Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Дослідження хмарних архітектур підтримки сервісів громадської взаємодії на базі сучасних фреймворків

Виконав: студент VI курсу, групи СНм-61
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)

_____ Лісовий М.В.
(підпис) (прізвище та ініціали)

Керівник _____ Никитюк В.В.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Дуда О.М.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Боднарчук І.О.
(підпис) (прізвище та ініціали)

Рецензент _____ Стадник М.А.
(підпис) (прізвище та ініціали)

Тернопіль
2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
« 17 » листопада 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)

Студенту Лісовому Максиму Володимировичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження хмарних архітектур підтримки сервісів громадської взаємодії на базі сучасних фреймворків

Керівник роботи Никитюк Вячеслав Вячеславович, к.т.н., доцент кафедри КН
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від « 27 » листопада 2025 року № 4/7-1042

2. Термін подання студентом завершеної роботи 22 грудня 2025 р.

3. Вихідні дані до роботи Наукові публікації про хмарні обчислення, мікросервісну архітектуру та технології Civic Tech для проєктування систем збору та аналізу громадської взаємодії; технічна документація Laravel, середовища Docker та протоколів WebSockets

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1 Аналіз предметної області та теоретичні засади побудови сервісів громадської взаємодії. 2 Проєктування та програмна реалізація прототипу мікросервісної системи. 3 Аналіз результатів дослідження та перспективи розвитку системи. 4 Охорона праці та безпека в надзвичайних ситуаціях. Висновки. Додатки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1 Титульна сторінка. 2 Тема, Мета, Об'єкт, Предмет дослідження. 3 Завдання дослідження. 4 Актуальність дослідження та наукова новизна. 5 Класифікація та функціональні вимоги до платформ громадської взаємодії. 6 Порівняльний аналіз архітектурних патернів (Моноліт vs Мікросервіси). 7 Загальна архітектура розробленої системи (Компонентна діаграма). 8 Схема даних та API-контракти мікросервісів. 9 Схема інфраструктури розгортання. 10 Алгоритм асинхронної обробки подій. Sequence Diagram. 11 Реалізація оновлень у реальному часі. 12 Результати тестування ключових сценаріїв користувача. 13 Аналіз потенціалу масштабування системи. 14 Оцінка безпеки та надійності архітектури. 15 Рекомендації щодо розгортання (CI/CD) та моніторингу. 16 Висновки. 17 Завершальний слайд.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Гурик О. Я., доцент кафедри МТ		
Безпека в надзвичайних ситуаціях	Теслюк В.М., проректор з адміністративно-господарської роботи та будівництва		

7. Дата видачі завдання 17 листопада 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	17.11.2025	*Виконано*
2.	Аналіз науково-технічних публікацій та збір даних по	17.11.2025-24.11.2025	*Виконано*
	темі кваліфікаційної роботи		
3.	Виконання дослідження згідно мети кваліфікаційної	25.11.2025-08.12.2025	*Виконано*
	роботи		
4.	Оформлення розділу «Аналіз предметної області та теоретичні засади побудови сервісів громадської взаємодії»	09.12.2025-11.12.2025	*Виконано*
5.	Оформлення розділу «Проєктування та програмна реалізація прототипу мікросервісної системи»	09.12.2025-11.12.2025	*Виконано*
6.	Оформлення розділу «Аналіз результатів дослідження та перспективи розвитку системи»	09.12.2025-11.12.2025	*Виконано*
7.	Виконання завдання до підрозділу «Охорона праці»	25.11.2025-08.12.2025	*Виконано*
8.	Виконання завдання до підрозділу «Безпека в надзвичайних ситуаціях»	25.11.2025-08.12.2025	*Виконано*
9.	Оформлення кваліфікаційної роботи	12.12.2025	*Виконано*
10.	Нормоконтроль	15.12.2025-16.12.2025	*Виконано*
11.	Перевірка на плагіат	17.12.2025	*Виконано*
12.	Попередній захист кваліфікаційної роботи	18.12.2025	*Виконано*
13.	Захист кваліфікаційної роботи	22.12.2025	

Студент _____ Лісовий М.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Никитюк В.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дослідження хмарних архітектур підтримки цифрових сервісів громадської взаємодії на базі сучасних фреймворків // Кваліфікаційна робота освітнього ступеня «Магістр» // Лісовий Максим Володимирович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СНм-61 // Тернопіль, 2025 // С. 65, рис. – 6, табл. – 6, кресл. – 17, додат. – 3, бібліогр. – 64.

Ключові слова: хмарні архітектури, мікросервіси, civic tech, laravel, docker, масштабованість, асинхронна обробка, websockets.

Кваліфікаційна робота присвячена розробці архітектури масштабованих платформ громадської взаємодії на основі мікросервісного підходу.

В першому розділі описано засади Civic Tech та вимоги до високонавантажених систем. Проаналізовано хмарні моделі, контейнеризацію та архітектурні патерни. Обґрунтовано перехід до мікросервісів та вибір фреймворку Laravel.

В другому розділі спроєктовано та реалізовано прототип системи. Розроблено схему взаємодії сервісів та їх API. Впроваджено механізми асинхронної обробки даних та оновлень у реальному часі.

В третьому розділі протестовано прототип та верифіковано роботу компонентів. Оцінено потенціал масштабування, рівень ізоляції даних та відмовостійкості. Надано рекомендації щодо автоматизованого розгортання та моніторингу.

Об'єкт дослідження: процеси побудови та масштабування високонавантажених веб-систем.

Предмет дослідження: проєктування мікросервісної архітектури Civic Tech платформ на базі Laravel та хмарних технологій.

ANNOTATION

Research into Cloud Architectures Supporting Digital Public Interaction Services Based on Modern Frameworks// The educational level "Master" qualification work // Lisovyi Maksym // Ternopil Ivan Pulyuy National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Computer Science, SNm-61 group // Ternopil, 2025 // P. 65, fig. – 6, tables – 6, posters – 17, annexes – 4, ref. – 64.

Key words: cloud architectures, microservices, civic tech, laravel, docker, scalability, asynchronous processing, websockets.

Thesis is devoted to the development of an architectural solution for building scalable civic engagement platforms using a microservices approach.

In the first section, the principles of Civic Tech and requirements for high-load systems are described. Cloud models, containerization, and architectural patterns are analyzed. The transition to microservices and the choice of the Laravel framework are justified.

In the second section, the system prototype is designed and implemented. The interaction scheme of services and their APIs are developed. Mechanisms for asynchronous data processing and real-time updates are implemented.

In the third section, the prototype is tested, and component interaction is verified. Scaling potential, data isolation, and fault tolerance levels are assessed. Recommendations for automated deployment and monitoring are provided.

Object of study: processes of building and scaling high-load web systems.

Subject of study: design of microservice architecture for Civic Tech platforms based on Laravel and cloud technologies.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API (англ. Application Programming Interface) – Програмний інтерфейс додатку.

CI/CD (англ. Continuous Integration / Continuous Delivery) – Безперервна інтеграція та безперервна доставка.

Civic Tech (англ. Civic Technology) – Громадянські технології.

Docker – Платформа для розробки, доставки та запуску контейнеризованих додатків.

HTTP (англ. HyperText Transfer Protocol) – Протокол передачі гіпертексту.

IaaS (англ. Infrastructure-as-a-Service) – Інфраструктура як послуга.

JSON (англ. JavaScript Object Notation) – Текстовий формат обміну даними.

JWT (англ. JSON Web Token) – Веб-токен JSON.

ORM (англ. Object-Relational Mapping) – Об'єктно-реляційне відображення.

PaaS (англ. Platform-as-a-Service) – Платформа як послуга.

REST (англ. Representational State Transfer) – Архітектурний стиль взаємодії компонентів розподіленого додатка.

ЗМІСТ

ВСТУП

Актуальність теми. У сучасному світі цифрова трансформація державного управління та громадської взаємодії (Civic Tech) стає критично важливою для розвитку демократичного суспільства. Сервіси електронних петицій, громадських бюджетів та онлайн-консультацій дозволяють громадянам впливати на прийняття рішень. Проте специфіка таких систем полягає у непередбачуваному, «вірусному» характері навантаження, коли кількість запитів може зростати в тисячі разів за короткий проміжок часу. Традиційні монолітні архітектури не здатні забезпечити необхідну гнучкість та стійкість у таких умовах. Тому розроблення та дослідження еластичних хмарних архітектур на базі мікросервісного підходу з використанням сучасних фреймворків є актуальним напрямком наукових досліджень.

Мета і задачі дослідження. Метою даної кваліфікаційної роботи рівня «Магістр» є підвищення ефективності, масштабованості та надійності платформ громадської взаємодії шляхом розробки архітектурного рішення на основі мікросервісів, асинхронної обробки даних та технологій реального часу.

Для досягнення поставленої мети потрібно виконати ряд завдань, зокрема:

- Проаналізувати стан досліджень в області Civic Tech та визначити вимоги до високонавантажених систем.
- Дослідити існуючі архітектурні патерни (моноліт, SOA, мікросервіси) та хмарні моделі розгортання.
- Обґрунтувати вибір технологічного стеку (Laravel, Docker) для реалізації розподіленої системи.
- Розробити компонентну архітектуру та програмний прототип системи, що включає сервіси автентифікації, обговорень та сповіщень.
- Реалізувати механізми асинхронної взаємодії сервісів та оновлень інтерфейсу в реальному часі.
- Протестувати працездатність прототипу та оцінити потенціал його масштабування.

Об’єкт дослідження: процеси побудови та масштабування високонавантажених веб-систем.

Предмет дослідження: проєктування мікросервісної архітектури Civic Tech платформ на базі Laravel та хмарних технологій.

Наукова новизна одержаних результатів кваліфікаційної роботи полягає у вирішенні науково-прикладної задачі забезпечення еластичної масштабованості Civic Tech платформ шляхом адаптації мікросервісного патерну до екосистеми Laravel, а також у комплексному застосуванні асинхронних черг та WebSocket-технологій для оптимізації продуктивності розподілених систем.

Практичне значення одержаних результатів. Виконано проєктування та програмну реалізацію робочого прототипу мікросервісної платформи, налаштовано середовище контейнеризації (Docker) для імітації хмарної інфраструктури, що дозволяє використовувати отримані рішення для розгортання реальних сервісів громадської взаємодії.

Апробація результатів магістерської роботи. Основні результати проведених досліджень обговорювались на XIV Міжнародній науково-практичній конференції молодих учених та студентів «Актуальні задачі сучасних технологій» та XIII науково-технічній конференції «Інформаційні моделі, системи та технології» Тернопільського національного технічного університету імені Івана Пулюя (м. Тернопіль, 2025 р.).

Публікації. Основні результати кваліфікаційної роботи опубліковано у двох працях конференції (Див. додатки А).

Структура й обсяг кваліфікаційної роботи. Кваліфікаційна робота складається зі вступу, трьох основних розділів, розділу з охорони праці, висновків, списку використаних джерел з 64 найменувань та 4 додатків. Загальний обсяг кваліфікаційної роботи складає 65 сторінок, з них 47 сторінок основного тексту, який містить 6 рисунків та 6 таблиць.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ТЕОРЕТИЧНІ ЗАСАДИ ПОБУДОВИ СЕРВІСІВ ГРОМАДСЬКОЇ ВЗАЄМОДІЇ

1.1 Концепція цифрової громадської взаємодії: форми, цілі та соціальні виклики

Цифрова трансформація фундаментально реконструює ландшафт суспільних відносин, особливо у сфері взаємодії між громадянами, громадянським суспільством та державними інституціями. Цей процес, що виходить далеко за межі простої автоматизації бюрократичних процедур, формує нову парадигму цифрової громадської взаємодії (digital civic engagement). Він охоплює екосистему технологій, платформ та практик, що мають на меті не лише підвищення ефективності державного управління, але й поглиблення демократичних процесів. Розуміння соціальної сутності, цілей та іманентних викликів цієї концепції є критично важливим для подальшого аналізу технологічних архітектур, що покликані її підтримувати.

Центральними поняттями, що структурують поле цифрової взаємодії, є «електронне урядування» (E-governance) та «громадянські технології» (Civic Tech). Хоча ці терміни часто використовуються як взаємозамінні, вони описують різні, хоча й взаємопов’язані аспекти.

Електронне урядування переважно стосується ініціатив «згори-донизу» (top-down), де державні органи використовують інформаційно-комунікаційні технології для оптимізації внутрішніх процесів та надання публічних послуг громадянам і бізнесу [1]. Згідно з дослідженням ООН (2024), акцент робиться на інвестиціях у цифрову інфраструктуру для «покращення надання державних послуг та сприяння інклюзивним і стійким суспільствам» [2]. Отже, ключова мета E-governance – це ефективність, оперативність та доступність сервісів.

Громадянські технології (Civic Tech), навпаки, частіше розвиваються «знизу-вгору» (bottom-up) ініціативами громадянського суспільства, громадських організацій, активістів та стартапів [3]. Вони визначаються як технології, що мають на меті посилення участі громадян, сприяння прозорості та

підзвітності урядування. Якщо E-governance фокусується на наданні послуг, то Civic Tech – на поглибленні демократії та посиленні впливу громадян на процеси прийняття рішень [4].

Цілі, що їх переслідує розвиток цифрової громадської взаємодії, є багатовимірними. По-перше, це підвищення прозорості та підзвітності влади. Відкриття державних даних, цифровізація закупівель та публікація бюджетів у машиночитних форматах дозволяють громадськості здійснювати ефективний контроль за діями інституцій. По-друге, це пряме залучення громадян до прийняття рішень. Такі інструменти, як електронні петиції, платформи громадських бюджетів чи онлайн-консультації, трансформують громадян із пасивних отримувачів послуг на активних співтворців політики. По-третє, це розвиток місцевих спільнот та спільне вирішення проблем [5]. Цифрові платформи слугують майданчиками для самоорганізації громадян, ідентифікації локальних проблем та колаборації з муніципалітетами для їх вирішення.

Незважаючи на значний потенціал, реалізація концепції цифрової громадської взаємодії стикається з низкою гострих соціальних викликів, що можуть нівелювати її переваги.

- Цифрова нерівність. Це ключова соціальна перешкода. Доступ до високошвидкісного інтернету, наявність необхідних пристроїв та рівень цифрової грамотності розподілені вкрай нерівномірно. Дослідження Eurofound (2025) підкреслює, що вразливі групи населення, зокрема, люди старшого віку, особи з низьким рівнем освіти та доходу мають найбільші труднощі з доступом до електронних послуг [6]. Це створює ризик, що цифровізація не подолає, а, навпаки, посилить існуючі форми соціальної ексклюзії.

- Ризики дезінформації та маніпуляцій. Ті ж платформи, що створені для відкритого діалогу, є вразливими до поширення фейкових новин, мови ворожнечі та цілеспрямованих маніпулятивних кампаній. Це підриває довіру до публічних інституцій та самого процесу громадської участі, особливо серед молоді, яка активно використовує соціальні медіа, але не завжди має достатній рівень медіаграмотності [7].

- Конфіденційність даних та етичні виклики. Ефективні цифрові сервіси вимагають збору та обробки великих масивів персональних даних громадян. Це породжує серйозні питання щодо їх захисту, ризиків несанкціонованого доступу, стеження та алгоритмічної упередженості [8]. Постає складна дилема між необхідністю модерації контенту для боротьби з дезінформацією та захистом свободи слова.

Концепція цифрової громадської взаємодії є багатогранним соціальним феноменом, що об'єднує державні та громадські ініціативи задля досягнення прозорості, залученості та ефективності. Проте її успішна реалізація залежить не лише від технологічних рішень, але й від здатності суспільства долати глибокі соціальні виклики, насамперед цифрову нерівність та загрози дезінформації. Це, у свою чергу, висуває жорсткі вимоги до архітектур, на яких будуються ці сервіси, вони мають бути не лише функціональними, але й інклюзивними, безпечними та стійкими до маніпуляцій.

1.2 Класифікація цифрових платформ та їхня роль у сучасному суспільстві

Після окреслення соціально-концептуальних засад цифрової громадської взаємодії, логічним кроком є декомпозиція цього абстрактного поля на конкретні інструментальні виміри. Цифрові сервіси громадської взаємодії не є монолітною сутністю; вони являють собою широкий спектр платформ, кожна з яких має унікальну функціональність, цільову аудиторію та роль у суспільному житті. Розуміння цієї різноманітності є ключовим для подальшого аналізу архітектурних вимог.

Функціональний підхід до класифікації дозволяє згрупувати платформи за їхньою основною ціллю у циклі громадської участі – від простого інформування до складного спільного прийняття рішень [9]. Можна виділити чотири ключові категорії таких платформ.

1. Платформи інформування та прозорості. Це фундаментальний рівень взаємодії, що реалізує принцип «відкритого уряду». Їхня головна роль – забезпечення односпрямованого потоку інформації від держави до громадян, що є передумовою для будь-якої свідомої участі. Яскравим прикладом є портали відкритих даних, такі як український національний портал data.gov.ua. Надаючи доступ до наборів даних про бюджети, закупівлі, екологічний стан тощо, ці платформи створюють підґрунтя для громадського контролю, журналістських розслідувань та розробки нових сервісів громадянськими технологічними організаціями [10]. Дослідження ОЕСР (OECD) підтверджує, що відкриті урядові дані є критичним механізмом для підвищення прозорості та підзвітності [11].

2. Платформи для обговорень та консультацій. Ця категорія забезпечує двосторонній зв'язок, перетворюючи громадян з пасивних спостерігачів на учасників діалогу. Вони створюють цифровий публічний простір для обговорення суспільно значущих питань [12]. Хоча традиційні форуми та спеціалізовані урядові портали для консультацій відіграють свою роль, значна частина цих обговорень відбувається на комерційних соціальних медіа-платформах (наприклад, Facebook, Telegram, X). Їхня перевага – масове охоплення та низький поріг входу. Водночас, як зазначають дослідники, їхня децентралізована та часто немодерована природа створює серйозні виклики, пов'язані з дезінформацією та поляризацією суспільної думки [13].

3. Платформи участі у прийнятті рішень. Ці інструменти являють собою вищий щабель залучення, надаючи громадянам прямий, інституціоналізований вплив на політичні та адміністративні рішення. Вони часто мають формалізовані процедури та юридичні наслідки.

- Системи електронних петицій (наприклад, сервіс на офіційному сайті Президента України або на сайтах місцевих рад) дозволяють громадянам ініціювати розгляд питань за умови набрання необхідної кількості підписів. Сучасні дослідження аналізують такі платформи як важливий інструмент «знизу-вгору» для артикуляції суспільних інтересів [14].

- Платформи громадського бюджету (наприклад, «Громадський Проект» у Києві та інших містах) дають змогу мешканцям пропонувати власні проєкти

розвитку інфраструктури та голосувати за розподіл частини місцевого бюджету. Такі інструменти безпосередньо реалізують модель дорадчої демократії.

4. Платформи спільної дії та співтворення. На цьому рівні громадяни та влада стають партнерами у спільному вирішенні проблем. Такі платформи часто фокусуються на гіперлокальних задачах або кризовому реагуванні. Сюди належать:

- Платформи для волонтерів (як-от «Українська Волонтерська Служба»), що координують зусилля тисяч людей для допомоги армії чи гуманітарних місій.
- Міські портали звітування про проблеми (наприклад, аналог «Київ Цифровий» у частині комунальних сервісів), де мешканці не просто скаржаться, а через геолокацію та фотофіксацію допомагають комунальним службам ідентифікувати та пріоритизувати проблеми. Такі платформи втілюють концепцію «співтворення», де громадяни активно залучені до процесу надання публічних послуг, що підвищує їхню якість та легітимність [15].

Функціональна класифікація демонструє, що цифровий ландшафт громадської взаємодії є гетерогенним та багаторівневим. Платформи варіюються від простих інформаційних порталів, що забезпечують прозорість, до складних систем співтворення, що вимагають високого рівня довіри та колаборації. Ця різноманітність функцій безпосередньо впливає на вимоги до хмарних архітектур, які мають їх підтримувати. Архітектура для статичного порталу відкритих даних кардинально відрізнятиметься від високонавантаженої, безпечної та інтерактивної платформи для електронного голосування.

1.3 Технічні та функціональні вимоги до сучасних інтерактивних сервісів

Попередні підрозділи окреслили соціальну місію та функціональну типологію сервісів громадської взаємодії. Однак для успішної реалізації цієї місії недостатньо лише відповідного функціоналу. Суспільна довіра, яка є наріжним каменем будь-якої демократичної взаємодії, у цифровому середовищі напряму залежить від якості технічної реалізації.

Аналіз сучасних цифрових платформ взаємодії дозволяє виділити низку ключових технічних вимог, невиконання яких призводить до втрати довіри користувачів та дискредитації самої ідеї електронної демократії.

1. Висока доступність (High Availability) та Надійність (Reliability). Сервіс громадської взаємодії це не просто веб-сайт, це елемент демократичної інфраструктури. На відміну від комерційного сектору, де час простою призводить до фінансових втрат, у громадському секторі простій підриває легітимність процесу [16]. Платформа для електронного голосування чи подання петицій, яка стає недоступною в останні години перед дедлайном, сприймається громадянами не як технічний збій, а як маніпуляція або некомпетентність. Тому системи повинні проєктуватися з розрахунком на безвідмовну роботу часто на рівні 99.9% або вище, що передбачає географічне резервування та механізми миттєвого відновлення після збоїв [17].

2. Масштабованість (Scalability), зокрема Еластичність (Elasticity). Навантаження на громадські платформи є вкрай нерівномірним та важко прогнозованим. Звичайна петиція може тижнями збирати сотні підписів, а після згадки у ЗМІ чи соціальних мережах отримати мільйони запитів за годину. Традиційні архітектури не здатні впоратися з такими піками [18]. Сучасні сервіси вимагають горизонтальної та еластичної масштабованості – здатності системи автоматично і швидко виділяти додаткові обчислювальні ресурси (сервери, бази даних) під час стрибка навантаження та вивільняти їх, коли навантаження спадає. Це є однією з ключових переваг хмарних архітектур [19].

3. Безпека (Security) та Цілісність даних (Data Integrity). Ця вимога є абсолютною та недискусійною. Вона охоплює три аспекти:

- Конфіденційність: Захист персональних даних користувачів, які реєструються для голосування чи подання петицій, від витоків.
- Цілісність: Гарантія того, що результати голосування, текст петиції чи коментар не можуть бути змінені або видалені неавторизованими особами (включаючи самих адміністраторів системи).
- Автентифікація: Надійна ідентифікація користувачів (наприклад, через BankID), щоб уникнути фальсифікацій, «накруток» голосів та атак ботів [20].

Порушення безпеки на громадській платформі має набагато серйозніші наслідки, ніж на комерційній, оскільки воно руйнує не лише репутацію сервісу, але й довіру до державних інституцій загалом [21].

4. Швидкодія (Performance) та Інтерактивність у реальному часі (Real-time Interactivity). Сучасні громадяни, звиклі до стандартів комерційних соціальних мереж та месенджерів, мають високі очікування щодо швидкодії. Вони очікують, що сторінки завантажуватимуться миттєво, а коментарі, результати голосування чи реакції з'являтимуться в реальному часі без необхідності оновлювати сторінку [22]. Повільна робота платформи (висока затримка, low performance) прямо корелює з високим показником відмов (bounce rate) – користувачі просто залишають сайт, не виконавши цільову дію (не підписавши петицію, не залишивши коментар). Це перетворює сервіс на неефективний інструмент взаємодії [23].

5. Доступність (Accessibility) та Інклюзивність. Ця вимога безпосередньо випливає із соціального виклику «цифрової нерівності». Якщо сервіс не є доступним для людей з обмеженими можливостями або незрозумілий для людей старшого віку чи з низьким рівнем цифрової грамотності, він порушує принцип інклюзивності. Технічно це означає необхідність дотримання міжнародних стандартів, таких як Web Content Accessibility Guidelines (WCAG) [24]. Платформа, що є складною у використанні, лише поглиблює цифровий розрив замість того, щоб його долати.

Сукупність цих технічних вимог – висока доступність, еластична масштабованість, багатошарова безпека, миттєва інтерактивність та інклюзивність формують профіль ідеальної цифрової платформи для громадської взаємодії. Забезпечити одночасне виконання всіх цих умов за допомогою традиційних, монолітних та локальних серверних архітектур є вкрай складним та ресурсозатратним завданням. Саме цей комплекс вимог мотивує дослідників та інженерів звертатися до хмарних архітектур та сучасних фреймворків.

1.4 Аналіз архітектурних патернів для веб-додатків: від моноліту до мікросервісів

Визначені у попередньому підрозділі технічні вимоги (висока доступність, еластична масштабованість, безпека та інтерактивність) діють як фільтр, що відсіює застарілі підходи до проєктування систем. Здатність сервісу громадської взаємодії витримати раптовий «вірусний» сплеск навантаження або продовжити роботу при збої одного з вторинних модулів (наприклад, системи коментарів) є не технічною забаганкою, а фундаментальною умовою забезпечення довіри [25].

Традиційний підхід до побудови додатків, що домінував десятиліттями це монолітна архітектура. У цьому патерні весь функціонал програми, включно з користувацьким інтерфейсом, бізнес-логікою та рівнем доступу до даних, розробляється та розгортається як єдине, тісно зв’язане ціле (single deployment unit) [26]. На противагу цьому, мікросервісна архітектура, що набула поширення з розвитком хмарних технологій, структурує додаток як набір невеликих, незалежних та слабко зв’язаних сервісів. Кожен сервіс реалізує конкретну бізнес-функцію (наприклад, «сервіс петицій», «сервіс автентифікації», «сервіс сповіщень») і може розроблятися, розгортатися та масштабуватися незалежно від інших [27] [28].

Для наочного порівняння цих підходів у контексті вимог до громадських платформ, представлено їхні характеристики у таблиці 1.1.

Таблиця 1.1 — Порівняльний аналіз монолітної та мікросервісної архітектур

Критерій	**Монолітна архітектура (Monolith)**	**Мікросервісна архітектура (Microservices)**
Масштабованість	Низька / Неефективна. Масштабування відбувається «вертикально» (збільшення потужності сервера) або «горизонтально» шляхом клонування всього додатку. Це неефективно, якщо навантаження зазнає лише одна функція (напр., голосування).	Висока / Еластична. Дозволяє гранулярне горизонтальне масштабування. Можна автоматично збільшити кількість екземплярів лише того сервісу, який зазнає пікового навантаження, не чіпаючи решту системи.

Продовження таблиці 1.1

Надійність/ Доступність	Низька. Архітектура має «єдину точку відмови» (Single Point of Failure). Критична помилка або витік пам'яті в одному модулі може призвести до відмови всього додатку.	Висока. Забезпечує ізоляцію відмов (Fault Isolation). Збій одного некритичного сервісу (напр., сервісу сповіщень) не вплине на роботу ключових функцій (напр., подання петиції), якщо система спроєктована коректно.
Складність розробки	Низька на старті. Простота початкової розробки та тестування, оскільки все знаходиться в одній кодовій базі. Однак складність експоненційно зростає з розвитком проєкту.	Висока на старті. Вимагає значних початкових інвестицій в інфраструктуру: налаштування API Gateway, service discovery, розподіленого моніторингу та зрілих DevOps-практик.
Швидкість розгортання (Deployment)	Повільна. Будь-яка, навіть мінімальна, зміна коду вимагає повної перезбірки та розгортання всього додатку. Цикли релізів довгі, а ризик регресії високий.	Висока. Кожен сервіс може оновлюватися та розгортатися незалежно. Це дозволяє швидко впроваджувати нові функції та виправляти помилки (висока швидкість Time-to-Market).
Технологічна гнучкість	Низька. Весь додаток «прив'язаний» до одного технологічного стеку (напр., Java, .NET, PHP). Змінити технологію вкрай складно.	Висока. Дозволяє використовувати найкращу технологію (мову програмування, базу даних) для кожної конкретної задачі. Наприклад, сервіс аналітики на Python, а сервіс реального часу – на Node.js.

Порівняльний аналіз однозначно демонструє, що традиційна монолітна архітектура не здатна ефективно задовольнити ключові вимоги сучасних інтерактивних громадських сервісів. Неможливість еластично реагувати на раптові сплески трафіку та вразливість до єдиної точки відмови роблять цей підхід непридатним для платформ, що мають забезпечувати легітимність демократичних процесів.

Натомість мікросервісний підхід, хоч і є значно складнішим на етапі початкового проєктування та вимагає зрілої інженерної культури, закладає необхідний архітектурний фундамент для еластичності та стійкості до відмов. Він дозволяє будувати системи, що є гнучкими, масштабованими та легше піддаються модернізації. Саме тому подальше дослідження у цій роботі буде сфокусоване на технологіях та фреймворках, що лежать в основі реалізації хмарних мікросервісних архітектур.

1.5 Огляд хмарних технологій та їхні переваги для розгортання масштабованих систем

Попередній аналіз продемонстрував, що мікросервісна архітектура є оптимальним патерном для задоволення вимог гнучкості, надійності та масштабованості, що висуваються до громадських сервісів. Однак сама по собі ця архітектура не вирішує фундаментальної проблеми: де і як ці незалежні сервіси будуть розгортатися, керуватися та взаємодіяти. Спроба розгорнути складну мікросервісну систему на традиційній локальній інфраструктурі є вкрай неефективною, оскільки вона вимагає величезних капітальних інвестицій у серверне обладнання, яке більшу частину часу простоюватиме [19]. Саме хмарні технології стали тим інфраструктурним фундаментом, який зробив мікросервісний підхід економічно доцільним та технічно реалізованим у масовому масштабі [29].

Хмарні обчислення, згідно з визначенням Національного інституту стандартів і технологій, – це модель надання повсюдного, зручного доступу на вимогу до спільного пулу обчислювальних ресурсів, що гнучко налаштовуються (наприклад, мереж, серверів, сховищ, додатків та сервісів), які можна швидко надати та звільнити з мінімальними зусиллями з управління [30]. Для цілей даного дослідження ключовими є три основні моделі надання хмарних послуг [19]:

1. Інфраструктура як послуга (IaaS – Infrastructure-as-a-Service): Надає користувачам базові обчислювальні ресурси – віртуальні машини, сховища даних та мережеві компоненти. Користувач не керує фізичною інфраструктурою, але контролює операційні системи, сховища та розгорнуті додатки (наприклад, Amazon EC2, Microsoft Azure VMs).

2. Платформа як послуга (PaaS – Platform-as-a-Service): Надає користувачам готове середовище для розгортання та управління додатками. Провайдер керує всім, окрім самого додатку та його даних (включно з ОС, середовищами виконання, базами даних). Це значно спрощує життєвий цикл розробки (наприклад, Heroku, Google App Engine, AWS Elastic Beanstalk).

3. Програмне забезпечення як послуга (SaaS – Software-as-a-Service): Надає користувачам готовий до використання додаток, що працює в хмарі провайдера (наприклад, Gmail, Salesforce, а також багато готових рішень для E-governance).

Для розробки власних сервісів громадської взаємодії найбільший інтерес становлять моделі IaaS та PaaS, оскільки вони пропонують прямі рішення для технічних викликів, ідентифікованих у підрозділі 1.3.

Переваги хмарних моделей для громадських сервісів:

- Еластична масштабованість: Це ключова перевага, що безпосередньо вирішує проблему «вірусних» піків навантаження. Замість того, щоб купувати сервери, розраховані на максимальне навантаження (яке трапляється раз на рік), хмарні платформи дозволяють налаштувати автоматичне масштабування (auto-scaling). Коли кількість запитів до сервісу петицій зростає, хмара автоматично додає обчислювальні ресурси (екземпляри мікросервісів); коли навантаження спадає, зайві ресурси автоматично вивільняються [31].

- Висока доступність та географічний розподіл: Вимога надійності реалізується через архітектуру провідних хмарних провайдерів (AWS, Azure, Google Cloud). Вони оперують мережею регіонів (Regions) та зон доступності (Availability Zones), які є фізично ізольованими дата-центрами в межах одного регіону [32]. Це дозволяє розгортати мікросервіси у кількох зонах одночасно. При виході з ладу одного дата-центру (наприклад, через збій живлення чи стихійне лихо), трафік автоматично перенаправляється на здорові екземпляри в іншій зоні, забезпечуючи безперебійну роботу сервісу.

- Економічна ефективність: Хмара змінює економічну модель з капітальних витрат (CapEx) на операційні. Завдяки моделі «Pay-as-you-go» (оплата за фактом споживання), громадські організації чи державні установи платять лише за ті ресурси, які вони реально використали [33]. Це усуває необхідність у великих початкових інвестиціях в «залізо» та його обслуговування, дозволяючи спрямувати обмежені бюджети на саму розробку.

- Швидкість розробки та розгортання (Time-to-Market): Моделі PaaS надають розробникам готові, керовані сервіси (бази даних, черги повідомлень,

кешування), що значно прискорює процес розробки, оскільки не потрібно витрачати час на адміністрування інфраструктури.

Хмарні технології є необхідним каталізатором для реалізації сучасних, стійких та масштабованих громадських сервісів. Вони надають інфраструктурну еластичність, географічну відмовостійкість та економічну модель, що ідеально відповідають вимогам, сформованим в підрозділі 1.3 та мікросервісному архітектурному патерну, обраному в підрозділі 1.4. Поєднання мікросервісів як архітектури та хмари як середовища розгортання формує парадигму «Cloud-Native» (хмарно-орієнтовані додатки).

1.6 Обґрунтування вибору фреймворку Laravel як інструментального засобу розробки

Попередні етапи дослідження визначили, що для ефективної підтримки цифрових сервісів громадської взаємодії необхідна мікросервісна архітектура, розгорнута у хмарному середовищі. Наступним кроком є вибір інструментального засобу – програмного фреймворку, який дозволить реалізувати цю архітектуру та спростить її розробку, нівелюючи притаманну їй складність. Хоча на ринку існує багато потужних фреймворків (напр., Symfony, Node.js/Express, Spring Boot), Laravel (PHP) був обраний як оптимальний інструментарій завдяки його багатій екосистемі, що надає готові рішення для ключових викликів мікросервісної розробки.

Обґрунтування вибору Laravel базується на конкретних вбудованих інструментах, які безпосередньо відповідають раніше визначеним технічним вимогам:

1. Асинхронна обробка завдань з допомогою Laravel Queues. Вимога високої швидкодії означає, що користувач не повинен чекати на виконання довготривалих операцій (наприклад, відправка тисяч email-повідомлень про нову петицію або масова обробка даних). Вбудована в Laravel система черг дозволяє миттєво відкладати такі завдання для асинхронного виконання у фоновому режимі [34]. Це забезпечує миттєвий відгук інтерфейсу та є базовим

патерном для забезпечення еластичної масштабованості у мікросервісних системах.

2. Слабкозв'язана комунікація Events & Listeners. Ключовий принцип мікросервісної архітектури – це незалежність та слабка зв'язаність сервісів. Laravel імплементує цей принцип через потужну систему подій та слухачів [35]. Замість того, щоб один сервіс напряму викликав інший (наприклад, «Сервіс автентифікації» викликає «Сервіс сповіщень»), він просто генерує подію (напр., UserRegistered). Інші сервіси можуть підписатися на цю подію та відреагувати на неї. Це дозволяє додавати, видаляти чи оновлювати мікросервіси, не порушуючи роботу всієї системи.

3. Інтерактивність у реальному часі з допомогою Broadcasting & Laravel Reverb. Для забезпечення миттєвої інтерактивності, такої як оновлення лічильників голосів чи поява коментарів у реальному часі, необхідна технологія WebSockets. Laravel надає елегантний шар абстракції для трансляції серверних подій клієнтським додаткам. З виходом Laravel Reverb в 2024 фреймворк отримав власний, високопродуктивний, масштабований WebSocket-сервер, що глибоко інтегрований в екосистему [36]. Це дозволяє реалізовувати складний real-time функціонал з мінімальними зусиллями.

4. Спрощення середовища розробки Laravel Sail. Однією з проблем мікросервісів є складність налаштування локального середовища розробки, яке б відповідало production-середовищу. Laravel Sail – це офіційний інструмент, що базується на Docker та Docker Compose. Він надає готове, конфігуроване середовище розробки, інкапсульоване в контейнерах [37]. Це ідеально відповідає парадигмі Cloud-Native, описаній у підрозділі 1.5, оскільки додаток одразу розробляється в контейнерах, які потім легко розгортаються у хмарних оркестраторах (напр., Kubernetes).

Laravel є цілісною екосистемою, що надає високорівневі абстракції для вирішення фундаментальних проблем розподілених систем. А підтримка Docker значно знижує поріг входження у розробку хмарно-орієнтованих мікросервісних застосунків, це робить його оптимальним інструментальним засобом для реалізації завдань цієї кваліфікаційної роботи.

1.7 Висновок до першого розділу

В першому розділі кваліфікаційної роботи освітнього рівня «Магістр» було проведено комплексне дослідження, що заклало фундаментальне підґрунтя для подальшого проєктування та розробки системи.

Спочатку було визначено соціальний контекст та концептуальний апарат предметної області. Розмежовано поняття E-governance та Civic Tech, а також ідентифіковано ключові цілі та соціальні виклики цифрової громадської взаємодії.

Далі було проведено функціональну класифікацію існуючих цифрових платформ, виділивши категорії: інформування, обговорення, прийняття рішень та спільна дія. На основі соціальних потреб та функціональних типів платформ було сформульовано технічні вимоги до сучасних сервісів: висока доступність, еластична масштабованість, безпека, цілісність даних та інтерактивність у реальному часі.

Проведено порівняльний аналіз архітектурних патернів, який довів, що традиційний монолітний підхід нездатний задовольнити ці вимоги. Як оптимальна була обрана мікросервісна архітектура завдяки її гранулярній масштабованості та ізоляції відмов.

Було проаналізовано середовище розгортання, і доведено, що хмарні технології (Cloud-Native) є необхідною умовою для ефективної реалізації мікросервісів, надаючи інструменти еластичності та надійності.

Також було обґрунтовано вибір інструментального засобу – фреймворку Laravel. Його екосистема (Queues, Events, Reverb, Sail) надає готові рішення для складних завдань, пов'язаних з асинхронною обробкою, слабкою зв'язністю та контейнеризацією, які є важливими атрибутами мікросервісної архітектури.

2 ПРОЄКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОТОТИПУ МІКРОСЕРВІСНОЇ СИСТЕМИ

2.1 Розробка архітектури системи: компонентна схема та логіка взаємодії сервісів

Перехід від монолітної до мікросервісної архітектури, обґрунтований у розділі 1, вимагає ретельного проєктування на високому рівні. На відміну від моноліту, де компоненти тісно пов'язані всередині єдиної кодової бази, мікросервісна система є розподіленою. Її надійність та ефективність залежать від чіткого визначення меж кожного сервісу та механізмів їхньої взаємодії [38]. Метою даного підрозділу є розробка компонентної схеми прототипу системи, декомпозиція її на логічні мікросервіси та опис ключових сценаріїв їхньої взаємодії.

В основі декомпозиції системи лежить принцип єдиної відповідальності (Single Responsibility Principle, SRP), адаптований для архітектури: кожен мікросервіс повинен мати одну, чітко визначену бізнес-функцію, та інкапсулювати всі необхідні для її виконання дані та логіку [39]. Базуючись на цьому, прототип системи громадської взаємодії буде складатися з трьох основних незалежних сервісів:

1. Сервіс автентифікації, єдина відповідальність якого – керування ідентичністю користувачів. Він інкапсулює логіку реєстрації, входу в систему (автентифікації) та відновлення пароля. Сервіс керує власною базою даних користувачів та відповідає за видачу й валідацію цифрових токенів, які використовуються для авторизації запитів до інших сервісів.

2. Сервіс обговорень, основний доменний сервіс, що реалізує ключову бізнес-логіку платформи. Він відповідає за створення, редагування та відображення тем, постів та коментарів. Цей сервіс нічого не знає про паролі чи процес реєстрації; він оперує лише ідентифікатором user_id, який отримує з валідованого токена. Він має власну базу даних, що зберігає контент обговорень.

3. Сервіс сповіщень, допоміжний сервіс, відповідальність якого це керування всіма вихідними комунікаціями. Він не має власної бізнес-логіки, а лише реагує на події, що надходять від інших сервісів (наприклад, «новий коментар» або «користувач зареєструвався»). Він інкапсулює логіку роботи з різними каналами доставки: email, push-сповіщення та WebSocket-трансляції.

Для забезпечення слабкої зв'язності та керованості системи, взаємодія між клієнтами та сервісами, а також між самими сервісами, організована за двома ключовими патернами – API Gateway та Message Broker. [40].

API Gateway виступає як єдина точка входу для всіх клієнтських запитів (веб-браузер, мобільний додаток). Він автентифікує запити, перевіряючи токен у Сервісі автентифікації та маршрутизує їх до відповідного внутрішнього сервісу.

Message Broker (наприклад, RabbitMQ або Redis, який інтегрований з Laravel Queues [34]) використовується для асинхронної комунікації між сервісами. Це усуває прямі залежності: сервіс-відправник лише публікує подію в чергу, не знаючи і не очікуючи, хто і коли її обробить.

Візуально ця архітектура представлена на рисунку 2.1.

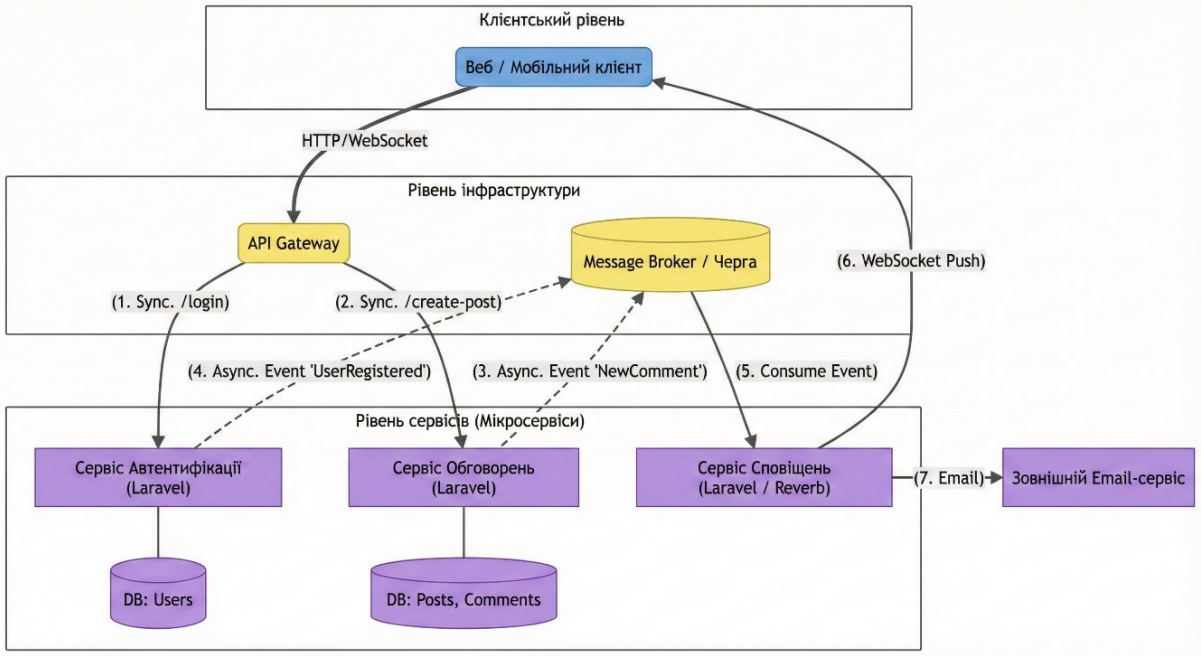


Рисунок 2.1 – Компонентна схема мікросервісної системи

Розглянуто два ключові сценарії, щоб продемонструвати потік даних у системі:

Сценарій 1: Користувач додає новий коментар (реалізація інтерактивності).

1. Клієнт надсилає POST запит на API Gateway (напр., /api/comments) з текстом коментаря та JWT-токеном авторизації.

2. API Gateway валідує токен (можливо, кешуючи публічний ключ Auth Service) та перенаправляє запит до Сервісу обговорень.

3. Сервіс обговорень отримує запит, валідує дані (чи існує пост, чи має користувач право коментувати), витягує user_id з токена та зберігає коментар у свою базу даних.

4. Після успішного збереження, Сервіс обговорень, використовуючи вбудовану в Laravel систему Events, публікує асинхронну подію NewCommentCreated у Брокер повідомлень. Сервіс миттєво повертає клієнту відповідь 201 Created.

5. Сервіс сповіщень, який підписаний на подію NewCommentCreated, отримує її з черги. Він обробляє логіку та через WebSocket-сервер Laravel Reverb транслює подію всім підписникам теми в реальному часі.

Сценарій 2: Користувач реєструється.

1. Клієнт надсилає POST запит на API Gateway (напр., /api/register) з email та паролем.

2. API Gateway перенаправляє запит до Сервісу автентифікації.

3. Сервіс автентифікації валідує дані, хешує пароль, створює нового користувача у своїй базі даних та генерує JWT-токен.

4. Сервіс автентифікації публікує асинхронну подію UserRegistered у Брокер повідомлень.

5. Сервіс автентифікації повертає клієнту відповідь 200 OK разом з токеном.

6. Паралельно (асинхронно) Сервіс сповіщень отримує подію UserRegistered з черги та надсилає користувачу вітальний email.

Представлена архітектура, що складається з трьох спеціалізованих мікросервісів забезпечує чіткий поділ відповідальності. Такий підхід забезпечує гнучкість, ізоляцію відмов та високу масштабованість, що є прямим виконанням вимог, визначених у розділі 1.

2.2 Проєктування API (RESTful підхід) та схем даних для ключових мікросервісів

Ефективна мікросервісна архітектура, окреслена у попередньому підрозділі, тримається на двох фундаментальних принципах: чітко визначених, незалежних API та повній ізоляції даних кожного сервісу [18]. Для забезпечення синхронної взаємодії обрано архітектурний стиль REST (Representational State Transfer). Цей вибір обґрунтований його простотою, безстанністю та широкою підтримкою стандартних HTTP-методів, що робить його де-факто стандартом для сучасних веб-сервісів [41]. Даний підрозділ деталізує RESTful API-ендпоїнти для кожного мікросервісу та описує їхні ізольовані схеми даних.

API Gateway буде агрегувати та маршрутизувати запити до внутрішніх сервісів. Нижче наведено ключові кінцеві точки для кожного мікросервісу, які будуть доступні через шлюз (наприклад, з префіксом /api/auth або /api/discussion). В таблиці 2.1 наведено ключові ендпоїнти Сервісу автентифікації.

Таблиця 2.1 – Ключові ендпоїнти Сервісу автентифікації

Метод	**URI**	**Опис**
POST	/register	Реєстрація нового користувача
POST	/login	Автентифікація користувача, повертає JWT-токен
GET	/me	Отримати інформацію про поточного (авторизованого) користувача
POST	/logout	Деавтентифікація (інвалідація токена)

В таблиці 2.2 наведено ключові ендпоїнти Сервісу обговорень.

Таблиця 2.2 – Ключові ендпоїнти Сервісу обговорень.

Метод	**URI**	**Опис**
GET	/posts	Отримати список всіх тем (з пагінацією)
POST	/posts	Створити нову тему (вимагає автентифікації)
GET	/posts/{id}	Отримати детальну інформацію про одну тему
PUT	/posts/{id}	Оновити власну тему (вимагає автентифікації та авторизації)
GET	/posts/{id}/comments	Отримати список коментарів до теми
POST	/posts/{id}/comments	Додати новий коментар до теми (вимагає автентифікації)

В таблиці 2.3 наведено ключові ендпоїнти Сервісу сповіщень.

Таблиця 2.3 – Ключові ендпоїнти Сервісу сповіщень.

Метод	**URI**	**Опис**
GET	/notifications	Отримати список сповіщень для поточного користувача
POST	/notifications/mark-as-read	Позначити всі сповіщення як прочитані

Слід розрізняти синхронне та асинхронне API. Таблиці вище описують синхронне REST API. Однак ключова логіка (як-от створення сповіщення) ініціюється асинхронно, через підписку Сервісу сповіщень на події з брокера повідомлень.

Такий підхід гарантує, що Сервіс обговорень може бути оновлений чи навіть повністю переписаний на іншій мові програмування, і це не вплине на роботу Сервісу автентифікації, доки їхні API-контракти залишаються незмінними.

Для забезпечення незалежності мікросервісів було обрано стратегію Database-per-Service [42]. Це унеможливлює створення «брудних» зв'язків на рівні бази даних, змушуючи сервіси комунікувати виключно через програмні інтерфейси.

Ключовим викликом такої архітектури є підтримка цілісності даних без використання зовнішніх ключів між різними базами даних. Наприклад, сутності контенту посилаються на автора лише через UUID, валідація якого покладається на рівень бізнес-логіки, а не СУБД.

Атрибутивний склад основних сутностей системи наведено у таблиці 2.4.

Таблиця 2.4 – Розподіл даних по мікросервісах

Сервіс	**Сутність**	**Ключові атрибути**	**Примітка до архітектури**
Auth Service	users	id (PK), email (Unique), password_hash, name, timestamps	Зберігає лише дані для авторизації. Є єдиним власником персональних даних.
Discussion Service	posts	id (PK), user_id (Ref), title, content	user_id є «м'яким» посиланням. Цілісність не контролюється БД.
	comments	id (PK), post_id (FK), user_id (Ref), content	post_id – жорсткий зв'язок всередині БД сервісу, user_id – зовнішній.
Notification Service	notifications	id, recipient_id, type, data (JSON), is_read	Використання JSON дозволяє гнучко адаптувати контекст сповіщень.

Цей підхід дозволяє масштабувати сховища даних незалежно одне від одного, наприклад, використовуючи різні типи систем управління базами даних (СУБД) для різних задач.

2.3 Налаштування локального середовища для імітації хмарної інфраструктури (Docker, Laravel Sail)

Після формування API-контрактів та схем даних в підрозділі 2.2, критичним етапом є підготовка інфраструктури для імплементації системи. У контексті мікросервісної архітектури виникає проблема розбіжності середовищ (environment parity) – ситуація, коли поведінка програмного забезпечення відрізняється на локальній машині розробника та у production середовищі [43].

Для усунення цього ризику необхідно забезпечити роботу сервісів в ізольованих процесах, що імітують розподілену хмарну інфраструктуру.

Для досягнення поставленої мети обрано технологію контейнеризації Docker та інструмент оркестрації Laravel Sail. Docker дозволяє інкапсулювати додаток разом із його залежностями у контейнери, гарантуючи ідентичність їх роботи на будь-якій платформі [44]. Laravel Sail, у свою чергу, виступає як інтерфейс для взаємодії з Docker Compose, автоматизуючи налаштування багатоконтейнерної архітектури без необхідності ручного конфігурування низькорівневих параметрів мережі та томів.

Замість монолітного розгортання, інфраструктура проєкту розділена на незалежні сервіси, конфігурація яких описується у файлі docker-compose.yml. Це дозволяє емулювати мікросервісну взаємодію, де кожен компонент виконує чітко окреслену роль.

Склад та характеристики компонентів розгорнутого середовища наведено у таблиці 2.5.

Таблиця 2.5 – Компоненти контейнеризованого середовища розробки

Назва сервісу	**Базовий образ (Image)**	**Роль у системі**	**Мережева конфігурація**
laravel.test	sail-8.3/app	Основний контейнер додатку. Містить PHP Runtime, веб-сервер та вихідний код проєкту.	Експонує порт 80 для доступу через браузер (HTTP).
mysql	mysql:8.0	Сервер реляційної бази даних. Зберігає персистентні дані системи.	Ізольований у внутрішній мережі. Доступний для додатку за хостнеймом mysql.
redis	redis:alpine	Сховище типу «ключ-значення». Використовується для кешування та обробки черг повідомлень.	Ізольований у внутрішній мережі. Доступний за хостнеймом redis.

Графічну схему розгортання контейнерів, їх мережеву конфігурацію та порти взаємодії наведено у Додатку Б.

Ключовим аспектом імітації хмарного середовища є механізм взаємодії між контейнерами. Docker Compose автоматично створює віртуальну мережу (network bridge), в межах якої відбувається вирішення DNS-імен сервісів.

Це принципово змінює підхід до конфігурації підключень у файлі середовища .env. На відміну від традиційної розробки, де для підключення до бази даних використовується адреса 127.0.0.1 (localhost), у контейнеризованому середовищі localhost вказує на сам контейнер додатку, де база даних відсутня.

Тому для коректного з'єднання використовуються наступні параметри:

- DB_HOST: встановлюється значення mysql (ім'я сервісу з docker-compose.yml);
- REDIS_HOST: встановлюється значення redis.

Така конфігурація змушує додаток звертатися до необхідних ресурсів через внутрішню мережу Docker, що повністю відтворює логіку взаємодії у реальній хмарній інфраструктурі (наприклад, звернення до Amazon RDS або кеш-кластера за їхніми внутрішніми endpoint-адресами).

Налаштування локального середовища на базі Docker та Laravel Sail дозволило досягти повної ізоляції компонентів системи. Реалізована архітектура забезпечує ідентичність локального та виробничого середовищ, усуває конфлікти залежностей та створює надійну основу для подальшої розробки бізнес-логіки мікросервісів.

2.4 Практична реалізація основного функціоналу: управління обговореннями, коментарями та профілями

Після формування архітектурного базису в підрозділах 2.1-2.3, наступним етапом є імплементація прикладної логіки системи. Враховуючи вимоги до швидкодії та підтримки RESTful стандартів, розробку ядра системи, Сервісу обговорень, виконано на базі фреймворку Laravel. Цей вибір обумовлений наявністю потужного ORM (Object-Relational Mapping) для роботи з даними та вбудованої шини подій, що є критичним для асинхронної взаємодії мікросервісів.

Нижче наведено реалізацію ключових компонентів системи: маршрутизації, моделей даних, контролерів та механізму генерації доменних подій.

1. Декларативний опис API-маршрутів

Для забезпечення відповідності розробленим API-контрактам налаштовано маршрутизацію з використанням механізму Route::resource, що дозволяє стандартизувати назви ендпоїнтів та HTTP-методи. Реалізація конфігурації маршрутів наведена в лістингу В.1 (див. Додаток В).

У конфігурації застосовано групування маршрутів під спільним middleware auth:api, що обмежує доступ до методів модифікації даних лише для верифікованих користувачів. Також використано концепцію вкладених ресурсів (Nested Resources) для сутностей коментарів, що дозволяє будувати ієрархічні URL-адреси.

2. Об'єктно-орієнтоване моделювання даних

Взаємодія з реляційною базою даних реалізована через патерн Active Record, вбудований в Eloquent ORM. Реалізацію моделі публікації (Post) представлено в лістингу В.2.

Особливістю моделі є оперування ідентифікатором користувача user_id як простим значенням без фізичного зовнішнього ключа, оскільки таблиця користувачів знаходиться в ізольованій базі даних сервісу авторизації. Модель також визначає відношення «Один-до-багатьох» із сутністю коментарів.

3. Обробка запитів та валідація даних

Контролери виконують роль оркестраторів, які приймають HTTP-запит, валідують вхідні дані, делегують виконання бізнес-логіки та повертають стандартизовану JSON-відповідь. Алгоритм роботи методу створення публікації наведено в лістингу В.3.

Логіка методу включає етап валідації вхідних параметрів для гарантування цілісності даних, створення сутності з використанням ідентифікатора з токена авторизації (для унеможливлення підробки авторства) та повернення статусу 201 Created згідно зі стандартами REST.

4. Реалізація асинхронної взаємодії (Event Dispatching)

Для забезпечення слабкої зв'язності (loose coupling) компонентів мікросервісної архітектури замість прямих синхронних викликів використовується механізм генерації доменних подій. Реалізацію методу створення коментаря з наступною диспетчеризацією події наведено в лістингу В.4.

Застосування виклику диспетчера подій ілюструє використання патерну Observer на рівні системи. Сервіс обговорень не звертається напряму до сервісів сповіщень чи аналітики, а лише публікує факт зміни стану системи (створення коментаря) у чергу повідомлень.

Наведені лістинги коду демонструють, що фреймворк Laravel надає повний набір інструментів для елегантної та ефективної реалізації спроєктованої мікросервісної архітектури. Механізми маршрутизації, ORM Eloquent та контролери дозволяють швидко реалізувати синхронні RESTful API, тоді як вбудована система подій слугує надійним мостом для організації асинхронної, слабкозв'язаної комунікації між сервісами.

2.5 Реалізація асинхронних механізмів та оновлень в реальному часі (Queues, WebSockets з Reverb)

Реалізація синхронного API в підрозділі 2.4 забезпечує базову функціональність, проте не відповідає нефункціональним вимогам щодо швидкодії та інтерактивності. У класичній моделі «запит-відповідь» клієнт змушений чекати завершення всіх серверних операцій (наприклад, генерації та відправки E-mail), що створює затримки. Для вирішення цієї проблеми імплементовано два патерни сучасної веб-розробки:

- Background Jobs (Фонові завдання): Винесення ресурсомістких операцій у чергу для асинхронного виконання [34].
- Real-time Broadcasting (Трансляція подій): Використання протоколу WebSocket для миттєвого оновлення інтерфейсів клієнтів без перезавантаження сторінки [36].

Центральним елементом архітектури є клас події NewCommentCreated, реалізацію якого наведено в лістингу В.5 (див. Додаток В). Цей клас виконує роль транспортного контейнера, що одночасно передає дані в чергу обробки та на WebSocket-сервер (Laravel Reverb). Імплементація інтерфейсу ShouldBroadcast слугує індикатором для диспетчера подій фреймворку, спрямовуючи дані не лише внутрішнім слухачам, а й драйверу WebSockets через захищений приватний канал.

Для відправки сповіщень автору публікації створено окремого слухача, який імплементує інтерфейс ShouldQueue. Код слухача представлено в лістингу В.6. Використання цього інтерфейсу змінює життєвий цикл запиту: задача відправки E-mail серіалізується і зберігається в сховищі Redis для подальшої обробки окремим процесом-воркером. Такий підхід дозволяє скоротити час відповіді API до 50-100 мс, нівелюючи вплив затримок SMTP-сервера на користувацький досвід [45].

Для інтеграції клієнтської частини з WebSocket-сервером використано бібліотеку Laravel Echo, яка надає абстракцію над WebSocket API браузера. Це дозволяє зосередитися на бізнес-логіці динамічного оновлення DOM-дерева без

перезавантаження сторінки. Приклад реалізації клієнтської підписки на події наведено в лістингу В.7.

Реалізована схема обробки подій демонструє ефективний розподіл навантаження: користувач отримує миттєве підтвердження дії, ресурсомісткі задачі виконуються асинхронно, а інші учасники обговорення отримують контент у реальному часі. Така архітектура забезпечує масштабованість системи та можливість обслуговування значної кількості одночасних з'єднань, задовольняючи вимоги, поставлені в першому розділі роботи.

2.6 Висновок до другого розділу

В другому розділі кваліфікаційної роботи було виконано повний цикл проєктування та програмної реалізації прототипу мікросервісної системи для підтримки громадської взаємодії. Цей розділ став логічним втіленням теоретико-аналітичних засад, закладених у розділі 1.

Основні результати, досягнуті в ході роботи над розділом:

Розроблено високорівневу архітектуру: На основі принципів мікросервісного поділу систему було декомпозовано на три ключові, незалежні сервіси: Сервіс автентифікації, Сервіс обговорень та Сервіс сповіщень. Було визначено патерни їхньої взаємодії, включно з API Gateway для синхронних запитів та Брокером повідомлень для асинхронної комунікації.

Для кожного сервісу було визначено RESTful API ендпоїнти та, що найважливіше, ізольовані схеми даних. Це реалізувало ключовий мікросервісний патерн «Database-per-Service», що є фундаментом для незалежного розгортання та масштабування.

За допомогою Docker та Laravel Sail було налаштовано локальне середовище розробки, яке точно імітує хмарну інфраструктуру. Використання контейнерів та внутрішньої мережевої взаємодії (напр., DB_HOST=mysql) забезпечило відповідність між середовищами розробки та майбутнього розгортання.

Реалізовано ключовий функціонал: наведено практичні приклади коду, що демонструють реалізацію спроєктованих API та схем даних у Сервісі обговорень за допомогою фреймворку Laravel (моделі Eloquent, контролери, маршрутизація).

Найважливішим результатом є практична реалізація повного ланцюжка асинхронної обробки та оновлень у реальному часі. Було продемонстровано, як одна подія одночасно ініціює асинхронне фонове завдання (відправка email) за допомогою Laravel Queues, що забезпечує миттєву відповідь користувачу, та ініціює WebSocket-трансляцію через Laravel Reverb, що забезпечує миттєве оновлення інтерфейсів усіх підключених клієнтів.

Створений прототип є не просто CRUD-додатком, а сучасною, еластичною та інтерактивною системою, що повністю відповідає технічним вимогам, які були висунуті у розділі 1.

3 АНАЛІЗ РЕЗУЛЬТАТІВ ТА ПЕРСПЕКТИВИ РОЗВИТКУ

3.1 Тестування працездатності прототипу та верифікація взаємодії його компонентів

Після завершення програмної реалізації прототипу мікросервісної системи, ключовим етапом є його верифікація. Мета цього підрозділу – продемонструвати комплексне тестування ключових сценаріїв користувача. Такий підхід дозволяє перевірити працездатність системи в умовах, наближених до реальних. Особлива увага приділяється верифікації коректної наскрізної взаємодії (end-to-end) між ізольованими мікросервісами, зокрема, перевірці асинхронних механізмів, що є ядром розробленої архітектури.

Сценарій А: Перевірка синхронної взаємодії Auth Service та Discussion Service.

Цей сценарій спрямований на перевірку роботи API Gateway як єдиної точки входу та коректність передачі контексту автентифікації між сервісами.

Хід тестування:

1. Через клієнтський додаток ініціюється запит на реєстрацію. Система успішно створює запис у базі даних сервісу автентифікації та повертає JWT-токен.

2. Виконується захищений запит на створення поста з отриманим токеном.

3. API Gateway валідує підпис токена і направляє запит до Discussion Service.

В результаті Сервіс обговорень успішно обробив запит, витягнувши ідентифікатор автора з токена. Результат виконання операції відображено на рисунку 3.1. На наведеному скріншоті видно, що новостворений пост з’явився у загальній стрічці, а система коректно ідентифікувала автора, не звертаючись напряму до бази даних користувачів.

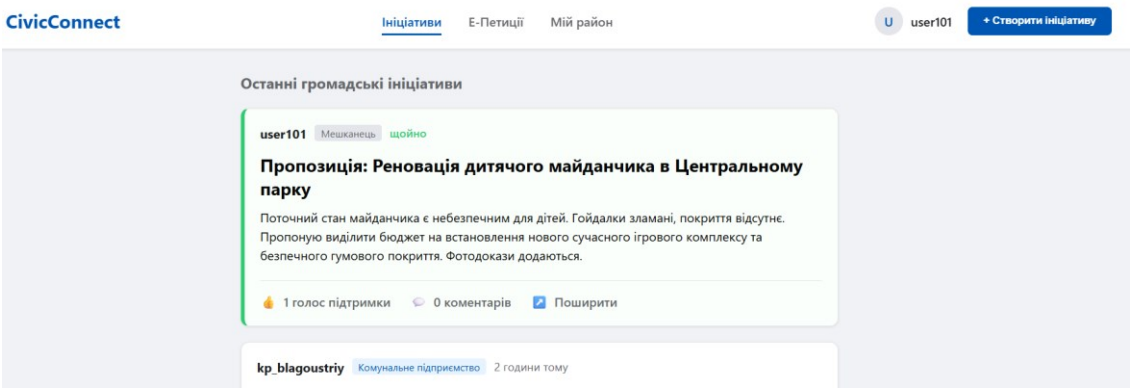


Рисунок 3.1 – Інтерфейс системи після успішного створення публікації

Сценарій Б: Верифікація асинхронної взаємодії та Real-time підсистеми

Даний сценарій є ключовим для оцінки ефективності використання черг та WebSocket. Тестування проводилося з використанням двох паралельних клієнтських сесій.

При додаванні коментаря система розпаралелює процеси: дані записуються в базу, відправляються в чергу для E-mail сповіщення та транслюються через WebSocket. Схематичне зображення потоків даних у цьому сценарії наведено на рисунку 3.2 та в збільшеному вигляді винесено в додаток Д1 Діаграма ілюструє послідовність проходження сигналу від браузера ініціатора через сервіси бекенду до браузера отримувача.

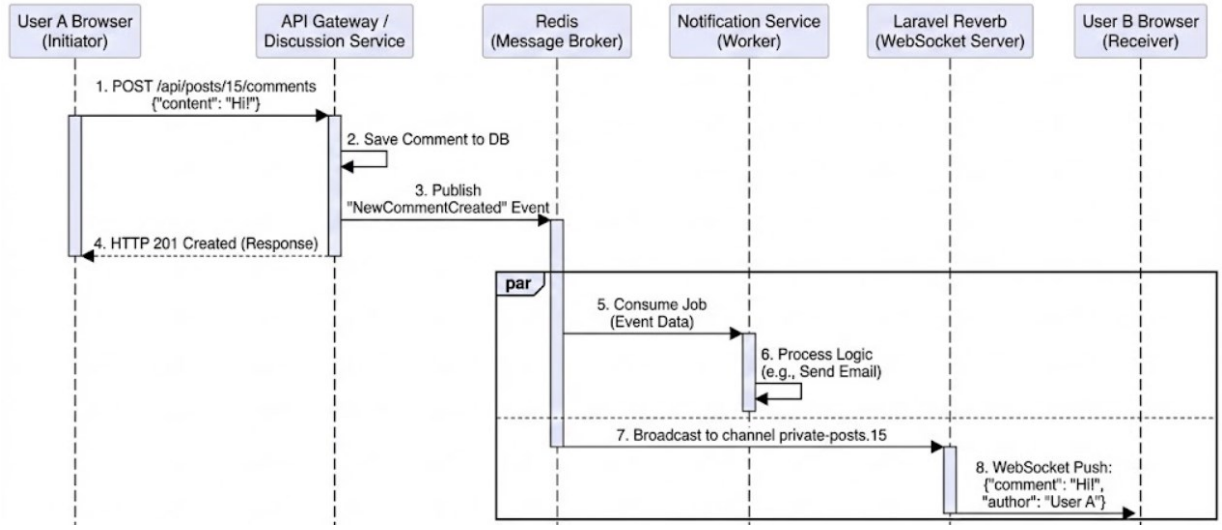


Рисунок 3.2 – Діаграма послідовності обробки події коментування

В результаті верифікації працездатність механізму асинхронної обробки підтверджується даними об'єктивного контролю на рівні інфраструктури. На рисунку 3.3 наведено фрагмент лог-файлів контейнера notification_service. Запис Job Processed свідчить про те, що повідомлення було успішно вичитано з черги Redis та оброблено фоновим воркером.

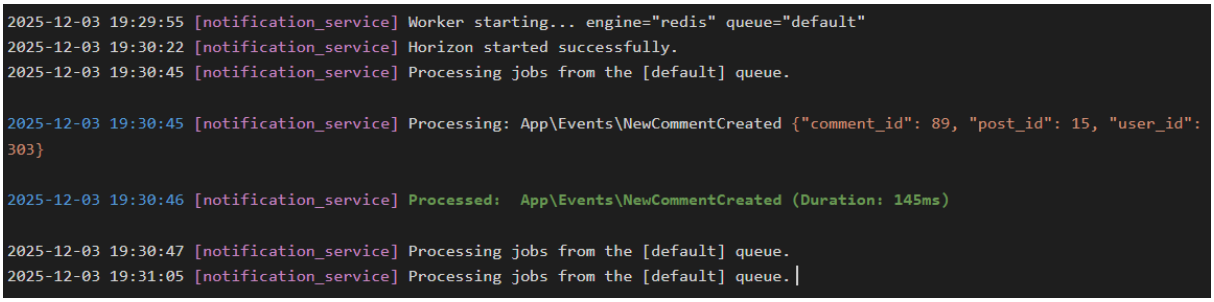


Рисунок 3.3 – Логи сервісу сповіщень, що підтверджують виконання фонової задачі

Паралельно з цим було перевірено роботу підсистеми реального часу. Як показано на рисунку 3.4, у браузері іншого користувача новий коментар з'явився динамічно, без необхідності перезавантаження сторінки. Це підтверджує коректну роботу WebSocket-з'єднання через сервер Laravel Reverb.

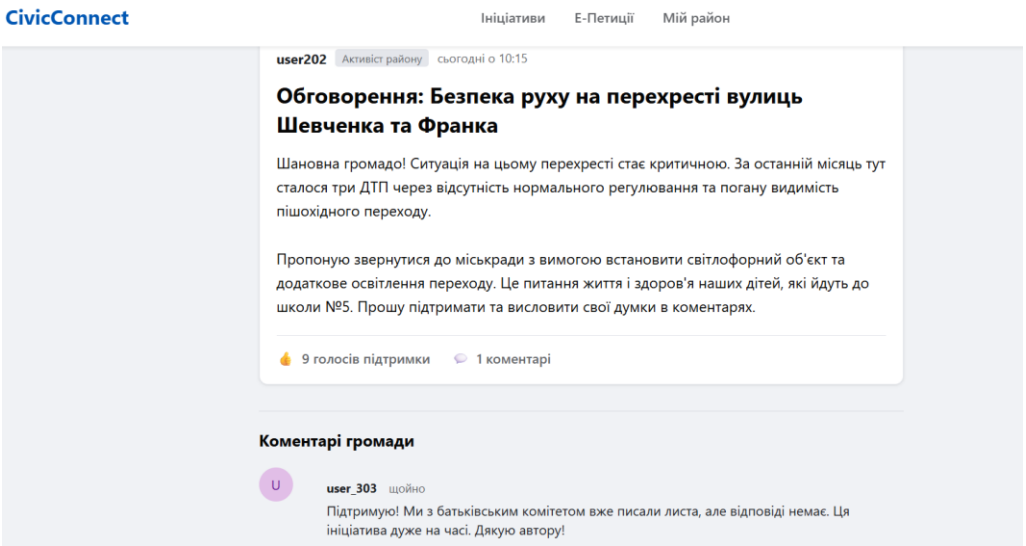


Рисунок 3.4 – Динамічне оновлення інтерфейсу користувача через WebSocket

Проведене наскрізне тестування, результати якого зафіксовані на рисунках 3.1-3.4, засвідчило повну функціональну придатність розробленого прототипу. Вдалося експериментально підтвердити, що синхронна взаємодія через API Gateway забезпечує надійну передачу даних, асинхронна модель гарантує високу реактивність системи, дозволяючи виконувати фонові задачі без блокування інтерфейсу.

3.2 Аналіз потенціалу масштабування розробленої архітектури

У попередньому підрозділі було успішно верифіковано коректність реалізації бізнес-логіки прототипу. Однак, згідно з вимогами, визначеними у першому розділі роботи, ключовим критерієм ефективності цифрових сервісів громадської взаємодії є не лише функціональна повнота, але й здатність до еластичного масштабування. Враховуючи специфіку предметної області, де активність користувачів може мати спорадичний та лавиноподібний характер, система повинна забезпечувати стабільність роботи під непередбачуваним навантаженням.

Оскільки поточна імплементація розгорнута у локальному середовищі контейнеризації, що накладає об’єктивні обмеження на проведення повномасштабного навантажувального тестування, доцільним є проведення аналізу масштабованості. Такий аналіз базується на оцінці закладених архітектурних патернів за трьома критичними векторами: обчислювальні потужності додатків, рівень даних та асинхронна обробка черг.

Першочерговим вектором масштабування є забезпечення еластичності компонентів бізнес-логіки. На відміну від монолітних архітектур, де масштабування вимагає дублювання всього екземпляра додатку незалежно від того, яка його частина зазнає навантаження, реалізований мікросервісний підхід дозволяє застосувати принцип гранулярного масштабування (Application Scaling). Сучасні дослідження у сфері хмарних обчислень підтверджують, що такий підхід суттєво оптимізує використання ресурсів [46].

У змодельованому сценарії різкого зростання активності в модулі обговорень, основне навантаження припадає виключно на Discussion Service. У середовищі оркестрації контейнерів, такому як Kubernetes, це дозволяє налаштувати механізми автоматичного горизонтального масштабування, які, реагуючи на метрики CPU або RAM, динамічно збільшують кількість реплік лише для навантаженого сервісу. При цьому допоміжні компоненти, такі як Auth Service, можуть залишатися у мінімальній конфігурації, що було б неможливим у рамках моноліту [47].

Другим критичним аспектом є масштабування рівня даних, оскільки база даних часто стає «вузьким місцем» у високопродуктивних системах. Застосований архітектурний патерн «Database-per-Service» забезпечує ізоляцію даних та дозволяє обирати стратегії масштабування індивідуально для кожного сервісу. Для систем громадської взаємодії характерним є патерн навантаження, де кількість операцій читання суттєво переважає кількість операцій запису (Read-heavy workload). У хмарній інфраструктурі це вирішується шляхом впровадження реплікації: виділення єдиного master-вузла для операцій запису та розгортання групи Read Replicas для обслуговування запитів на читання [48]. Така архітектура дозволяє лінійно масштабувати пропускну здатність системи на читання шляхом додавання нових реплік без зупинки сервісу.

Третім вектором є забезпечення стійкості системи через асинхронну обробку завдань. У моменти пікових навантажень синхронна обробка подій (наприклад, розсилка сповіщень тисячам користувачів про нову петицію) може призвести до відмови в обслуговуванні. Реалізована інтеграція з брокером повідомлень Redis дозволяє декумулювати навантаження, перетворюючи його на потік подій у черзі. Згідно з принципами Event-Driven Architecture, це дозволяє масштабувати кількість обробників незалежно від основного додатку [49]. Якщо черга повідомлень зростає швидше, ніж її встигає обробляти один екземпляр Notification Service, система автоматично ініціює запуск додаткових воркерів, які паралельно опрацьовують завдання у фоновому режимі. Це гарантує, що час відгуку основного інтерфейсу для користувача залишається мінімальним навіть при значних фонових навантаженнях.

Підсумовуючи, аналіз архітектури підтверджує, що обраний стек технологій та патерни проектування забезпечують високий потенціал масштабування, недосяжний для традиційних монолітних систем. Незалежне масштабування бізнес-логіки, баз даних та асинхронних процесів дозволяє стверджувати, що розроблена архітектура здатна ефективно адаптуватися до динамічних навантажень, характерних для сучасних цифрових сервісів громадської взаємодії.

3.3 Оцінка аспектів безпеки та надійності мікросервісного підходу

Аналіз потенціалу масштабування довів гнучкість розробленої архітектури. Але для сервісів громадської взаємодії, які оперують даними громадян та мають бути доступні 24/7, не менш важливими є аспекти безпеки та відмовостійкості. Мікросервісний підхід кардинально змінює ландшафт ризиків у порівнянні з монолітом: він вирішує одні проблеми, але водночас створює нові виклики, зокрема пов'язані зі збільшенням «поверхні атаки» через мережеву взаємодію [18]. Цей підрозділ оцінює переваги та недоліки обраної архітектури в контексті безпеки та надійності.

Перехід до розподіленої системи вимагає переосмислення моделі безпеки. Замість захисту єдиного периметра як у моноліті, необхідно захищати комунікацію між десятками сервісів.

Перевагою є ізоляція ризиків. Це ключовий аргумент на користь мікросервісів. В розробленій архітектурі кожен сервіс має власну, ізольовану базу даних. Компрометація одного сервісу не означає автоматичної компрометації всієї системи. Наприклад, якщо зловмисник знайде вразливість у Сервісі сповіщень і отримає доступ до його контейнера, він не зможе отримати доступ до даних користувачів. Найважливіші дані – хеші паролів, email та персональні дані надійно ізольовані в базі даних Сервісу автентифікації. У монолітній системі, отримавши доступ до додатка, зловмисник, скоріш за все, отримав би доступ одразу до всіх таблиць бази даних.

Збільшення кількості мережевих точок взаємодії (API) створює нові вектори атак. У даному прототипі ця проблема вирішується на двох рівнях:

- Зовнішня комунікація: Як було реалізовано в розділі 2, кожен запит від клієнта до API Gateway має бути автентифікований за допомогою JWT-токена. Шлюз виступає охоронцем, який не пропустить неавтентифікований запит до внутрішніх сервісів [50].

- Внутрішня комунікація: У реальному production-проєкті недостатньо довіряти запитам лише тому, що вони прийшли з внутрішньої мережі. Для посилення безпеки впроваджуються додаткові механізми, такі як статичні API-ключі для міжсервісної авторизації або повноцінні Service Mesh (напр., Istio) для забезпечення взаємної автентифікації [51].

Надійність, або відмовостійкість – це здатність системи продовжувати роботу навіть у разі збою одного чи кількох її компонентів.

Перевагою є відсутність єдиної точки відмови (SPOF). У монолітній архітектурі критична помилка в одному, навіть некритичному, модулі (може вичерпати ресурси всього сервера і призвести до повної відмови). У мікросервісній архітектурі збої ізольовані. Якщо Сервіс сповіщень перестане працювати (наприклад, через збій WebSocket-сервера Reverb або відмову брокера черг Redis), це не зупинить роботу всієї платформи. Користувачі все ще зможуть реєструватися, писати пости та залишати коментарі. Discussion Service просто не зможе відправити подію в чергу або вона накопичуватиметься в ній. Система переходить у режим поступової деградації, але ядро функціоналу залишається доступним.

Викликом є складність моніторингу. Якщо в моноліті достатньо було моніторити один додаток, то в мікросервісній системі необхідно відстежувати стан десятків сервісів та їхні мережеві взаємодії. Це вимагає впровадження комплексних систем розподіленого трасування (Distributed Tracing, напр., Jaeger) та централізованого збору логів [18].

Окремо слід зазначити вбудовані механізми безпеки фреймворку Laravel, які були використані для захисту прототипу від найпоширеніших веб-вразливостей. Враховуючи публічний характер Civic Tech платформ, вони є

об'єктом підвищеного інтересу з боку зловмисників, тому захист на рівні коду є обов'язковим:

- Захист від міжсайтової підробки запитів (CSRF). Усі форми, що модифікують стан системи (POST, PUT, DELETE), автоматично захищені CSRF-токенами. Laravel генерує унікальний токен для кожної активної сесії користувача. Спеціальне проміжне програмне забезпечення VerifyCsrfToken перевіряє наявність цього токена в заголовках запиту. Якщо токен відсутній або не співпадає, запит відхиляється зі статусом 419 Page Expired ще до того, як він дійде до контролера.

- Захист від XSS-атак (Cross-Site Scripting). Оскільки платформа передбачає публікацію контенту користувачами, існує ризик впровадження шкідливого JavaScript-коду. Використаний шаблонний рушій Blade автоматично екранує всі дані, що виводяться через конструкцію {{ $variable }}. Це перетворює спеціальні символи HTML на безпечні сутності (наприклад, < на <), роблячи виконання скриптів неможливим.

- Захист від SQL-ін’єкцій. Використання Eloquent ORM забезпечує надійний захист від ін'єкцій на рівні доступу до даних. ORM використовує підготовлені запити (PDO parameter binding), де параметри запиту передаються окремо від самої SQL-команди. Це унеможливлює модифікацію структури SQL-запиту через вхідні дані користувача, навіть якщо вони містять синтаксичні конструкції SQL.

Застосування цих механізмів дозволяє значно знизити ризики людського фактору при написанні коду безпеки та зосередитися на бізнес-логіці захисту.

Обрана мікросервісна архітектура демонструє значні переваги у сфері безпеки та надійності. Ключовою перевагою є ізоляція ризиків та відмовостійкість. Хоча цей підхід і створює нові виклики, вони є керованими за допомогою сучасних інструментів і є прийнятною ціною за досягнення високого рівня надійності та безпеки, що є критичним для громадських цифрових сервісів.

3.4 Рекомендації щодо розгортання, моніторингу та подальшої підтримки системи

Розроблений та протестований у локальному середовищі Docker прототип є фундаментом, але ще не готовим до експлуатації продуктом. Перехід від прототипу до повноцінного хмарного сервісу вимагає впровадження індустріальних практик автоматизації розгортання, всебічного моніторингу та планування подальшої підтримки. Цей підрозділ формулює ключові рекомендації для забезпечення життєвого циклу розробленої мікросервісної системи.

Локальний запуск через docker compose не підходить для робочого середовища. Необхідна автоматизація, яка реалізується через практики Continuous Integration та Continuous Deployment (CI/CD).

Для кожного мікросервісу (Auth, Discussion, Notification) має бути налаштований власний, незалежний конвеєр (pipeline), наприклад, за допомогою GitHub Actions або GitLab CI. Коли розробник вносить зміни у код, наприклад, Сервісу обговорень, CI/CD конвеєр автоматично запускає тести (unit, integration). У разі успіху, збирає новий Docker-образ сервісу і завантажує його у хмарний репозиторій. Дає команду хмарному оркестратору провести оновлення, плавно замінюючи старі контейнери сервісу на нові.

Це дозволяє безпечно та швидко впроваджувати зміни лише в один сервіс, не торкаючись і не зупиняючи решту системи, що є ключовою бізнес-перевагою мікросервісів [52].

Для автоматизації процесу доставки коду рекомендується впровадження конвеєра CI/CD. Схема пропонованого конвеєра для розгортання мікросервісів наведена на рисунку 3.5 та розширено в додатку Д.2.

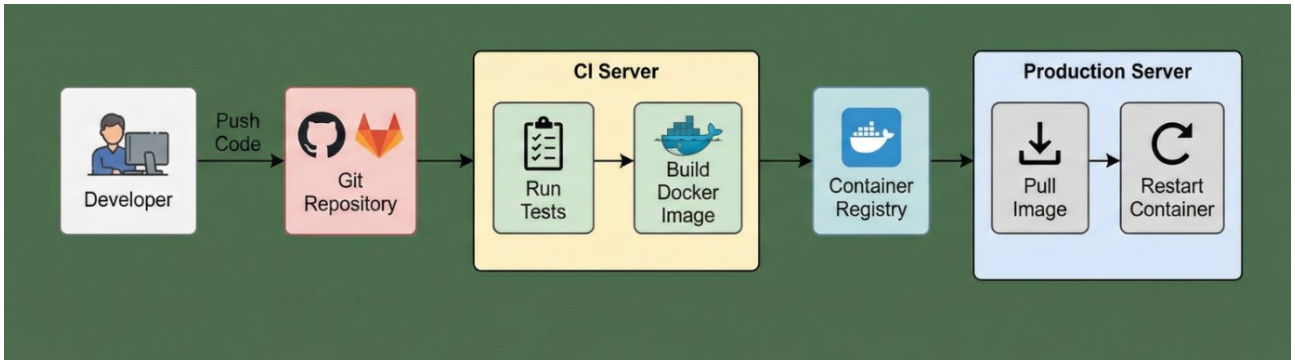


Рисунок 3.5 – Схема автоматизованого конвеєра розгортання (CI/CD Pipeline)

Процес розгортання, зображений на схемі, складається з наступних етапів:

- Commit & Push: розробник відправляє зміни до репозиторію коду.
- Automated Testing: CI-сервер автоматично запускає набір модульних та інтеграційних тестів (PHPUnit). Якщо хоча б один тест не проходить, процес зупиняється, а розробник отримує сповіщення.
- Build & Push: у разі успішного тестування збирається новий Docker-образ, який тегується версією релізу та завантажується у приватний реєстр контейнерів (Container Registry).
- Deploy: оркестратор на продуктивному сервері отримує команду на оновлення. Він завантажує новий образ та виконує заміну контейнерів за стратегією Rolling Update (поступова заміна без простою сервісу).

Така схема мінімізує ризик потрапляння непрацездатного коду в продуктивне середовище та стандартизує процес оновлення всіх мікросервісів системи.

У моноліті помилку можна було знайти в одному файлі логів. Але як знайти причину помилки, якщо запит пройшов через 3 різних сервіси, і кожен з них працює у своєму контейнері? Це головний виклик мікросервісів, який вирішується інструментами спостережуваності [53].

Неприпустимо збирати логи вручну з кожного контейнера. Необхідно впровадити систему, де всі потоки логів з усіх сервісів збираються в єдиному, індексованому сховищі. Популярним рішенням є ELK Stack (Elasticsearch, Logstash, Kibana) або хмарні аналоги, такі як AWS CloudWatch Logs. Це дозволяє аналізувати наскрізний шлях запиту.

- Необхідно відстежувати «здоров'я» кожного сервісу окремо. За допомогою систем, як-от Prometheus (збір метрик) та Grafana (візуалізація), потрібно налаштувати дашборди, що показують:

- Навантаження на ЦП та використання пам'яті кожного контейнера.

- Час відповіді API кожного ендпоїнту.

- Кількість помилок.

- Розмір черг Redis, щоб бачити, чи встигають воркери. Налаштування сповіщень на аномальні показники дозволить реагувати на проблеми до того, як їх помітять користувачі [54].

Спроєктована архітектура є гнучкою для розширення. Подальший розвиток не вимагатиме модифікації існуючих сервісів, а полягатиме у додаванні нових, наприклад:

- Сервіс модерації: замість ускладнення Сервісу обговорень, можна створити окремий Moderation Service. Він буде так само підписаний на подію NewCommentCreated, асинхронно аналізуватиме текст коментаря на наявність спаму чи мови ворожнечі та, у разі потреби, автоматично приховуватиме його або надсилатиме скаргу модератору.

- Сервіс рейтингів або голосування: функціонал вподобань чи дизлайків для постів та коментарів ідеально виноситься в окремий Rating Service з власною базою даних, що уникне блокувань таблиць у головному Discussion Service.

Перетворення розробленого прототипу на надійний продукт вимагає трьох ключових кроків: впровадження CI/CD для автоматизованого та незалежного розгортання, налаштування централізованого моніторингу та логування для повної прозорості системи, та планування розвитку шляхом додавання нових, ізольованих мікросервісів. Виконання цих рекомендацій дозволить повною мірою реалізувати потенціал масштабованості, надійності та гнучкості, закладений у мікросервісну архітектуру.

3.5 Висновок до третього розділу

У межах третього розділу було проведено всебічну оцінку розробленого прототипу мікросервісної системи, що дозволило критично переосмислити архітектурні рішення в контексті вимог, сформульованих на початку дослідження.

Результати верифікації функціональності продемонстрували, що спроєктована логіка взаємодії компонентів є повністю життєздатною. Наскрізне тестування сценаріїв користувача підтвердило коректність роботи повного асинхронного ланцюжка обробки даних: від ініціації події у Discussion Service до її проходження через брокер повідомлень та миттєвої доставки клієнту засобами Notification Service.

Аналіз архітектури підтвердив її відповідність вимогам еластичної масштабованості. На відміну від монолітних рішень, запропонований підхід забезпечує можливість гранулярного горизонтального масштабування. Система дозволяє незалежно нарощувати потужності для бізнес-логіки, реплік баз даних або фонових обробників черг у відповідь на специфічні вектори навантаження, що є критичним для сервісів громадської взаємодії з їх непередбачуваним трафіком.

Окремим важливим результатом дослідження стало підтвердження переваг мікросервісного підходу у площині надійності та безпеки. Архітектурна ізоляція компонентів забезпечує механізм поступової деградації функціоналу, запобігаючи повному краху системи у разі локальних збоїв. Крім того, сегментація даних значно зменшує blast radius у випадку інцидентів кібербезпеки, що є суттєвою перевагою над централізованим зберіганням даних у моноліті.

Водночас аналіз виявив, що прототип, будучи надійним технологічним фундаментом, потребує доопрацювання для переходу в режим промислової експлуатації. Сформульовані рекомендації щодо впровадження автоматизованих конвеєрів розгортання CI/CD та систем централізованої спостережуваності ELK

Stack, Prometheus визначають чітку дорожню карту трансформації прототипу у повноцінний продукт.

Узагальнюючи результати розділу, можна стверджувати, що обраний технологічний стек у поєднанні з мікросервісною архітектурою є ефективним рішенням для побудови сучасних хмарних сервісів. Такий підхід забезпечує необхідний баланс між продуктивністю, гнучкістю та надійністю, створюючи платформу, здатну адаптуватися до зростаючих потреб цифрового суспільства.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Правові та організаційні заходи безпеки при розробці програмного забезпечення

Організація робочого місця розробника хмарних архітектур повинна базуватися на принципах ергономіки та мінімізації ризиків розвитку професійних захворювань. Згідно з Гігієнічною класифікацією праці, робота з розробки програмного забезпечення, яка вимагає зосередженого спостереження за екраном понад 75% робочого часу, відноситься до категорії робіт з підвищеним напруженням зору та нервової системи [55].

Відповідно до НПАОП 0.00-7.15-18, роботодавець зобов'язаний проінформувати працівника під розписку про умови праці та наявність ризиків пошкодження здоров'я [56]. Ключовим елементом безпеки є правильна організація робочого простору.

Вимоги до робочого столу та простору: згідно з п. 3 розділу II Вимог [56], робочий стіл повинен мати поверхню з низькою відбивною здатністю (матова поверхня), щоб уникнути відблисків, які підвищують зорове напруження. Розміри столу повинні дозволяти гнучке розміщення екрана, клавіатури, документів і допоміжного обладнання.

Висота робочої поверхні повинна бути регульованою або становити 725 мм (± 15 мм) для нерегульованих столів. Простір для ніг повинен мати глибину не менше 600 мм та ширину не менше 500 мм.

Вимоги до робочого крісла: оскільки робота розробника є переважно статичною (сидячою), крісло має відповідати вимогам стандарту ДСТУ EN ISO 9241-5 (Ергономічні вимоги до роботи з відеотерміналами) [57]. Робочий стілець повинен бути стійким і дозволяти працівнику вільно рухатися та займати зручну позу. Обов'язковими є:

- Регулювання висоти сидіння в межах 400–550 мм.
- Наявність опори для попереку (пасивна або регульована підтримка).

- Механізм нахилу спинки, що дозволяє динамічне сидіння (синхромеханізм) для запобігання застою кровообігу в органах малого тазу.

Освітлення є критичним фактором для профілактики комп'ютерного зорового синдрому. Проєктування освітлення здійснюється відповідно до ДБН В.2.5-28:2018 «Природне і штучне освітлення» [58].

Для робіт високої точності, до яких відноситься написання та налагодження програмного коду, встановлено наступні нормативи [58]:

- Освітленість робочої поверхні: не менше 500 лк. Рекомендується комбінована система освітлення (загальне + місцеве), проте місцеві джерела світла не повинні створювати відблисків на екрані монітора.

- Коефіцієнт пульсації: не більше 10%. Використання сучасних LED-світильників з якісними драйверами дозволяє знизити цей показник до 1–5%, що є безпечним для зору.

- Обмеження засліплювальної дії: показник дискомфорту (UGR) не повинен перевищувати 19.

Сучасний підхід, закріплений у НПАОП 0.00-7.15-18, відходить від жорсткого розкладу перерв на користь гнучкого регулювання [55]. Встановлено, що працівник має право на перерви для відпочинку та особистих потреб, а також на перерви для виконання вправ з метою зняття зорового та м'язового напруження. Рекомендується застосовувати методику мікропауз: 10-15 хвилин відпочинку після кожних 90-120 хвилин інтенсивної роботи. Під час перерв ефективним є виконання вправ для акомодації ока (фокусування на віддалених предметах) для профілактики міопії.

4.2 Санітарно-гігієнічні вимоги до мікроклімату та психофізіологічні аспекти праці

Забезпечення належних параметрів мікроклімату та врахування психофізіологічних особливостей інженерної праці є передумовою високої продуктивності та відсутності помилок у коді, що особливо важливо при роботі з критичними даними громадян.

Параметри мікроклімату в приміщеннях з комп'ютерною технікою регламентуються ДБН В.2.5-67:2013 «Опалення, вентиляція та кондиціонування» [59] та гігієнічними нормативами. Оскільки робота програміста відноситься до категорії легкої фізичної праці (категорія Іа), оптимальними є наступні параметри:

- Температура повітря: у холодний період року – 22-24 °С, у теплий період – 23-25 °С.
- Відносна вологість: 40-60%. Підтримання вологості не нижче 40% є важливим для запобігання пересиханню слизової оболонки очей, що посилюється при рідкому морганні під час роботи за монітором.
- Швидкість руху повітря: не більше 0.1 м/с.

Система вентиляції повинна забезпечувати достатній повітрообмін для видалення надлишків тепла від обладнання (системні блоки, сервери, монітори) та вуглекислого газу, що видихається людьми.

Робота розробника відноситься до категорії висококваліфікованої розумової праці, що вимагає концентрації уваги та аналітичного мислення. Акустичний дискомфорт є одним із головних стрес-факторів, що знижує продуктивність. Відповідно до ДБН В.1.1-31:2013 «Захист територій, будинків і споруд від шуму», допустимий еквівалентний рівень звуку в приміщеннях для конструкторської та проєктної діяльності (до яких прирівнюється розробка ПЗ) становить 50 дБА [60].

Основними джерелами шуму в ІТ-інфраструктурі є активні системи охолодження. Для забезпечення акустичного комфорту в проєкті реалізовано:

- Архітектурне рішення: винесення обчислювальних потужностей у хмару (модель IaaS) повністю усуває шум серверних кулерів з робочої зони.
- Апаратне рішення: робочі станції оснащено системами охолодження з низьким рівнем шуму (< 25 дБА) та твердотільними накопичувачами (SSD), які працюють безшумно на відміну від HDD.

Електромагнітна безпека гарантується використанням моніторів, сертифікованих за міжнародним екологічним стандартом TCO Certified Generation 9 (або новіше). Такі пристрої мають вбудований захист від

електромагнітних випромінювань, а напруженість електростатичного поля на робочому місці не перевищує фонових значень, безпечних для здоров'я людини.

Найбільш суттєвим шкідливим фактором у роботі IT-спеціаліста є психоемоційне напруження. Сучасний міжнародний стандарт ISO 45003:2021 «Управління охороною здоров'я та безпекою праці – Психологічне здоров'я та безпека на роботі» вводить поняття психосоціальних ризиків [61].

У контексті теми кваліфікаційної роботи (мікросервісна архітектура) виділяються наступні фактори ризику:

- Когнітивне перевантаження (Cognitive Overload): необхідність одночасно контролювати роботу багатьох незалежних сервісів, аналізувати розподілені логи та метрики.

- Технострес (Technostress): стан адаптаційного напруження через необхідність постійного опанування нових технологій (фреймворки Laravel, Docker, Kubernetes) та високу швидкість втрати актуальності знань [61].

- Відповідальність: ризики втрати даних громадян або простою сервісу громадської взаємодії створюють постійний тиск.

Заходами профілактики є атоматизація рутинних процесів через впровадження CI/CD пайплайнів (описаних у Розділі 3), що знижує ризик людської помилки та зменшує стрес при розгортанні оновлень. Архітектура мікросервісів дозволяє інженеру фокусуватися на одному модулі, не розпорошуючи увагу на всю систему одночасно. Використання зручних інтерфейсів моніторингу (Grafana dashboard) знижує сенсорне навантаження при аналізі стану системи.

4.3 Забезпечення безпеки та стійкості інфраструктури в надзвичайних ситуаціях

В умовах воєнного стану та перманентної загрози ракетних обстрілів, а також нестабільності об'єднаної енергосистеми, питання безпеки виходить за межі класичної охорони праці. Воно трансформується у комплексну стратегію забезпечення життєздатності персоналу та цифрової стійкості інфраструктури.

Правовою основою для розробки заходів безпеки є Кодекс цивільного захисту України [62], Закон України «Про основні засади забезпечення кібербезпеки України» та галузеві правила пожежної безпеки.

4.3.1 Пожежна безпека та засоби пожежогасіння

Приміщення, де розташовані робочі місця розробників та комп'ютерна техніка, відносяться до категорії пожежної небезпеки В (пожежонебезпечні), а можливі пожежі класифікуються як клас Е (горіння електроустановок під напругою).

Відповідно до «Правил пожежної безпеки в Україні» (НАПБ А.01.001-2014) [63], для таких приміщень встановлено суворі вимоги:

- Приміщення повинні бути обладнані автоматичними системами пожежної сигналізації з димовими сповіщувачами, що реагують на появу продуктів горіння до появи відкритого полум'я.

- Як первинні засоби дозволено використовувати вуглекислотні (ВВК) або хладонові вогнегасники.

Категорично заборонено використовувати пінні або водні вогнегасники для гасіння електроніки під напругою. Вода є провідником електричного струму, що призведе до короткого замикання, остаточного знищення дороговартісного обладнання та ризику ураження персоналу електричним струмом [63].

Розрахунок необхідної кількості первинних засобів пожежогасіння:Кількість вогнегасників визначається фізико-хімічними властивостями горючих речовин та площею приміщення. Для приміщень з ПЕОМ (персональними електронно-обчислювальними машинами) норма становить один вогнегасник ВВК-3.5 (маса заряду 3.5 кг) на 20 $м^2$ площі підлоги.

4.3.2 Алгоритм дій персоналу під час сигналу «Повітряна тривога»

Пріоритетом номер один є фізична безпека працівників. Згідно з рекомендаціями ДСНС України та вимогами Кодексу цивільного захисту [62], розроблено наступний алгоритм дій для розробника при отриманні сигналу «Повітряна тривога»:

- Отримати сигнал через офіційні канали комунікації (міська система оповіщення, мобільний додаток «Повітряна тривога», радіо).

- Негайно припинити роботу.

- Якщо виконується критична операція (наприклад, деплой на продуктивний сервер або транзакція бази даних), її слід перервати.

- Якщо завершення операції займає менше 30 секунд (наприклад, git commit локальних змін), дозволяється її завершити в аварійному режимі.

- Заблокувати робочу станцію (Win+L / Cmd+Ctrl+Q) або забрати ноутбук із собою. Це запобігає несанкціонованому доступу до коду та даних у разі пошкодження офісу або проникнення сторонніх осіб.

- Вимкнути комп'ютер від електромережі (витягнути вилку з розетки) для захисту блоку живлення від можливих стрибків напруги внаслідок пошкодження електромереж. Перекрити газові та водяні крани (за наявності).

- Взяти заздалегідь підготовлений «тривожний рюкзак» (документи, аптечка, вода, павербанк) та пройти до найближчого укриття (бомбосховища, паркінгу або споруди подвійного призначення).

Вимоги до укриття для продовження роботи: Згідно з ДБН В.2.2-5:2023 «Захисні споруди цивільного захисту» [64], для забезпечення безперервності роботи укриття повинно мати два незалежні виходи, примусову вентиляцію, резервне джерело живлення та стабільний канал зв'язку (Wi-Fi/Ethernet). Робота продовжується тільки за умови знаходження працівника у безпечній зоні.

4.3.3 Енергетична безпека та безперервність бізнесу (BCP)

В умовах ризику повного знеструмлення, стратегія Business Continuity Planning (BCP) для Civic Tech платформи базується на трьох рівнях:

1. Хмарна архітектура та географічне резервування: система спроєктована за моделлю Cloud-Native. Усі сервери, бази даних та черги повідомлень розгорнуті у хмарних провайдерів (AWS/DigitalOcean) у регіонах, що знаходяться за межами зони бойових дій (наприклад, eu-central-1 Frankfurt). Це гарантує, що фізичне знищення або знеструмлення офісу розробника не вплине на доступність сервісу для кінцевих користувачів.

2. Енергетична автономність робочого місця: для забезпечення коректного завершення роботи та підтримки зв'язку під час відключень, робоче місце інженера має бути обладнане джерелом безперебійного живлення.

3. Резервування каналів зв'язку: критичним для Civic Tech є безперервний доступ до мережі Інтернет. Реалізовано схему резервування:

- Основний канал: оптоволоконна лінія GPON, яка є енергонезалежною, працює при знеструмленні будинку, якщо заживити ONU-термінал від павербанка.
- Резервний канал: мобільний інтернет 4G/LTE через USB-модем або точку доступу.
- Аварійний канал: супутниковий зв'язок Starlink для критичних ситуацій при пошкодженні магістральних ліній.

У разі використання паливних генераторів для резервного живлення, суворо дотримуватися правил безпеки: генератор розміщувати тільки на відкритому повітрі на відстані не менше 6 метрів від вікон та дверей для уникнення отруєння чадним газом. Заборонено об'єднувати клеми генератора з будинковою мережею без використання перекидного рубильника, щоб уникнути подачі зворотної напруги в загальну мережу, що є смертельно небезпечним для електриків, які відновлюють лінії.

4.4 Висновок до четвертого розділу

У четвертому розділі кваліфікаційної роботи здійснено аналіз умов праці розробника програмного забезпечення та обґрунтовано комплекс заходів із забезпечення безпеки життєдіяльності.

Встановлено, що організація робочого місця відповідно до вимог НПАОП 0.00-7.15-18 та дотримання санітарно-гігієнічних норм мінімізують вплив шкідливих виробничих факторів, зокрема зорового та нервово-емоційного напруження. Визначено, що використання сучасних інструментів розробки (CI/CD, мікросервіси) сприяє зниженню рівня техностресу та когнітивного навантаження.

Розроблено алгоритм дій персоналу в надзвичайних ситуаціях та стратегію забезпечення безперервності роботи в умовах енергетичної нестабільності, що базується на використанні хмарних технологій та автономного живлення. Запропоновані організаційні та технічні рішення гарантують створення безпечних умов праці, збереження здоров’я фахівця та стійкість цифрової інфраструктури до зовнішніх загроз.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальну науково-прикладну задачу забезпечення еластичної масштабованості, надійності та інтерактивності цифрових сервісів громадської взаємодії (Civic Tech). Це досягнуто шляхом розробки та дослідження хмарної мікросервісної архітектури на базі екосистеми фреймворку Laravel, що дозволяє нівелювати ризики пікових навантажень та покращити досвід користувачів.

В першому розділі кваліфікаційної роботи освітнього рівня «Магістр»:

- Подано аналіз концепції цифрової громадської взаємодії, визначено її форми, цілі та соціальні виклики, зокрема проблему цифрової нерівності.
- Розглянуто класифікацію існуючих цифрових платформ (інформування, обговорення, прийняття рішень) та їхню роль у сучасному демократичному суспільстві.
- Висвітлено критичні технічні та функціональні вимоги до високонавантажених сервісів, серед яких ключовими є висока доступність, еластичність та безпека даних.
- Проаналізовано архітектурні патерни веб-додатків, проведено порівняння монолітного та мікросервісного підходів, доведено переваги останнього для Civic Tech проєктів.
- Досліджено особливості хмарних технологій (IaaS, PaaS) як середовища розгортання та переваги контейнеризації для ізоляції сервісів.
- Обґрунтовано вибір фреймворку Laravel та його екосистеми (Sail, Queues, Reverb) як оптимального інструментального засобу розробки.

В другому розділі кваліфікаційної роботи:

- Розроблено компонентну архітектуру системи, що складається з незалежних мікросервісів (Автентифікації, Обговорень, Сповіщень) та визначено логіку їхньої взаємодії.
- Спроєктовано RESTful API-контракти та ізольовані схеми баз даних для кожного сервісу згідно з принципом Database-per-Service.

- Описано налаштування локального середовища розробки на базі Docker та Laravel Sail, що імітує реальну хмарну інфраструктуру.

- Подано програмну реалізацію механізмів асинхронної обробки подій через черги повідомлень та оновлень інтерфейсу в реальному часі за допомогою WebSockets.

В третьому розділі кваліфікаційної роботи:

- Протестовано працездатність прототипу та верифіковано коректність наскрізної взаємодії компонентів у ключових сценаріях користувача.

- Проаналізовано потенціал горизонтального масштабування розробленої архітектури в умовах пікових навантажень.

- Оцінено аспекти безпеки та надійності системи, зокрема переваги ізоляції даних та відмовостійкості окремих сервісів.

- Сформовано рекомендації щодо подальшого розгортання системи з використанням практик CI/CD, централізованого логування та моніторингу.

У розділі «Охорона праці та безпека в надзвичайних ситуаціях» обґрунтовано комплекс заходів із забезпечення безпеки праці розробників програмного забезпечення відповідно до сучасних нормативів. Проаналізовано вплив психофізіологічних факторів, зокрема техностресу та когнітивного навантаження, і визначено роль автоматизації процесів у їх мінімізації. Розроблено стратегію забезпечення безперервності роботи в умовах воєнного стану та енергетичної нестабільності, що базується на використанні хмарної архітектури, географічному резервуванні даних та енергетичній автономності робочих місць.

ПЕРЕЛІК ДЖЕРЕЛ

1. Ali, M. (2023). E-governance and E-democracy: A digital revolution. *SSRN.* https://doi.org/10.2139/ssrn.4623414.

2. United Nations Department of Economic and Social Affairs. (2024). *E-Government survey 2024: Accelerating digital transformation for sustainable development.* https://desapublications.un.org/sites/default/files/publications/2024-09/(Web%20version)%20E-Government%20Survey%202024%201392024.pdf.

3. Zisengwe, M. (2024). Intersections between civic technology (civic tech) and governance in Nigeria and South Africa. *The African Journal of Information and Communication, 33*, 1–26. https://doi.org/10.23962/ajic.i33.18883.

4. Grigalashvili, V. (2023). Digital government and digital governance: Grand concept. *International Journal of Scientific and Management Research, 6*, 1–25. https://doi.org/10.37502/IJSMR.2023.6201.

5. OECD. (2020*). The OECD Digital Government Policy Framework: Six dimensions of a digital government* (OECD Public Governance Policy Papers, No. 02). OECD Publishing. https://doi.org/10.1787/f64fed2a-en.

6. Eurofound. (2025). *Narrowing the digital divide: Economic and social convergence in Europe's digital transformation.* Publications Office of the European Union. https://www.eurofound.europa.eu/en/publications/all/narrowing-digital-divide-economic-and-social-convergence-europes-digital.

7. OECD. (2024*). Facts not fakes: Tackling disinformation, strengthening information integrity.* OECD Publishing. https://doi.org/10.1787/d909ff7a-en

8. UNESCO. (2023). *Guidelines for the governance of digital platforms: Safeguarding freedom of expression and access to information through a multi-stakeholder approach.* https://unesdoc.unesco.org/ark:/48223/pf0000387339

9. Липак, Г. І., & Кунанець, Н. Е. (2023). Проект платформи для консолідованого інформаційного ресурсу територіальної громади. *Бібліотека. Наука. Комунікація: актуальні тенденції у цифрову епоху.*

10. Kunanets, N., & Lypak, H. (2017). Consolidated information resources of social memory institutions: Challenges of modernity. *Вісник Національної академії керівних кадрів культури і мистецтв, (3)*, 21–25.

11. OECD. (2025). *Government at a glance 2025: Open government data.* OECD Publishing. https://www.oecd.org/en/publications/government-at-a-glance-2025_0efd0bcd-en/full-report/open-government-data_619b668c.html

12. Linders, D. (2012). From E-Government to We-Government: Defining a Typology for Citizen Coproduction in the Age of Social Media. *Government Information Quarterly, 29*, 446-454. https://doi.org/10.1016/j.giq.2012.06.003

13. Cinelli, M., De Francisci Morales, G., Galeazzi, A., Quattrociocchi, W., & Starnini, M. (2021). The echo chamber effect on social media. *Proceedings of the National Academy of Sciences, 118(9),* e2023301118.

14. Lindner, Ralf & Ulrich, Riehm. (2009). Electronic Petitions and Institutional Modernization. International Parliamentary E-Petition Systems in Comparative Perspective. *eJournal of eDemocracy & Open Government. 1.* 10.29379/jedem.v1i1.3.

15. Meijer, A., & Bolívar, M. P. R. (2016). Governing the smart city: A review of the literature on smart urban governance. *International Review of Administrative Sciences, 82(2)*, 392–408.

16. Twizeyimana, J. D., & Andersson, A. (2019). The public value of E-Government: A literature review. *Government Information Quarterly, 36(2)*, 167–178. https://doi.org/10.1016/j.giq.2019.01.001

17. Richards, M., & Ford, N. (2020). *Fundamentals of software architecture: An engineering approach*. O'Reilly Media.

18. Newman, S. (2021). *Building microservices: Designing fine-grained systems (2nd ed.)*. O'Reilly Media.

19. Marinescu, D. C. (2022). *Cloud computing: Theory and practice (3rd ed.).* Morgan Kaufmann.

20. Nykytyuk, V., Dozorskyi, V., Dozorska, O., Karnaukhov, A., & Matiichuk, L. (2022). The method of user identification by speech signal. *Proceedings of the 2nd International Workshop on Information Technologies: Theoretical and Applied Problems (ITTAP 2022)*, 225–232.

21. Janssen, M., Brous, P., Estevez, E., Barbosa, L. S., & Janowski, T. (2020). Data governance: Organizing data for value creation and reliance. *Government Information Quarterly, 37(3)*, 101483. https://doi.org/10.1016/j.giq.2020.101493/

22. Biørn-Hansen, A., Majchrzak, T. A., & Grønli, T.-M. (2018). Progressive web apps for the unified development of mobile applications. *In Web information systems and technologies* (pp. 64–86). Springer. https://doi.org/10.1007/978-3-319-93527-0_4

23. Arapakis, I., Park, S., & Pielot, M. (2021). Impact of response latency on user behaviour in mobile web search. *arXiv.* https://doi.org/10.48550/arXiv.2101.09086

24. Acosta-Vargas, P., Luján-Mora, S., & Salvador-Ullauri, L. (2022). Accessibility analysis of worldwide COVID-19-related information portals. *International Journal of Environmental Research and Public Health,* 19(19), 12102. https://doi.org/10.3390/ijerph191912102

25. Bass, L., Clements, P., & Kazman, R. (2021). *Software architecture in practice* (4th ed.). Addison-Wesley Professional.

26. Fowler, M. (2014). *Microservices.* martinfowler.com. https://martinfowler.com/articles/microservices.html

27. Gartner. (2023). *2024 planning guide for application architecture, integration and platforms.*

28. Microsoft. (2024). *Microservices architecture style*. Microsoft Learn. https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

29. Lypak, H., Rzheuskyi, A., Kunanets, N., & Pasichnyk, V. (2018). Formation of a consolidated information resource by means of cloud technologies. *2018 International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T)*, 635–640.

30. Mell, P., & Grance, T. (2011). *The NIST definition of cloud computing* (NIST Special Publication 800-145). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-145

31. Alharthi, S., Alshamsi, A., Alseiari, A., & Alwarafy, A. (2024). Auto-scaling techniques in cloud computing: Issues and research directions. *Sensors, 24*(17), 5551. https://doi.org/10.3390/s24175551

32. Microsoft. (2024). *Regions and availability zones in Azure*. https://learn.microsoft.com/en-us/azure/availability-zones/az-overview

33. Attaran, M., & Woods, J. (2019). Cloud computing technology: improving small business performance using the Internet. *Journal of Small Business & Entrepreneurship, 31(6),* 495–519. https://doi.org/10.1080/08276331.2018.1466850

34. Laravel. (2025). *Queues.* Laravel documentation. https://laravel.com/docs/12.x/queues

35. Laravel. (2025). *Events.* Laravel documentation. https://laravel.com/docs/12.x/events

36. Laravel. (2025). *Laravel Reverb*. Laravel documentation. https://laravel.com/docs/12.x/reverb

37. Laravel. (2025). *Laravel Sail.* Laravel documentation. https://laravel.com/docs/12.x/sail

38. Kunanets, N., Pasichnyk, V., Lypak, H., Duda, O., & Matsiuk, O. (2017). Modeling of consolidated information resource for social data institutions. *ECONTECHMOD: An International Quarterly Journal on Economics of Technology, 6(2)*, 53–60.

39. Martin, R. C. (2017). *Clean architecture: A craftsman's guide to software structure and design.* Prentice Hall.

40. Kryazhych, O., Itskovych, V., Iushchenko, K., Hrytsyshyna, V., Bruvier, D., et al. (2023). The use of abstract Moore automaton to control the sensors of a service-oriented alarm and emergency notification network.

41. Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* [Doctoral dissertation, University of California, Irvine]

42. Pacheco, V. F. (2018). *Microservice patterns and best practices.* Packt Publishing.

43. Humble, J., & Farley, D. (2010). *Continuous delivery: Reliable software releases through build, test, and deployment automation.* Addison-Wesley.

44. Docker Inc. (2025). *Docker overview*. Docker Documentation. https://docs.docker.com/get-started/overview/

45. Stauffer, M. (2023). *Laravel: up and running: A framework for building modern PHP apps (3rd ed.)*. O'Reilly Media.

46. Blinowski, G., Ojdowska, A., & Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access, 10*, 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803

47. Burns, B., Beda, J., & Hightower, K. (2022). *Kubernetes: Up and running: Dive into the future of infrastructure* (3rd ed.). O'Reilly Media.

48. Özsu, M. T., & Valduriez, P. (2020). *Principles of distributed database systems* (4th ed.). Springer. https://doi.org/10.1007/978-3-030-26253-2

49. Bellemare, A. (2020). *Building event-driven microservices: Leveraging organizational data at scale*. O'Reilly Media.

50. Siriwardena, P., & Dias, N. (2020). *Microservices security in action*. Manning Publications

51. Chandramouli, R. (2019*). Security strategies for microservices-based application systems* (NIST Special Publication 800-204). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-204

52. Arundel, J., & Domingus, J. (2022). *Cloud native DevOps with Kubernetes: Building, deploying, and scaling modern applications in the cloud* (2nd ed.). O'Reilly Media.

53. Majors, C., Fong-Jones, L., & Miranda, G. (2022*). Observability engineering: Achieving production excellence.* O'Reilly Media.

54. Кліщ, М., Липак, Г., Кунанець, Н., Пасічник, С., & Липак, Т. (2025). *Структура інформаційної системи передбачення та інтерпретації зміни стану користувача сервісу*.

55. Міністерство соціальної політики України. (2018). *Вимоги до безпеки та захисту здоров'я працівників під час роботи з екранними пристроями* (Наказ № 207). https://zakon.rada.gov.ua/laws/show/z0458-18.

56. Міністерство охорони здоров'я України. (2014). *Гігієнічна класифікація праці за показниками шкідливості та небезпечності факторів виробничого*

середовища, важкості та напруженості трудового процесу (Наказ № 248). https://zakon.rada.gov.ua/laws/show/z0472-14

57. ДП «УкрНДНЦ». (2004). *Ергономічні вимоги до роботи з відеотерміналами в офісі. Частина 5. Вимоги до розташування робочого місця та пози* (ДСТУ EN ISO 9241-5:2004).

58. Міністерство регіонального розвитку, будівництва та житлово-комунального господарства України. (2018). *Природне і штучне освітлення* (ДБН В.2.5-28:2018). Інформаційний бюлетень Мінрегіону.

59. Міністерство регіонального розвитку, будівництва та житлово-комунального господарства України. (2013). *Опалення, вентиляція та кондиціонування* (ДБН В.2.5-67:2013).

60. Міністерство регіонального розвитку, будівництва та житлово-комунального господарства України. (2014). *Захист територій, будинків і споруд від шуму* (ДБН В.1.1-31:2013).

61. International Organization for Standardization. (2021). *Occupational health and safety management – Psychological health and safety at work – Guidelines for managing psychosocial risks* (ISO 45003:2021). https://www.iso.org/standard/64283.html

62. Верховна Рада України. (2012). *Кодекс цивільного захисту України* (Закон № 5403-VI). Відомості Верховної Ради України. https://zakon.rada.gov.ua/laws/show/5403-17

63. Міністерство внутрішніх справ України. (2014). *Правила пожежної безпеки в Україні* (Наказ № 1417). https://zakon.rada.gov.ua/laws/show/z0252-15

64. Міністерство розвитку громад, територій та інфраструктури України. (2023). *Захисні споруди цивільного захисту* (ДБН В.2.2-5:2023).

ДОДАТКИ

Додаток А.1

Тези конференції

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Тернопільський національний технічний університет імені Івана Пулюя
Мариборский университет (Словенія)
Технічний університет у Кошице (Словаччина)
Вільнюський технічний університет ім. Гедимінаса (Литва)
Краківський економічний університет (Польща)
Вроцлавський економічний університет (Польща)
Університет «Опольська Політехніка» (Польща)
Національний університет «Полтавська політехніка імені Юрія Кондратюка»
Вінницький національний аграрний університет
Львівський національний університет ім. І. Франка
Головне управління Пенсійного фонду в Тернопільській області
Наукове товариство ім. Шевченка
Тернопільський обласний комунальний інститут післядипломної педагогічної освіти
Сумський державний педагогічний університет
Запорізький національний університет

АКТУАЛЬНІ ЗАДАЧІ СУЧАСНИХ ТЕХНОЛОГІЙ

Збірник

тез доповідей

XIV Міжнародної науково-технічної конференції молодих учених та студентів

11-12 грудня 2025 року

УКРАЇНА
ТЕРНОПІЛЬ – 2025

УДК 004.75
М. В. Лісовий, Г. І. Липак, к.соц.ком., доц.
(Тернопільський національний технічний університет імені Івана Пулюя)

ПРОЄКТУВАННЯ ЕЛАСТИЧНИХ ХМАРНИХ АРХІТЕКТУР ДЛЯ СТІЙКОСТІ ЦИФРОВИХ СЕРВІСІВ ГРОМАДСЬКОЇ ВЗАЄМОДІЇ

M. V. Lisovyi, H. I. Lypak, Ph.D, Assoc. Prof.

DESIGNING ELASTIC CLOUD ARCHITECTURES FOR RESILIENCE DIGITAL PUBLIC INTERACTION SERVICES

Цифрові сервіси громадської взаємодії (Civic Tech), такі як системи електронних петицій чи громадські бюджети, стають ключовим інструментом сучасної демократії. Однак їхня ефективність напряму залежить від продуктивності та стабільності платформи, що, в свою чергу, формує довіру громадян до е-урядування [1].

Ключовою проблемою таких систем є непередбачуваний, «вірусний» характер навантаження. Платформа може місяцями працювати з низьким трафіком, але в момент резонансної події отримати експоненціальне зростання запитів. Традиційні монолітні архітектури демонструють низьку ефективність у таких умовах. Вони вразливі до «єдиної точки відмови» (SPOF), де збій одного модуля призводить до відмови всієї системи, а масштабування вимагає дороговартісного клонування всього додатку [2].

Як оптимальне рішення пропонується мікросервісна архітектура, розгорнута у хмарному (Cloud-Native) середовищі. Такий підхід дозволяє вирішити дві фундаментальні проблеми. По-перше, він забезпечує гранулярне масштабування: хмарні оркестратори, зокрема Kubernetes, дозволяють автоматично масштабувати лише ті сервіси, що зазнають пікового навантаження, не витрачаючи ресурси на решту системи [3]. По-друге, досягається висока відмовостійкість через ізоляцію збоїв. Збій некритичного сервісу не вплине на ядро системи, забезпечуючи «поступову деградацію» (graceful degradation) функціоналу замість повного колапсу [4].

Технологічна спроможність урядових платформ підтримувати відкритість, надійність і своєчасність обробки запитів є ключовим чинником підвищення довіри до цифрового урядування [5]. Архітектурна трансформація від монолітів до хмарних мікросервісів є не лише технічним, але й соціально значущим кроком, що забезпечує стійкість інфраструктури та зміцнює легітимність демократичних процесів у цифрову епоху.

Література

1. Janssen, M., & van der Voort, H. (2016). Adaptive governance: Towards a stable, accountable and responsive government. Government Information Quarterly, 33(1), 1–5. https://doi.org/10.1016/j.giq.2016.02.003
2. Newman, S. (2021). Building microservices: Designing fine-grained systems (2nd ed.). O'Reilly Media.
3. Cloud Native Computing Foundation. (2023). CNCF annual survey 2023. Retrieved from https://www.cncf.io/reports/cncf-annual-survey-2023/
4. Richards, M., & Ford, N. (2020). Fundamentals of software architecture: An engineering approach. O'Reilly Media.
5. Mergel, I., Edelmann, N., & Haug, N. (2019). Defining digital transformation: Results from expert interviews. Government Information Quarterly, 36(4), 101385. https://doi.org/10.1016/j.giq.2019.06.002

Додаток А.2

Тези конференції

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ПУЛЮЯ

М А Т Е Р І А Л И

XIII НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

17-18 грудня 2025 року

ТЕРНОПІЛЬ
2025

УДК 004.451
М. Лісовий
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

РЕАЛІЗАЦІЯ АСИНХРОННИХ ТА REAL-TIME МЕХАНІЗМІВ У СЕРВІСАХ ГРОМАДСЬКОЇ ВЗАЄМОДІЇ НА БАЗІ ФРЕЙМВОРКУ LARAVEL

UDC 004.451
M. Lisovyi

IMPLEMENTATION OF ASYNCHRONOUS AND REAL-TIME MECHANISMS IN PUBLIC INTERACTION SERVICES BASED ON THE LARAVEL FRAMEWORK

Сучасні користувачі цифрових сервісів очікують миттєвого зворотного зв'язку, де будь-яка затримка відповіді інтерфейсу понад одну секунду помітно погіршує досвід користувача (UX) [1]. Для сервісів громадської взаємодії це є критичним. Неприпустимим є блокування інтерфейсу під час виконання довготривалих операцій або необхідність оновлювати сторінку, щоб побачити нові коментарі чи результати голосування. Синхронна обробка запитів у високонавантажених системах створює негативний UX та неефективно використовує обчислювальні ресурси.

Для вирішення цієї задачі пропонується підхід, що комбінує асинхронну обробку та оновлення у реальному часі на базі екосистеми фреймворку Laravel. Для усунення блокувань інтерфейсу, операції, що не потребують миттєвого завершення, делегуються у фоновий режим за допомогою системи черг Laravel Queues на базі брокера повідомлень, такого як Redis [2]. Наприклад, при додаванні коментаря, головний сервіс миттєво повертає користувачу відповідь «201 Created», а завдання на сповіщення відправляє у чергу. Окремий сервіс-воркер асинхронно виконує це завдання, забезпечуючи повне декаплінгування (decoupling) компонентів.

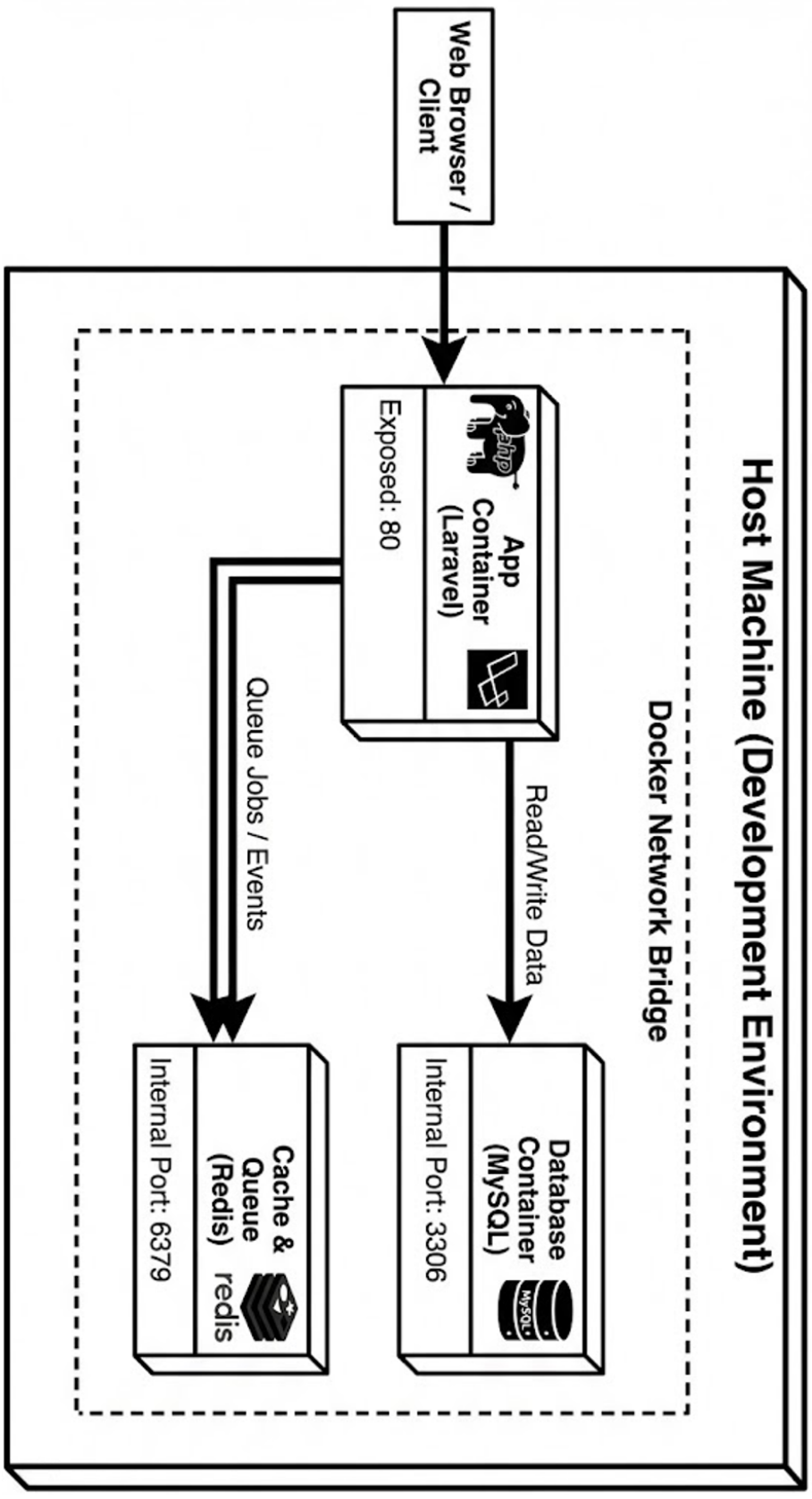
Для миттєвої доставки оновлень клієнтам використано технологію WebSocket, реалізовану через сервер Laravel Reverb [3]. Цей сервер дозволяє воркеру після обробки завдання з черги транслювати (broadcasts) подію у приватний канал. Клієнтські додатки, підписані на цей канал за допомогою Laravel Echo, отримують дані та миттєво оновлюють DOM-структуру сторінки без її перезавантаження. На відміну від традиційного запит-орієнтованого HTTP, WebSocket встановлює постійний двонаправлений канал зв'язку, що є критичним для додатків реального часу, оскільки мінімізує затримку при передачі даних [4]. Тестування прототипу довело, що дана технологічна зв'язка є потужним та готовим до хмарного розгортання рішенням для побудови сучасних інтерактивних сервісів громадської взаємодії.

Література

1. J. Nielsen. (2012). Response times: The 3 important limits. Nielsen Norman Group. https://www.nngroup.com/articles/response-times-3-important-limits/
2. Laravel. (2025). Laravel 12 documentation: Queues. https://laravel.com/docs/12.x/queues
3. Laravel. (2025). Laravel 12 documentation: Laravel Reverb. https://laravel.com/docs/12.x/reverb
4. Murley, P., Ma, Z., Mason, J., Bailey, M., & Kharraz, A. (2021). WebSocket adoption and the landscape of the real-time web. In Proceedings of the Web Conference 2021 (WWW '21) (12 pages). https://doi.org/10.1145/3442381.3450063

Додаток Б

Схема розгортання компонентів системи у контейнеризованому середовищі



Додаток В

Лістинг В.1 – Конфігурація маршрутів (файл routes/api.php)

```
Route::middleware('auth:api')->group(function () {
    // Реєстрація ресурсу 'posts' за винятком методів HTML-форм
    Route::resource('posts', PostController::class)
         ->except(['create', 'edit']);

    // Вкладений ресурс для коментарів (posts.comments)
    Route::resource('posts.comments', CommentController::class)
         ->only(['index', 'store']);
});
```

Лістинг В.2 – Реалізація моделі Post (файл app/Models/Post.php)

```
class Post extends Model
{
    protected $fillable = ['user_id', 'title', 'content'];

    // Зв'язок "Один-до-Багатьох"
    public function comments(): HasMany
    {
        return $this->hasMany(Comment::class);
    }
}
```

Лістинг В.3 – Метод створення публікації (файл PostController.php)

```
public function store(Request $request): JsonResponse
{
    $validated = $request->validate([
        'title'   => 'required|string|max:255',
        'content' => 'required|string',
    ]);

    $post = Post::create([
        'user_id' => Auth::id(),
        'title'   => $validated['title'],
        'content' => $validated['content'],
    ]);

    return response()->json($post, 201);
}
```

Лістинг В.4 – Генерація події при коментуванні (файл CommentController.php)

```
public function store(Request $request, Post $post): JsonResponse
{
    $validated = $request->validate(['content' => 
'required|string']);

    $comment = $post->comments()->create([
        'user_id' => Auth::id(),
        'content' => $validated['content'],
    ]);

    // Генерація доменної події для асинхронної обробки
    NewCommentCreated::dispatch($comment);

    return response()->json($comment, 201);
}
```

Лістинг В.5 – Клас події (файл Events/NewCommentCreated.php)

```
class NewCommentCreated implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $comment;

    public function __construct(Comment $comment)
    {
        $this->comment = $comment;
    }

    // Визначення приватного каналу трансляції 'posts.{id}'
    public function broadcastOn(): array
    {
        return [
            new PrivateChannel('posts.' . $this->comment->post_id),
        ];
    }
}
```

Лістинг В.6 – Асинхронний слухач (файл Listeners/NotifyPostAuthor.php)

```
// Імплементація ShouldQueue переводить виконання у фоновий режим
class NotifyPostAuthor implements ShouldQueue
{
    use InteractsWithQueue;

    public function handle(NewCommentCreated $event): void
    {
```

```
        // Отримання автора через Lazy Loading
        $author = $event->comment->post->user;

        // Асинхронна відправка E-mail
        Mail::to($author)->send(new CommentReceived($event->comment));
    }
}
```

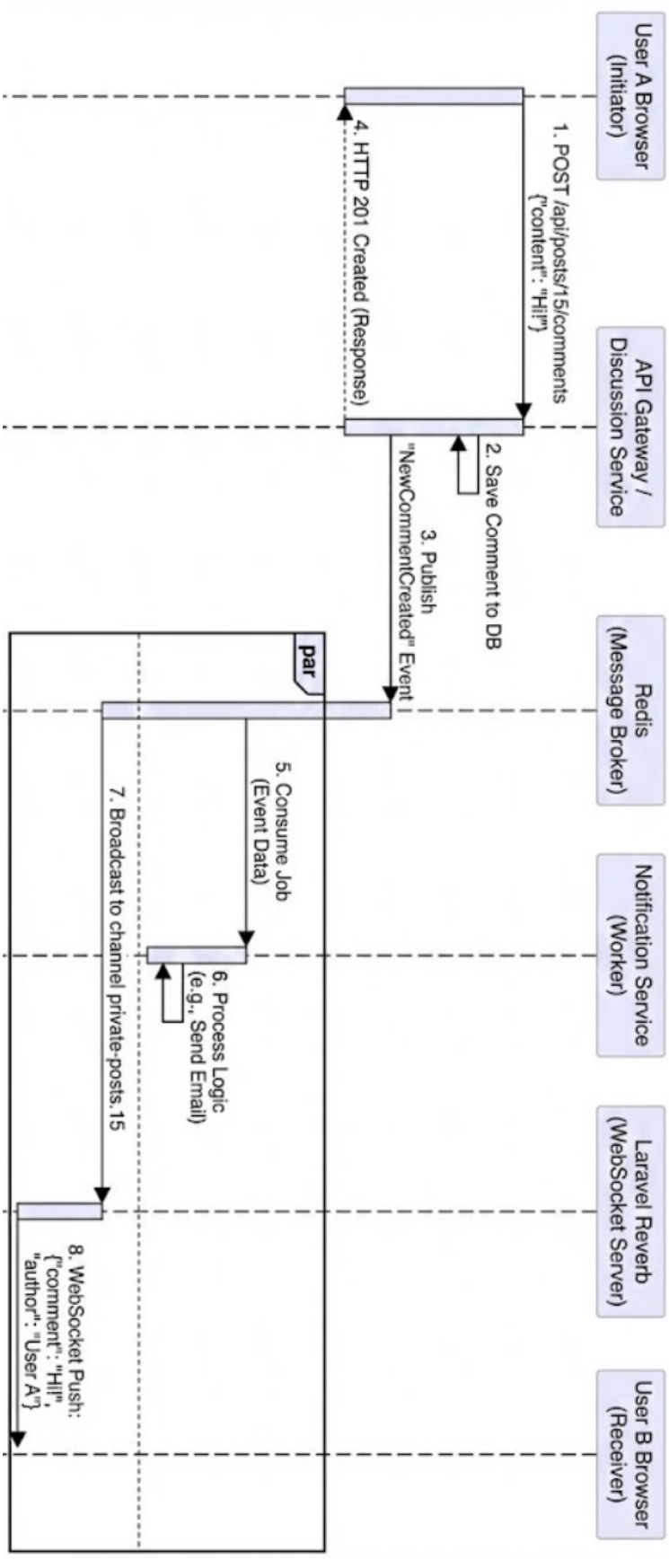
Лістинг В.7 – Підписка на події (JavaScript / Client-side)

```
// Підписка на приватний канал конкретного посту
window.Echo.private(`posts.${postId}`)
    .listen('NewCommentCreated', (e) => {
        console.log('Real-time update received:', e.comment);

        // Динамічне оновлення інтерфейсу
        appendCommentToFeed(e.comment);
    });
```

Додаток Д.1

Діаграма послідовності обробки події коментування



Додаток Д.2

Схема автоматизованого конвеєра розгортання