та аналіз апаратних ресурсів. Саме тому розроблена CI/CD система має суттєву практичну цінність: вона дозволяє забезпечити автоматичне, ізольоване та масштабоване розгортання середовищ для будь-яких команд розробки й будь-яких проєктів.

Паралельно з розгортанням serverless-середовищ у системі створювалися також проміжні середовища на основі EC2 екземплярів, щоб забезпечити однакові умови для коректного порівняння двох архітектур. Методологія дослідження була побудована таким чином, щоб результати максимально точно відображали реальну поведінку CI/CD процесів, характерних для сучасних команд, які активно працюють із Git, створюють нові гілки, виконують рев’ю коду та тестують функціональність у паралельних середовищах. Тому в експеримент включалися саме ті параметри, що безпосередньо впливають на швидкість тестування, стабільність процесів та економічну ефективність,

виключаючи фактори, які залишаються незмінними незалежно від обраної архітектури.

У процесі дослідження було визначено набір ключових параметрів порівняння: часові характеристики, доступні апаратні ресурси, фінансові витрати, безпекові аспекти та потреба в людських ресурсах. Ці критерії виявилися визначальними для практичного тестування двох інфраструктурних моделей, оскільки прямо впливали на темп розробки, точність автоматизованих перевірок та загальну вартість підтримки системи. Для забезпечення об’єктивності аналізу було виключено параметри, що не мають прямого відношення до CI/CD, такі як енергоефективність дата-центрів, екологічні фактори чи використання спеціалізованих апаратних прискорювачів, оскільки їхній вплив є сталим для обох архітектур і не змінює результатів експерименту [20].

3.1.1 Параметри порівняння систем

У ході експериментального порівняння першим параметром досліджувався час – ключовий показник продуктивності CI/CD процесу. Під час тестувань фіксувався час підготовки інфраструктури, виконання тестів та повного розгортання середовища для кожної гілки. У serverless-середовищах значний вплив мали холодні старти та тривалість ініціалізації функцій, тоді як на EC2 вирішальним був час підняття віртуальних екземплярів та сервісів у середині них. Зафіксовані значення дозволили оцінити загальний час до готовності середовища для кожного підходу [20].

Другим параметром стали апаратні ресурси – CPU, RAM, пропускна здатність мережі та швидкість дискових операцій. Під час тестування фіксувалися реальні навантаження, характерні для автоматизованих тестів, інтеграційних перевірок та запуску фонових обчислень. На serverless-інфраструктурі ресурси масштабувалися автоматично, що забезпечувало високу гнучкість, але супроводжувалося обмеженням на максимальні параметри одного виконання. На EC2 ресурси були фіксовані, проте

стабільні, що дозволяло досягти прогнозованої продуктивності при високому навантаженні [21].

Третім досліджуваним параметром були фінансові витрати. Під час експерименту вимірювалася фактична вартість виконання CI/CD операцій за двома підходами: постійної роботи EC2 та моделі оплати тільки аза час використання у serverless. У реальних умовах команди, що створюють значну кількість гілок та виконують часті тести, ці відмінності суттєво впливають на витрати, оскільки EC2 вимагає постійно підтримувати інфраструктуру, незалежно від активності, тоді як serverless генерує витрати лише під час виконання робочих процесів.

Четвертим критерієм виступила безпека. У ході дослідження перевірялися відмінності між повністю керованим підходом serverless, де хмарна платформа бере на себе оновлення ОС, оновлення середовища виконання та мережеві аспекти, та EC2, де всі ці процеси контролювала команда. Це дозволило оцінити, наскільки архітектура впливає на складність захисту тестових середовищ та ризики, пов’язані з людським фактором.

П’ятим параметром стали людські ресурси. Під час практичного впровадження було зафіксовано, що serverless знижує потребу в DevOps-фахівцях, оскільки більшість операцій, пов’язаних з управлінням інфраструктурою, виконується автоматично. Для EC2 навпаки потрібні спеціалісти, які регулярно здійснюють контроль серверів, оптимізацію конфігурацій, оновлення та моніторинг збоїв, що впливає на вартість підтримки. Також тестувальники повідомили про відсутність будь яких затримок для налаштувань тестових середовищ.

3.1.2 Моделі тестування

Для коректного порівняння двох підходів до формування середовищ у CI/CD процесі було застосовано дві практичні моделі тестування, які відтворювали роботу реальної команди розробників. Обидві моделі перевірялися на реальних проєктах та за участю декількох розробників, що паралельно оновлювали репозиторій, створювали гілки в Git і запускали CI/CD процеси.

Такий підхід дозволив отримати об’єктивні дані щодо стабільності, швидкодії та зручності кожної моделі в умовах командної роботи.

У першій моделі використовувалася інфраструктура на основі Amazon EC2 із класичною схемою постійних середовищ: dev, qa, stage та production. Під час тестування кожен розробник створював нову гілку у Git, після чого коміт активував CI-процес: збірку застосунку, встановлення залежностей, виконання тестів та деплой у dev-середовище. Подальше тестування проводилось тестувальниками або іншими членами команди у вже існуючих середовищах. Усі дії відбувалися на завчасно підготовленій інфраструктурі, що дозволяло оцінити поведінку системи у стабільних, але спільних для всіх коритсувачів середовищах.

Друга модель передбачала тестування за допомогою тимчасових serverless-середовищ, що створювалися автоматично для кожної гілки через Laravel Vapor. Після push у Git запускалися GitHub Actions, які виконували збірку, тести та ініціювали створення окремого середовища зі своїми Lambda-функціями, Variables, базою даних та S3-сховищем. У ході реального тестування команда отримувала унікальні ізольовані середовища для кожної гілки, що дозволяло проводити перевірку API, користувацького інтерфейсу та автоматизованих тестів без ризику вплинути на роботу інших розробників. Після завершення роботи гілка видалялася, а разом з нею автоматично видалялися всі пов’язані ресурси.

Обидві моделі використовували однаковий підхід до роботи з Git, містили автоматичну збірку та тестування коду, однак принципово відрізнялися організацією середовищ. Перша модель працювала на постійних EC2-серверах, тоді як друга базувалася на динамічних serverless-середовищах, що створювалися й видалялися в режимі «на вимогу». Це дозволило провести реальне порівняння двох підходів з точки зору зручності, гнучкості та відповідності принципам сучасних CI/CD практик.

3.2 Порівняння serverless і EC2 для проміжних середовищ. Переваги і недоліки

Порівняльний аналіз двох підходів – традиційних проміжних середовищ на базі Amazon EC2 і динамічних serverless-середовищ, що створюються через Laravel Vapor та AWS Lambda. Результати тестування на однаковому проєкті дали змогу зафіксувати об'єктивні вимірювання, які значно відрізняються залежно від архітектурного підходу.

3.2.1 Аналіз за параметром «Час»

Тривалість розгортання стала найпоказовішою різницею між підходами. У традиційній моделі, де всі середовища розгорнуті на EC2 постійно, час очікування був мінімальним – застосунок уже працював, і перевірка починалася одразу після деплою. Проте деплой у готове середовище часто затягувався через черги у CI-CD, одночасне оновлення іншими розробниками, наявність конфліктів або оновлення залежностей. В середньому оновлення dev-середовища тривало 3-5 хвилин, а stage – 7-12 хвилин в залежності від масштабності зміни.

У serverless-підході час включає збірку, розгортання і створення нового середовища. На практиці середній час розгортання нового ізольованого середовища становив від 40 секунд до 2 хвилин. Повне видалення після злиття гілки займало до 20-40 секунд. Попри те, що час створення середовища більший, ніж миттєвий доступ до статичного EC2, розробники отримували повністю чистий ізольований простір, що унеможливлювало конфлікти між гілками. Графік порівняння середнього часу розгортання зображений на рисунку 3.1. На якому видно що використання serverless підходу значно зменчує витрати ресурсів на виконання а також на очікування готовності середовищ іншими учасниками процесу розробки. Додатково було зафіксовано, що ізольованість кожного середовища у serverless-моделі суттєво прискорювала процес рев'ю та тестування, оскільки усувала залежність між паралельними змінами та зменшувала кількість повторних перевірок через побічні ефекти.

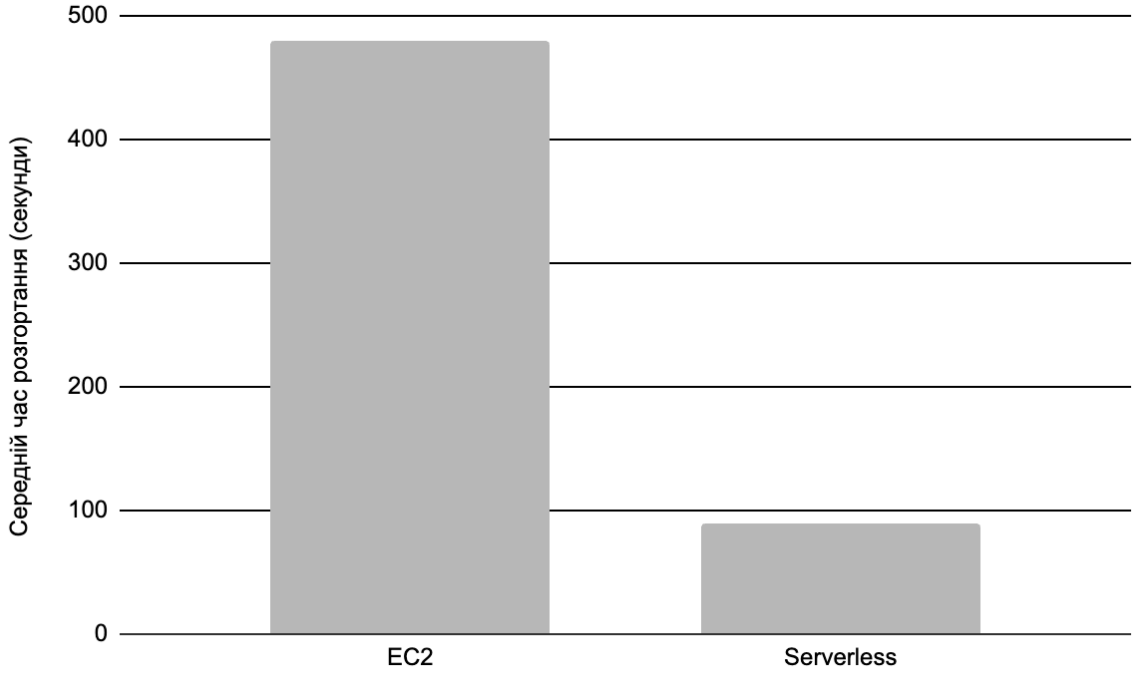

Рисунок 3.1 – Порівняння середнього часу до готовності середовища для EC2 і Serverless

З точки зору затраченого часу для тестування serverless-модель виявилася швидшою у командній роботі при наявності багатьох паралельних задач. Якщо у трьох розробників одночасно надходили зміни, EC2-сервери створювали чергу деплою і тестування, тоді як Lambda дозволяла виконувати розгортання паралельно, без конкуренції за обчислювальні ресурси. Результати порівняння наведені у таблиці 3.1.

Таблиця 3.1 – Часові показники середовищ

Підхід	Середній час до готового тестування	Паралельність	Конфлікти між гілками
EC2	3 – 12 хвилин	Обмежена	Часті
Serverless	40 – 120 секунд	Необмежена	Відсутні

3.2.2 Аналіз за параметром «Апаратні ресурси»

У тестуванні EC2-підхід використовував виділені екземпляри із фіксованою кількістю vCPU, оперативної пам’яті та дискових каналів. Це означає, що навіть у періоди простою середовище споживало ресурси, а під час пікових навантажень швидкість масштабування залежала від вертикального або

горизонтального масштабування EC2-кластеру. Крім того, декілька тестових гілок могли ділити одні й ті самі апаратні ресурси, що спричиняло падіння продуктивності або уповільнення роботи CI-процесів. На рисунку 3.2 детально зображені графіки завантаження центрального процесору із збільшенням кількості паралельних гілок.

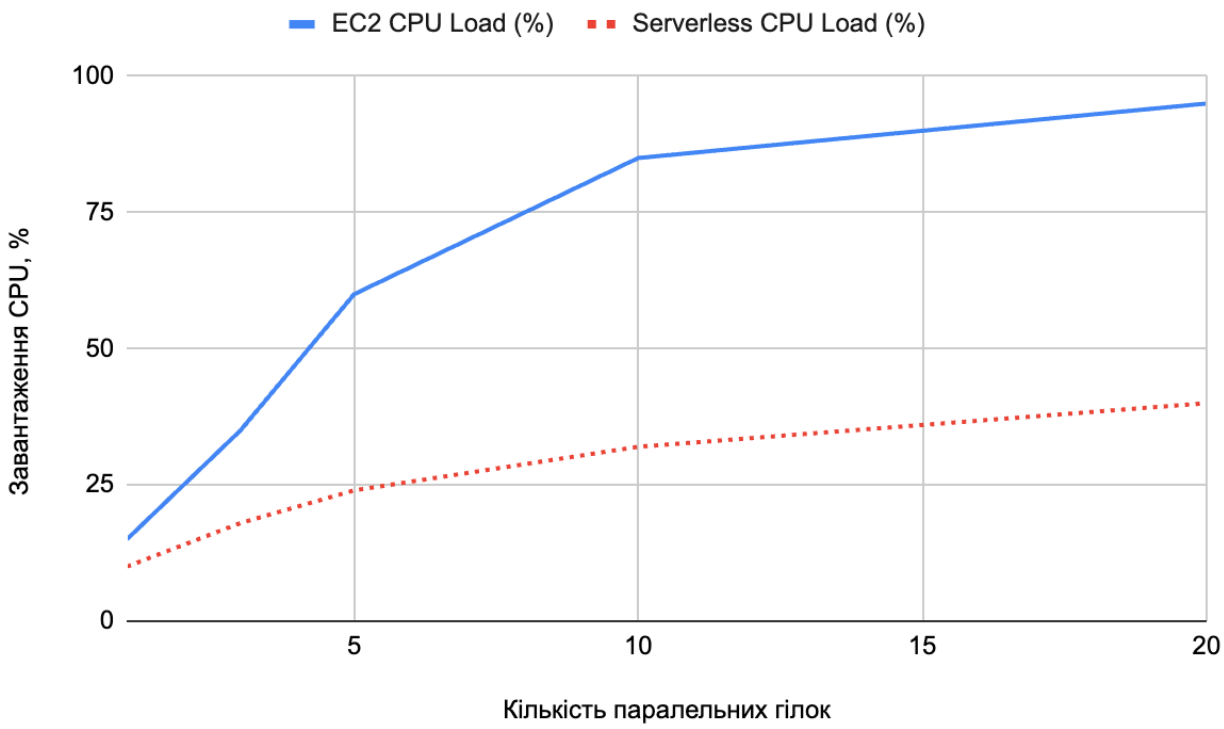

Рисунок 3.2 – Використання ресурсів центрального процесора

У serverless-підході продуктивність визначається параметрами AWS Lambda. Вибір конфігурації дозволив встановити обсяг пам’яті (від 128 МБ до 10 ГБ), кількість CPU, а також дискові операції, що виконувались у тимчасовому середовищі. Кожен запит оброблявся незалежною інстанцією функції, тому навіть десять розробників могли запускати тестування одночасно без взаємного впливу на швидкість роботи. Завантаження CPU та I/O розподілялися автоматично, без ручного масштабування та без ризику переповнення ресурсів. При збільшенні одночасно піднятих середовищ кількість оперативної пам’яті зростала для EC2 екземплярів, так як для кожного екземпляру резервувались

додаткові ресурси. Водночас обсяг для serverless підходу фактично не змінювався. Це пов’язано з принципами роботи Lambda-функцій. Графік порівняння використання ресурсів оперативної пам’яті зображений на рисунку 3.3.

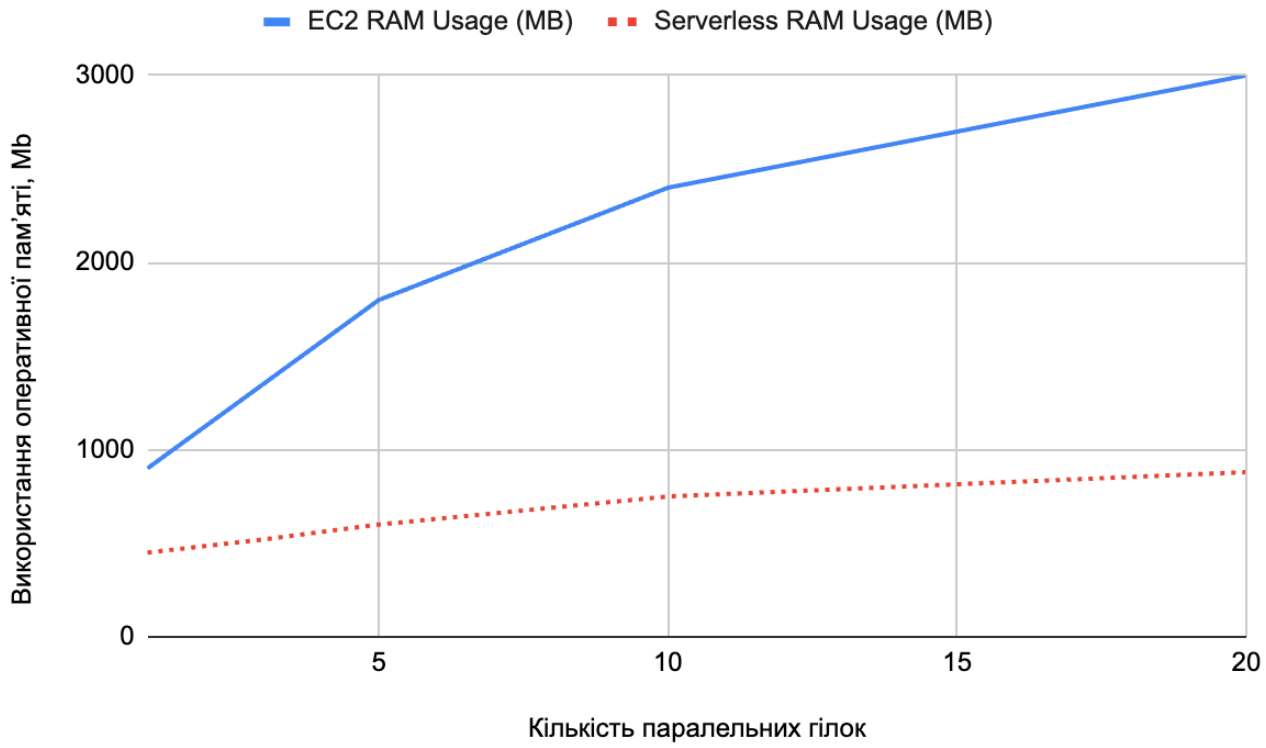

Рисунок 3.3 – Використання оперативної пам’яті

3.2.3 Аналіз за параметром «Фінансові витрати»

У традиційній конфігурації EC2 середовища працювали постійно та вимагали цілодобового утримання незалежно від активності команди. Це означало стабільні щомісячні витрати на CPU, RAM, дискову підсистему, мережевий трафік та балансування. Навіть у моменти, коли жоден розробник не запускав тести чи не виконував деплой, EC2-інстанси залишалися активними і генерували однакові витрати. У результаті місячна вартість утримання dev, qa та stage середовищ була передбачуваною, але не оптимальною, оскільки не враховувала реального навантаження. Фінансову модель роботи EC2 можна подати у вигляді формули (3.1):

$$C_{EC2} = \sum_{i=1}^{n} \left(C_{instance}^{(i)} \cdot H_{month} + C_{storage}^{(i)} + C_{network}^{(i)} \right), \quad (3.1)$$

де n – кількість середовищ (dev, qa, stage);

$C_{instance}^{(i)}$ – вартість 1 години роботи EC2 екземпляру;

$H_{month} = 24 \cdot 30$ – кількість годин у місяці;

$C_{storage}^{(i)}$ – вартість EBS-дисків для середовища;

$C_{network}^{(i)}$ – вартість мережевого трафіку.

Ця формула показує, що витрати зростають із кількістю середовищ, але не залежать від частоти виконання тестів чи кількості гілок.

На противагу цьому, у serverless-підході витрати виникали лише під час фактичної роботи CI/CD процесу: створення середовища, виконання Lambda-функцій, звернення до бази даних та операцій зі сховищем S3. Якщо нові гілки не створювалися, а тестування не проводилося, витрати зменшувалися майже до нуля. Вартість виконання функцій Lambda описується у вигляді формули (3.2):

$$C_{Lambda} = N_{req} \cdot C_{req} + \left(\frac{T_{exec}}{1000} \cdot R_{GBs} \cdot C_{GBs} \right), \quad (3.2)$$

де N_{req} – кількість запусків Lambda-функцій;

C_{req} – вартість 1 запиту;

T_{exec} – середня тривалість виконання функції (мс);

$R_{GBs} = \frac{Memory\_MB}{1024}$ – виділена пам’ять у GB;

C_{GBs} – ціна за 1 GB/сек.

Із цих формул видно, що витрати масштабуються лише в моменти, коли система реально виконує роботу, а отже є динамічними та пропорційними навантаженню. Під час реального використання у команді з 5-10 гілками на день

сукупні витрати serverless-рішення виявилися нижчими, ніж у випадку EC2, насамперед за рахунок відсутності постійно активних серверів. Таким чином, serverless-підхід демонструє значно кращу економічну ефективність у сценаріях з нерівномірним або піковим навантаженням. Яка залежить від методів використання підходів. Порівняльні результати фінансових моделей наведені у таблиці 3.2.

Таблиця 3.2 – Фінансова модель

Параметр	EC2	Serverless
Оплата за простій	Є	Немає
Оплата за піки навантаження	Потрібно масштабувати	Оплата за фактичні виконання
Середня вартість місяця при низьких навантаженнях	Висока	Мінімальна
Середня вартість місяця при високих навантаженнях	Зростає	Зростає повільно

На рисунку 3.4 зображений графік витрат на апаратні ресурси в залежності від підходу.

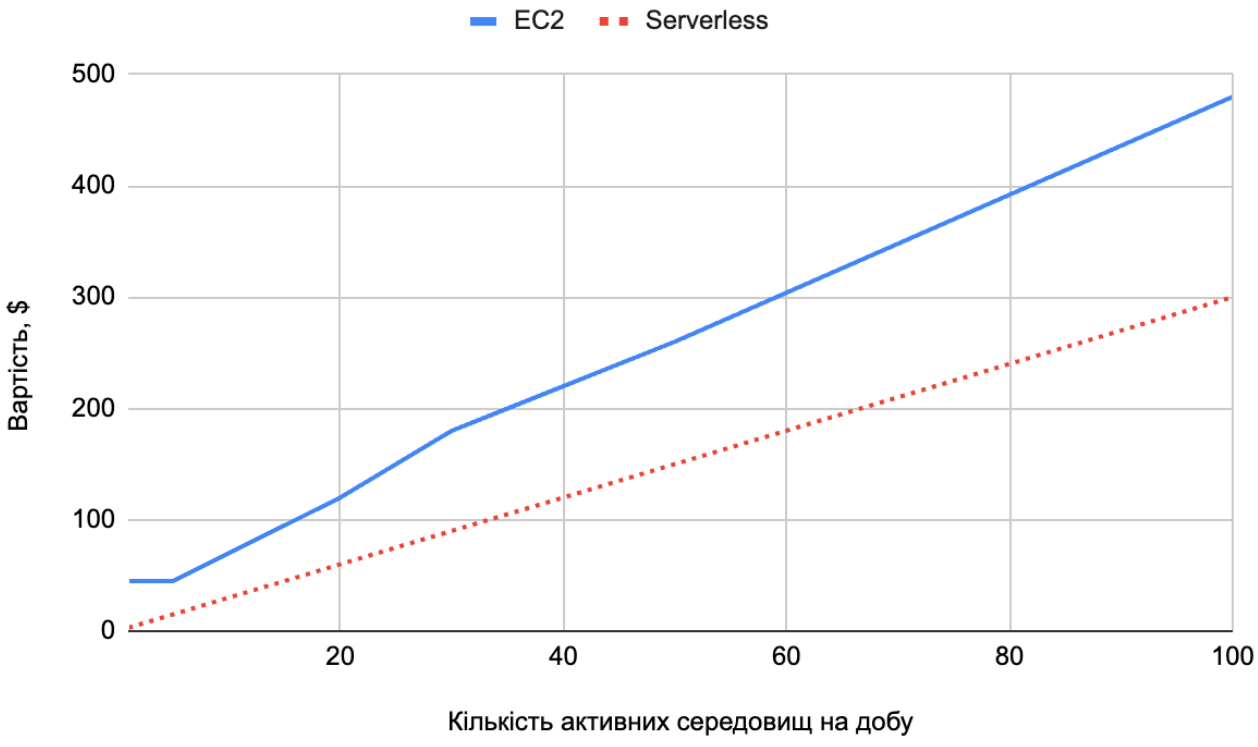

Рисунок 3.4 – Графік порівняння витрат в залежності від типу підходу

3.2.4 Аналіз за параметром «Безпека»

Безпека традиційних EC2-середовищ залежала від того, наскільки якісно налаштовані мережі, доступи, оновлення та захист ОС. Оскільки сервери існують постійно, вони можуть стати потенційною ціллю для атак, а неправильне керування ключами або мережевими правилами здатне призвести до компрометації середовища.

У serverless-підході поверхня атаки зменшується, бо середовище не існує постійно. Доступ до Lambda-функцій і сховищ контролюється IAM-ролями, а конфіденційні дані передаються через зашифровані змінні середовища. Після видалення гілки знищується і середовище, тому зникає й можливість зловживання вразливостями, які могли бути залишені в процесі тестування.

Обслуговування EC2-інфраструктури вимагає постійної участі DevOps-інженера: оновлення ОС, конфігурації балансувальників, моніторинг продуктивності, журналів та стану серверів. При збільшенні кількості гілок потрібне ручне масштабування або оркестрація контейнерів.

Serverless підхід практично не потребує підтримки. Уся інфраструктура створюється та знищується автоматично, а за станом комп’ютерних ресурсів слідкують сервіси AWS. Це дає змогу зменшити час, витрачений DevOps-фахівцем, і перекласти обслуговування інфраструктури на платформу.

3.3 Оцінка продуктивності, використання ресурсів та фінансових витрат

Проведене дослідження дозволило оцінити поведінку двох різних архітектурних підходів до організації проміжних середовищ розробки: традиційної моделі, що базується на EC2, та serverless підходу з автоматичним створенням середовищ на вимогу. Порівняння проводилось на основі узгоджених параметрів, серед яких найвагомішими були час підготовки середовища, ступінь використання обчислювальних ресурсів, продуктивність під час паралельної роботи декількох гілок розробки та фінансові витрати, пов’язані з утриманням інфраструктури. Для наочності отриманих результатів

було побудовано діаграму за основними параметрами продуктивності та споживання ресурсів (рисунок 3.5).

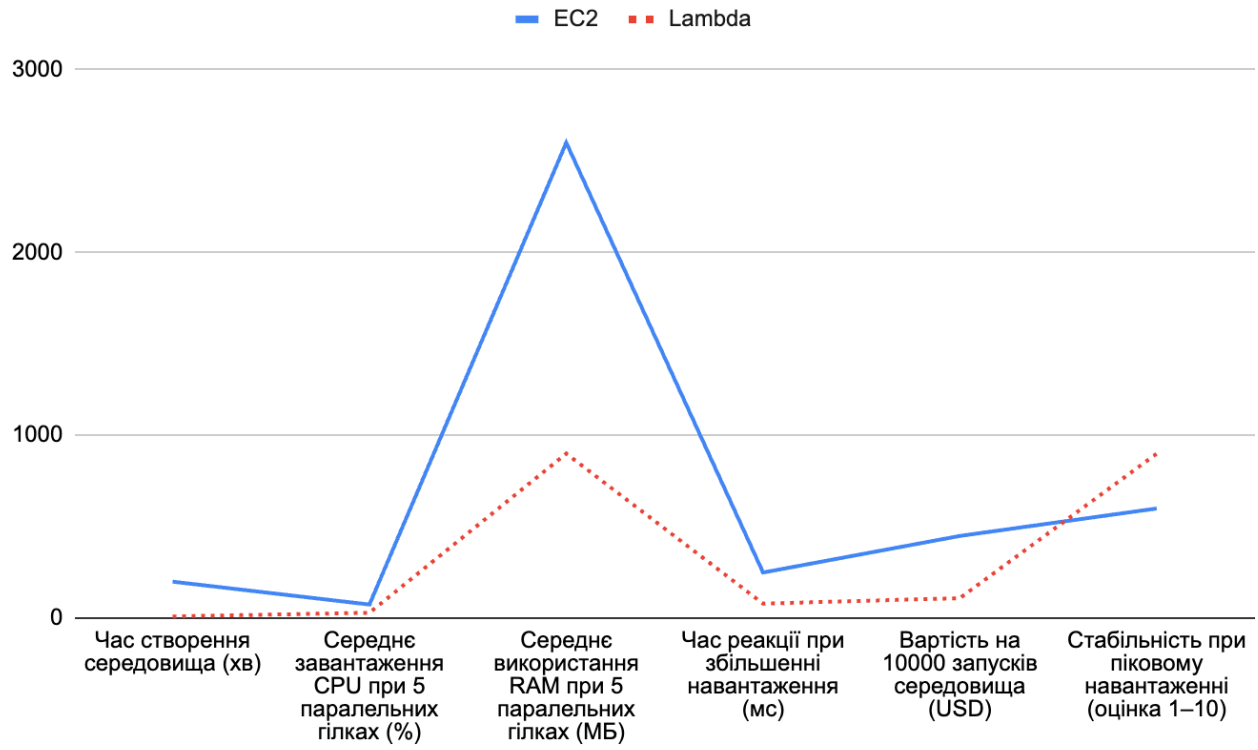

Рисунок 3.5 – Графік порівняння систем за обраними параметрами

Результати вимірювань засвідчили суттєву різницю у часі готовності середовищ. У випадку EC2 створення повноцінного окремого середовища займало від десяти до двадцяти хвилин залежно від об’єму застосунку та розміру використовуваних образів. У serverless-рішення цей показник зменшувався до приблизно однієї хвилини, що вказує на здатність виконувати робочі процеси майже миттєво після створення гілки у Git. Така різниця безпосередньо впливає на швидкість розробки: кожен розробник отримує середовище значно швидше, що зменшує час очікування та кількість заблокованих задач у команді.

Продуктивність системи також продемонструвала переваги serverless-архітектури. EC2-середовище потребує попереднього розподілення ресурсів, тому навіть невелика кількість паралельних гілок призводить до високого навантаження процесора та збільшення споживання пам’яті. У

serverless-інфраструктурі навантаження розподіляється динамічно, а ресурси виділяються під кожен окремий виклик, тому збільшення кількості середовищ не спричиняє різких піків споживання. Аналіз таблиці, використаної для побудови рисунку 3.2, підтверджує, що при зростанні кількості паралельних середовищ EC2 майже досягає граничних значень, тоді як serverless зберігає стабільну роботу та помірне використання апаратних ресурсів. Така поведінка свідчить про краще масштабування та можливість безпечного паралельного тестування великої кількості гілок.

Фінансовий аспект також демонструє суттєву різницю. EC2-підхід вимагає постійного утримання серверів навіть у періоди, коли активність команди мінімальна, тому оплата здійснюється за доступність інфраструктури, а не за її фактичне використання. Навпаки, serverless забезпечує оплату відповідно до реальної активності розробників, що особливо помітно при використанні підходу Preview Environments. У період інтенсивної розробки витрати зростають, але вони все одно залишаються нижчими, ніж вартість постійного утримання розгорнутих EC2-екземплярів. У періоди простою витрати майже зникають, що робить serverless більш гнучким і передбачуваним рішенням.

Оцінка цих параметрів показала, що переваги serverless значно впливають на ефективність командної роботи. Зменшення часу очікування середовищ дає можливість виконувати цикл розробки, тестування та виправлення помилок швидше. Автоматичне масштабування виключає ситуації, у яких велика кількість гілок викликає деградацію продуктивності основних сервісів. Водночас EC2-підхід показує себе стійкішим у контексті повного контролю над інфраструктурою та можливості ручного налаштування конфігурацій на низькому рівні, однак ці переваги мають ціну у вигляді затримок, перевитрат ресурсів та вищої фінансової складності.

Загальна оцінка демонструє, що з точки зору продуктивності, використання апаратних ресурсів і фінансової ефективності serverless-архітектура є більш придатною для побудови проміжних середовищ у сучасних CI/CD-процесах. Вона дозволяє масштабувати середовища відповідно

до навантаження, виключає потребу у витратах на постійно працюючі сервери та забезпечує швидкий цикл розробки. EC2-підхід залишається доцільним для систем зі специфічними вимогами до ручного контролю або складними процесами розгортання, проте у контексті командної розробки і частих створень тимчасових середовищ його переваги стають менш актуальними. Таким чином, результати дослідження підтвердили доцільність переходу до serverless-моделі для оптимізації процесу розробки та зменшення витрат на інфраструктуру.

ВИСНОВКИ

У ході дослідження було здійснено аналіз сучасних підходів до побудови CI/CD систем, включно з моделями на основі віртуальних машин, контейнеризації та serverless-архітектур. Було порівняно їхні переваги й недоліки щодо використання обчислювальних ресурсів, масштабованості, вартості та гнучкості. Особливу увагу приділено тому, як кожен підхід впливає на час виконання пайплайнів, ступінь ізоляції середовищ та можливість командної роботи. На основі цього аналізу сформовано узагальнену оцінку ефективності моделей у контексті сучасних вимог до CI/CD процесів.

У межах практичної частини роботи було реалізовано прототип CI/CD системи, що використовує serverless-архітектуру для автоматичного створення проміжних тестових середовищ. На основі інтеграції GitHub Actions та платформи Laravel Vapor було розроблено повністю автоматизований процес розгортання ізольованих середовищ для кожної гілки, включно з конфігурацією Lambda-функцій, бази даних, файлового сховища та змінних середовища. Прототип був успішно протестований у реальних умовах командної розробки, що підтвердило його працездатність та відповідність моделі «on demand environments».

Також було досліджено вплив різних типів апаратних ресурсів на швидкість і стабільність роботи CI/CD процесів. Практичні тести проводилися на інфраструктурі Amazon EC2 та у serverless-середовищах, що дозволило порівняти поведінку системи за умов варіацій CPU, обсягу оперативної пам’яті, продуктивності дискової підсистеми та мережевих характеристик. У результаті отримано кількісні та якісні спостереження, які відображають залежність часу виконання етапів збірки, тестування та деплою від характеристик обчислювальних ресурсів, а також визначено фактори, що найбільше впливають на стабільність CI/CD процесів.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Войтович М. CI/CD система з використанням serverless архітектури. *Цифрова трансформація: виклики та стратегії* : зб. тез доп. міжнар. наук.-практ. конф., м. Луцьк, 25 лют. 2025 р. Луцьк, 2025. С. 154-155.

2. Driessen V. A successful Git branching model. *nvie.com*. URL: https://nvie.com/posts/a-successful-git-branching-model/ (date of access: 20.07.2025).

3. Loeliger J., McCullough M. *Version Control with Git*. 3rd ed. Sebastopol : O'Reilly Media, 2021. 454 p.

4. GitHub flow - GitHub Docs. *GitHub Docs*. URL: https://docs.github.com/en/get-started/quickstart/github-flow (date of access: 24.07.2025).

5. What is GitLab Flow?. *about.gitlab.com*. URL: https://about.gitlab.com/topics/
version-control/what-is-gitlab-flow/ (date of access: 25.07.2025).

6. Fowler M. Continuous Integration. *martinfowler.com*. URL: https://martinfow
ler.com/articles/continuousIntegration.html (date of access: 08.08.2025).

7. Adrian L. Rapid application development with AWS amplify: build cloud-native mobile and web apps from scratch through continuous delivery and test automation. Packt Publishing, 2021. 344 p.

8. Гармаш Д. В. *Сучасні сервери на AWS Lambda функції* : дипломна робота бакалавра. Київ : КПІ ім. Ігоря Сікорського, 2024. 65 с.

9. Choudhury T. Autonomic computing in cloud resource management in industry 4.0. Springer International Publishing AG, 2021. 411 p.

10. Єршова Ю. В. *Метод та засіб побудови веб-сервісу на основі Serverless обчислень* : магістерська дис. Київ : КПІ ім. Ігоря Сікорського, 2021. С. 41-42.

11. Barrak A., Ksontini E., Atike R., Jaafar F. *FaaSGuard: Secure CI/CD for Serverless Applications.* URL: https://arxiv.org/pdf/2509.04328 (date of access: 11.09.2025)

12. Introduction – Laravel Vapor. *Introduction – Laravel Vapor*. URL: https://docs.vapor.build/introduction (date of access: 15.09.2025).

13. What is Amazon CloudFront? - Amazon CloudFront. URL: https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html (date of access: 22.09.2025).

14. Lambda runtimes - AWS Lambda. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html (date of access: 22.09.2025).

15. What is Amazon API Gateway? - Amazon API Gateway. URL: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html (date of access: 01.10.2025).

16. Getting started with CloudWatch Logs - Amazon CloudWatch Logs. URL: https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_GettingStarted.html (date of access: 08.10.2025).

17. GitHub Actions documentation - GitHub Docs. *GitHub Docs*. URL: https://docs.github.com/en/actions (date of access: 20.10.2025).

18. AWS Cloudfornation. URL: https://docs.aws.amazon.com/cloudformation/ (date of access: 22.10.2025).

19. AWS EC2. URL: https://docs.aws.amazon.com/ec2/ (date of access: 23.10.2025).

20. Sachin S. DevOps on Google Cloud Platform: Covering Docker, Jenkins, Kubernetes : Independently Published, 2021. 292 p.

21. Lambda quotas - AWS Lambda. URL: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html (date of access: 08.11.2025).