

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Інформаційних управляючих систем
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)
Дослідження методів удосконалення процесів розробки та
управління ІТ-проектами для створення музичних композицій
(тема)

Виконав:

здобувач 2 року навчання,
групи УПГІТМ-24-1

Антон РОМАНЮК
(власне ім'я, прізвище)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Управління проектами в
галузі інформаційних технологій

Керівник доц. каф. ІУС Віталій БРУСЕНЦЕВ
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ІУС


(підпис)

Костянтин ПЕТРОВ
(власне ім'я, прізвище)

2026_ р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
Кафедра Інформаційних управляючих систем
Рівень вищої освіти другий (магістерський)
Спеціальність 122 Комп'ютерні науки
(код і повна назва)
Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)
Освітня програма Управління проектами в галузі інформаційних технологій
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри 
(підпис)

« 06 » квітня 20 26 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Романюку Антону Сергійовичу
(прізвище, ім'я, по батькові)

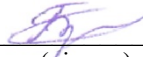
1. Тема роботи «Дослідження методів удосконалення процесів розробки та управління ІТ-проектами для створення музичних композицій»
затверджена наказом університету від 23 березня 2026 р. № 231Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 19 травня 2026 р.
3. Вихідні дані до роботи Об'єкт дослідження – процеси розробки та управління ІТ-проектами у сфері створення музичного контенту. Предмет дослідження – методи, моделі та інструменти автоматизації життєвого циклу розробки програмних систем. Предмет розробки – компоненти архітектури та інфраструктура автоматизованого розгортання інформаційної системи для створення музичних композицій. Мета кваліфікаційної роботи – підвищення ефективності процесів розробки та управління ІТ-проекту шляхом удосконалення процесів інтеграції, тестування та доставки програмного забезпечення. Технічне забезпечення: IBM-сумісний персональний комп'ютер.
4. Перелік питань, що потрібно опрацювати в роботі Вступ. Аналіз предметної області. Визначення проблеми управління ІТ-проектами у сфері створення музичних композицій. Дослідження існуючих рішень. Аналіз процесів розробки програмних систем. Постановка задачі. Обґрунтування вибору підходів до організації процесів розробки та управління ІТ-проектами. Опис архітектури розробленої системи. Організація контролю версій та управління потоками розробки. Контейнеризація середовища розробки та розгортання. Реалізація автоматизованих сценаріїв інтеграції та доставки. Встановлення засобів моніторингу та супроводу системи. Практичний аналіз організації процесів розробки та інтеграції змін. Оцінка впливу впроваджених методів на стабільність і надійність системи. Висновки. Перелік джерел посилання.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Визначення проблеми управління ІТ-проектами	06.04.2026 - 07.04.2026	Виконано
2	Дослідження існуючих рішень	07.04.2026 - 08.04.2026	Виконано
3	Аналіз процесів розробки систем	08.04.2026 - 09.04.2026	Виконано
4	Розробка плану покращення процесів	09.04.2026 - 11.04.2026	Виконано
5	Визначення структури процесів розробки	11.04.2026 - 13.04.2026	Виконано
6	Організація контролю версій	13.04.2026 - 15.04.2026	Виконано
7	Організація процесів автоматизації	15.04.2026 - 17.04.2026	Виконано
8	Розробка та тестування програмної частини	17.04.2026 - 01.05.2026	Виконано
9	Аналіз результатів	01.05.2026 - 05.05.2026	Виконано
10	Оформлення пояснювальної записки	05.05.2026 - 15.05.2026	Виконано
11	Підготовка презентаційного матеріалу та перевірка	15.05.2026 - 19.05.2026	Виконано
12	Захист кваліфікаційної роботи	21.05.2026	Виконано

Дата видачі завдання 06 квітня 2026 р.

Здобувач 
(підпис)

Керівник роботи 
(підпис) доц. каф. ІУС Віталій БРУСЕНЦЕВ
(посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка до магістерської кваліфікаційної роботи: 95 сторінок, 7 рисунків, 3 таблиці, 1 додаток, 30 джерел інформації.

АВТОМАТИЗАЦІЯ ПРОЦЕСІВ, МУЗИКАЛЬНІ КОМПОЗИЦІЇ, ОРГАНІЗАЦІЯ РОЗРОБКИ, ПРОЦЕСИ РОЗРОБКИ, УПРАВЛІННЯ ІТ-ПРОЄКТАМИ.

Об'єкт дослідження – процеси розробки програмного забезпечення у сфері створення музичних композицій.

Предмет дослідження – методи та підходи до організації та управління процесами розробки ІТ-проєктів.

Мета кваліфікаційної роботи – підвищення ефективності процесів розробки та управління ІТ-проєктами створення музичних композицій за рахунок удосконалення організації робочих процесів, взаємодії команди та автоматизації основних етапів розробки.

Методи дослідження – методи організації процесів розробки програмного забезпечення, підходи до управління ІТ-проєктами, а також сучасні інформаційні технології та практики автоматизації процесів розробки.

Виконано аналіз предметної галузі розробки програмних систем для створення музичних композицій, визначено основні підходи до організації процесів розробки та управління проєктами, встановлено ключові етапи життєвого циклу програмного забезпечення. На цій основі досліджено особливості організації роботи команди розробників, управління змінами у програмному коді та автоматизації процесів тестування і розгортання.

Галузь застосування розробки – організація процесів розробки програмного забезпечення у сфері створення музичних композицій.

ABSTRACT

Master's thesis: 95 pages, 7 figures, 3 tables, 1 appendices, 30 references.

DEVELOPMENT ORGANIZATION, IT PROJECT MANAGEMENT, MUSICAL COMPOSITIONS, PROCESS AUTOMATION, SOFTWARE DEVELOPMENT PROCESSES.

The object of research is the software development processes in the field of musical composition systems.

The subject of the research is methods and approaches to organizing and managing IT project development processes.

The purpose of the thesis is to improve the efficiency of development and management processes in IT projects related to musical compositions by optimizing workflow organization, enhancing team collaboration, and implementing automation practices.

Research methods include approaches to software development process organization, IT project management methodologies, and modern information technologies used to support development workflows.

An analysis of the subject area of software systems for musical composition has been conducted, key approaches to development process organization and project management have been identified, and the main stages of the software lifecycle have been defined. Based on this, particular attention has been given to team coordination, code management practices, and the automation of testing and deployment processes.

The field of application of the research is the organization and improvement of software development processes in systems for creating musical compositions.

ЗМІСТ

	С.
Скорочення та умовні позначки.....	8
Вступ.....	9
1 Аналіз предметної області.....	11
1.1 Визначення проблеми управління ІТ-проєктами у сфері створення музичних композицій.....	11
1.2 Дослідження існуючих рішень.....	12
1.3 Аналіз процесів розробки програмних систем у сфері створення музичних композицій.....	15
1.4 Постановка задачі.....	18
2 Методи та підходи до організації процесів розробки та управління іт-проєктами.....	21
2.1 Підходи до організації процесів розробки програмного забезпечення.....	21
2.2 Методи управління командною роботою в ІТ-проєктах.....	24
2.3 Методи управління змінами та контролю версій програмного коду....	27
2.4 Методи автоматизації процесів тестування та розгортання.....	28
2.5 Обґрунтування вибору підходів до організації процесів розробки.....	32
3 Практичне дослідження та реалізація методів управління при розробці музичної платформи.....	35
3.1 Проєктування функціональної структури та архітектури системи.....	35
3.2 Організація контролю версій та управління потоками розробки.....	39
3.3 Контейнеризація середовища розробки та розгортання з використанням Docker.....	44
3.4 Автоматизація процесів безперервної інтеграції та доставки програмного забезпечення.....	49
3.4.1 Обґрунтування архітектури та інструментарію автоматизованих процесів CI/CD.....	49
3.4.2 Автоматизація безперервної інтеграції (CI).....	51
3.4.3 Конвеєр безперервної доставки та розгортання (CD).....	55
3.5 Засоби моніторингу та супроводу програмної системи в процесі експлуатації.....	57

4 Аналіз результатів дослідження та оцінка ефективності впроваджених методів.....	59
4.1 Критерії оцінки ефективності процесів розробки та управління версіями.....	59
4.2 Практичний аналіз організації процесів розробки та інтеграції змін.....	62
4.3 Оцінка впливу GitFlow, Docker та CI/CD на стабільність і надійність системи.....	68
4.4 Рекомендації щодо застосування запропонованого підходу та можливості подальшого розвитку системи.....	73
Висновки.....	77
Перелік джерел посилання.....	79
Додаток А Графічний матеріал кваліфікаційної роботи.....	82

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ПЗ – програмний застосунок

API – Application Programming Interface – інтерфейс прикладного програмування

CDN – Content Delivery Network – географічно розподілена мережа серверів

CI/CD – Continuous Integration/Continuous Deployment – безперервна інтеграція та безперервне розгортання

DevOps – Development and Operations – розробка та експлуатація

DSP – Digital Signal Processing – цифрова обробка сигналів

ELK Stack – набір open-source інструментів Elasticsearch, LogStash, Kibana

GIT – Global Information Tracker – глобальний трекер інформації

HTTP – Hypertext Transfer Protocol – протокол передачі гіпертексту

IaC – Infrastructure as Code – інфраструктура як код

IT – Information Technology – інформаційні технології

JSON – JavaScript Object Notation – нотація об'єктів JavaScript

JWT – JSON Web Tokens – веб-токени JSON

MIDI – Musical Instrument Digital Interface – цифровий інтерфейс музичних інструментів

MTTR – Mean Time To Recovery – середній час відновлення

OAuth – Open Authorization – відкритий протокол авторизації

ORM – Object-Relational Mapping – об'єктно-реляційне відображення

REST – Representational State Transfer – передача репрезентативного стану

UI – User Interface – інтерфейс користувача

URL – Uniform Resource Locator – уніфікований локатор ресурсу

UX – User Experience – користувацький досвід

XML – eXtensible Markup Language – розширювана мова розмітки

ВСТУП

У сучасних умовах розвитку цифрових технологій програмне забезпечення відіграє важливу роль у процесах створення, обробки та поширення музичних композицій. Розвиток веб-технологій, хмарних сервісів і засобів автоматизації сприяв появі великої кількості програмних систем для створення музичного контенту, які потребують ефективної організації процесів розробки та управління.

Сучасні IT-проекти у сфері створення музичних композицій характеризуються високою складністю, необхідністю командної взаємодії, швидким внесенням змін до програмного коду та потребою у підтримці стабільності ПЗ. Неефективна організація, недостатня автоматизація та слабка координація роботи команди можуть призводити до збільшення часу розробки, виникнення помилок і ускладнення підтримки програмних систем.

Особливо актуальною є проблема організації спільної роботи над програмним кодом, автоматизації тестування та розгортання застосунків. Багато команд використовують системи Git, засоби контейнеризації та інструменти безперервної інтеграції й доставки ПЗ як окремі елементи процесу розробки, що не завжди забезпечує достатній рівень узгодженості та ефективності роботи.

Використання сучасних підходів до управління IT-проектами, зокрема Agile, Scrum і Development and Operations (DevOps), а також застосування Git, Docker і засобів Continuous Integration/Continuous Deployment (CI/CD) дозволяє підвищити ефективність процесів розробки, зменшити кількість помилок і покращити взаємодію між учасниками команди.

Дослідження спрямоване на аналіз та удосконалення процесів розробки і управління IT-проектами у сфері створення музичних композицій. Для досягнення поставленої мети необхідно:

- проаналізувати сучасні підходи до організації процесів розробки

програмного забезпечення;

- дослідити методи управління ІТ-проєктами та особливості їх застосування у сфері створення музичних композицій;

- визначити особливості використання систем контролю версій у командній розробці;

- дослідити можливості контейнеризації та автоматизації процесів розробки;

- проаналізувати підходи до автоматизації тестування та розгортання програмного забезпечення;

- сформулювати рекомендації щодо підвищення ефективності процесів розробки та управління ІТ-проєктами.

Об'єктом дослідження є процеси розробки програмного забезпечення у сфері створення музичних композицій, а предметом дослідження – методи та підходи до організації, автоматизації та управління процесами розробки ІТ-проєктів.

Наукова новизна роботи полягає у дослідженні комплексного підходу до організації процесів розробки програмного забезпечення для створення музичних композицій із використанням сучасних засобів автоматизації, контейнеризації та управління командною розробкою. Запропонований підхід дозволяє підвищити ефективність командної взаємодії, спростити підтримку програмних систем та оптимізувати процеси тестування і розгортання програмного забезпечення.

Практична значущість роботи полягає у можливості використання запропонованих рекомендацій і підходів для організації ефективних процесів розробки сучасних програмних систем у музичній сфері. Результати дослідження можуть бути використані при створенні веб-застосунків та інформаційних систем для роботи з музичними композиціями.

За матеріалами кваліфікаційної роботи було опубліковано тези у збірнику праць 30-го Міжнародного молодіжного форуму «Радіоелектроніка та молодь у XXI столітті» [1].

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Визначення проблеми управління ІТ-проектами у сфері створення музичних композицій

Сучасна музична індустрія тісно пов'язана з використанням інформаційних технологій, які забезпечують процеси створення, обробки, зберігання та поширення музичних композицій [2]. Програмні системи, що використовуються у цій сфері, охоплюють широкий спектр рішень: від онлайн-платформ з музичними бібліотеками до спеціалізованих програмних середовищ для створення музики, цифрових аудіостанцій, а також інструментів аналізу та обробки звуку.

До таких систем належать як веб-додатки для роботи з музичним контентом: семпли, файли Musical Instrument Digital Interface (MIDI-файли), лупи; так і складні програмні продукти, що дозволяють створювати повноцінні музичні композиції, виконувати мікшування, обробку сигналів та інтеграцію з іншими цифровими сервісами. Крім того, сучасні рішення часто включають інструменти автоматизованого аналізу звуку, рекомендаційні системи, а також компоненти, пов'язані з використанням технологій штучного інтелекту.

З огляду на таке різноманіття функціональних можливостей, розробка подібних програмних систем є складним процесом, що вимагає ефективної організації роботи команди, чіткої координації дій учасників проєкту та використання сучасних підходів до управління розробкою. Основна складність полягає не лише у технічній реалізації функціоналу, але й у забезпеченні узгодженості всіх етапів розробки програмного продукту.

Однією з ключових проблем є організація спільної роботи розробників. У процесі створення програмного забезпечення різні учасники команди працюють над окремими компонентами системи, що потребує постійної синхронізації змін. За відсутності належних підходів до управління процесами розробки можуть виникати

конфлікти у програмному коді, дублювання функціоналу та зниження якості кінцевого продукту.

Важливим аспектом також є управління змінами у програмному забезпеченні. У сфері створення музичних композицій вимоги до програмних продуктів є динамічними та можуть змінюватися залежно від нових технологічних можливостей, тенденцій у музичній індустрії та потреб користувачів. Особливо це актуально в умовах стрімкого розвитку інструментів, пов'язаних із застосуванням штучного інтелекту, що формують нові підходи до створення музики та впливають на очікування користувачів від програмних систем.

Часті зміни вимог ускладнюють процес розробки, оскільки потребують гнучкого реагування з боку команди та ефективного контролю впливу цих змін на вже реалізований функціонал. Відсутність належної організації цього процесу може призводити до затримок у розробці, появи помилок та зниження стабільності системи.

Окрему роль у вирішенні зазначених проблем відіграє автоматизація процесів розробки. Використання підходів до автоматизації тестування, інтеграції та розгортання програмного забезпечення дозволяє зменшити кількість помилок, прискорити процес розробки та підвищити загальну ефективність управління ІТ-проєктами. Водночас відсутність таких підходів значно ускладнює підтримку та розвиток програмних систем.

Таким чином, проблема управління ІТ-проєктами у сфері створення музичних композицій полягає у необхідності забезпечення ефективної організації процесів розробки, координації роботи команди, управління змінами та впровадження сучасних підходів до автоматизації. Врахування специфіки музичної індустрії, зокрема динамічності вимог та впливу нових технологій, є важливою умовою створення якісних і конкурентоспроможних програмних продуктів.

1.2 Дослідження існуючих рішень

У сучасній музичній індустрії існує велика кількість програмних систем, які забезпечують створення, обробку, поширення та аналіз музичних композицій. Дані системи постійно розвиваються, вдосконалюються та адаптуються до нових вимог користувачів, що робить їх важливим об'єктом дослідження з точки зору організації процесів розробки та управління ІТ-проектами.

Серед існуючих рішень можна виділити декілька основних категорій, кожна з яких має свої особливості як з точки зору функціональності, так і з точки зору підходів до їх розробки. До таких категорій належать цифрові аудіостанції, онлайн-платформи для розповсюдження музики, а також сучасні системи, що використовують технології штучного інтелекту для генерації музичного контенту.

Одним із найбільш відомих представників програмного забезпечення для створення музики є FL Studio, яка відноситься до класу цифрових аудіостанцій. Дана система надає користувачам широкий набір інструментів для створення, редагування та обробки аудіо, включаючи роботу з MIDI, віртуальними інструментами та ефектами [3]. З точки зору розробки, подібні системи характеризуються високим рівнем складності, оскільки потребують підтримки великої кількості функцій, сумісності з різними форматами та інтеграції з зовнішніми плагінами.

Розробка таких програмних продуктів потребує чіткої організації процесів, оскільки велика кількість компонентів системи створює ризик виникнення помилок, конфліктів у коді та труднощів у тестуванні. Крім того, необхідність постійного оновлення функціоналу відповідно до нових вимог користувачів ускладнює процес управління проектом та вимагає використання гнучких підходів до розробки.

Іншою важливою категорією є онлайн-платформи для розповсюдження та

обміну музичним контентом, прикладом яких є SoundCloud. Подібні системи орієнтовані на взаємодію великої кількості користувачів, забезпечення зберігання аудіофайлів, стрімінг та соціальні функції, такі як коментарі, підписки та рекомендації [4].

З точки зору процесів розробки, такі системи мають свої специфічні особливості. Вони повинні забезпечувати високу масштабованість, стабільність роботи при великому навантаженні, а також безперервне оновлення функціоналу. Це вимагає ефективної організації процесів тестування, розгортання та моніторингу системи. Крім того, велика кількість користувачів створює необхідність швидкого реагування на зміни у вимогах та поведінці аудиторії.

Окрему увагу слід приділити сучасним системам, що використовують технології штучного інтелекту, таким як Suno AI [5]. Подібні рішення дозволяють автоматизувати процес створення музичних композицій, генеруючи мелодії, тексти або повноцінні треки на основі заданих параметрів.

З розвитком таких технологій суттєво змінюються підходи до розробки програмного забезпечення у музичній сфері. Інтеграція компонентів штучного інтелекту потребує використання нових методів тестування, обробки даних та оптимізації продуктивності. Крім того, такі системи потребують постійного вдосконалення моделей, що створює додаткові виклики для управління життєвим циклом програмного продукту. В таблиці 1.1 наведене порівняння описаних музичних платформ.

Аналіз зазначених систем дозволяє зробити висновок, що незалежно від їх типу, всі вони мають спільні риси з точки зору процесів розробки. По-перше, це висока складність архітектури, яка вимагає чіткої організації роботи команди розробників. По-друге, це необхідність постійного оновлення функціоналу, що пов'язано з динамічністю музичної індустрії. Тренди у музиці, інструментах та підходах до створення контенту швидко змінюються, що безпосередньо впливає на вимоги до програмних систем.

Таблиця 1.1 – Порівняння сучасних музичних платформ

Система	Тип	Основні функції	Особливості розробки
FL Studio	DAW	Створення та обробка музики	Висока складність архітектури
SoundCloud	Вебплатформа	Стрімінг та взаємодія користувачів	Масштабованість та CI/CD
Suno AI	AI-система	Генерація музики	Інтеграція AI та обробка даних

Особливої актуальності набуває цей фактор у контексті стрімкого розвитку технологій штучного інтелекту, який формує так звану “AI-хвилю” у музичній індустрії. Це призводить до появи нових типів функціоналу, які необхідно оперативно інтегрувати у вже існуючі системи. У таких умовах традиційні підходи до розробки часто виявляються недостатньо ефективними, що обумовлює необхідність їх удосконалення.

Також важливо відзначити, що більшість сучасних систем розробляється великими командами, які працюють над різними частинами продукту. Це створює додаткові труднощі у координації роботи, управлінні змінами у програмному коді та забезпеченні узгодженості між компонентами системи. Відсутність ефективних процесів управління може призводити до затримок, зниження якості продукту та ускладнення його подальшого розвитку.

Таким чином, проведений аналіз існуючих рішень показує, що сучасні системи для створення музичних композицій мають високий рівень складності та динамічності, що створює значні виклики для процесів їх розробки. Це підтверджує актуальність дослідження методів удосконалення процесів розробки та управління ІТ-проєктами, що дозволить підвищити ефективність створення та підтримки програмних систем у музичній сфері.

1.3 Аналіз процесів розробки програмних систем у сфері створення музичних композицій

Процеси розробки програмних систем у сфері створення музичних композицій мають ряд специфічних особливостей, що відрізняють їх від розробки інших типів програмного забезпечення. Це пов'язано як зі складністю функціоналу таких систем, так і з високими вимогами до їх гнучкості, продуктивності та зручності використання [6]. Сучасні програмні рішення у музичній сфері, незалежно від їх типу, проходять класичні етапи життєвого циклу програмного забезпечення, які включають аналіз вимог, проєктування, реалізацію, тестування, розгортання та підтримку. Однак кожен із цих етапів має свої особливості, обумовлені специфікою предметної області.

Етап аналізу вимог у даному випадку є особливо складним, оскільки вимоги до систем часто є нечіткими або змінюються в процесі розробки. Це пов'язано з тим, що музична індустрія характеризується високою динамічністю, а користувачі очікують постійного оновлення функціоналу та впровадження нових можливостей. Зокрема, поява нових форматів контенту, платформ дистрибуції та інструментів на базі штучного інтелекту суттєво впливає на формування вимог до програмних систем. Наприклад, сучасні музичні сервіси дедалі частіше інтегрують AI-функції генерації композицій, автоматичного мастерингу або рекомендаційних алгоритмів, що створює необхідність постійного оновлення архітектури програмного забезпечення.

На етапі проєктування важливу роль відіграє вибір архітектурних рішень, які повинні забезпечити масштабованість, гнучкість та можливість подальшого розширення системи. Це особливо актуально для програмних продуктів, що інтегрують велику кількість функціональних модулів, таких як робота з аудіофайлами, обробка медіаконтенту, система взаємодії користувачів та інтеграція

із зовнішніми сервісами. Для таких систем часто використовуються модульна або мікросервісна архітектура, що дозволяє ізолювати окремі компоненти та спрощує їх подальший розвиток.

Етап реалізації пов'язаний із необхідністю координації роботи команди розробників, які можуть працювати над різними частинами системи одночасно. Наприклад, окремі учасники команди можуть відповідати за серверну логіку, інтерфейс користувача, систему зберігання медіафайлів або інтеграцію сервісів обробки аудіо. У таких умовах виникає потреба в ефективному управлінні змінами у програмному коді, що дозволяє уникнути конфліктів та забезпечити стабільність роботи системи. Для цього використовуються системи контролю версій Git, стратегії розгалуження Gitflow, а також механізми code review та pull request.

Особливістю музичних платформ є також необхідність роботи з великими мультимедійними файлами. Аудіофайли займають значний обсяг пам'яті та створюють додаткове навантаження на серверну інфраструктуру. Це вимагає використання оптимізованих механізмів зберігання контенту, кешування та асинхронної обробки даних. У багатьох сучасних системах для цього використовуються хмарні сховища, мережі Content Delivery Network (CDN-мережі) та системи кешування, наприклад Redis [7].

Тестування програмних систем у музичній сфері також має свої особливості. Окрім перевірки функціональності, необхідно враховувати продуктивність системи, швидкість обробки медіаданих та стабільність роботи при високому навантаженні. Наприклад, платформи для роботи з музичним контентом повинні забезпечувати стабільне завантаження аудіофайлів, коректну обробку метаданих та безперебійну взаємодію між користувачами. Це ускладнює процес тестування та вимагає використання автоматизованих підходів, зокрема CI/CD-пайплайнів та автоматичного запуску тестів після внесення змін у код [8].

Етап розгортання та підтримки системи передбачає забезпечення її стабільної роботи у реальних умовах використання. Для сучасних систем характерним є

використання контейнеризації та хмарних технологій, що дозволяє стандартизувати середовище виконання застосунку та спростити процес оновлення програмного забезпечення. Використання Docker та Docker Compose дозволяє забезпечити однаковість середовищ розробки, тестування та production, а також спрощує інтеграцію із процесами безперервної інтеграції та доставки [9-10].

Важливо також зазначити, що всі етапи розробки взаємопов'язані між собою, і неефективна організація одного з них може негативно вплинути на весь процес створення програмного продукту. Це особливо актуально для складних систем, які розробляються у динамічному середовищі та потребують постійного вдосконалення функціональності відповідно до нових вимог музичної індустрії.

Таким чином, аналіз процесів розробки програмних систем у сфері створення музичних композицій показує, що їх ефективність значною мірою залежить від правильного вибору підходів до організації роботи, управління змінами, автоматизації тестування та використання сучасних DevOps-практик. Це підтверджує необхідність дослідження методів удосконалення процесів розробки та управління ІТ-проєктами у даній сфері.

1.4 Постановка задачі

Метою даної роботи є підвищення ефективності процесів розробки та управління ІТ-проєктами у сфері створення музичних композицій шляхом дослідження та удосконалення підходів до організації процесів розробки, координації роботи команди та автоматизації основних етапів життєвого циклу програмного забезпечення.

Для досягнення поставленої мети необхідно виконати комплекс взаємопов'язаних задач, що охоплюють як теоретичні аспекти дослідження, так і їх

практичне застосування.

Перш за все, необхідно провести аналіз предметної області, зокрема дослідити існуючі програмні системи для створення музичних композицій та визначити особливості їх розробки. Це дозволить виявити основні проблеми, пов'язані з організацією процесів розробки та управління ІТ-проектами у даній сфері.

Наступним етапом є дослідження існуючих підходів до організації процесів розробки програмного забезпечення, включаючи управління командною роботою, контроль змін у програмному коді, а також автоматизацію процесів тестування та розгортання систем. На основі цього аналізу необхідно визначити найбільш ефективні практики, які можуть бути застосовані для вдосконалення процесів розробки.

Важливим завданням є формування узагальненого підходу до організації процесів розробки та управління ІТ-проектами, який враховує специфіку створення програмних систем у музичній сфері, зокрема динамічність вимог, необхідність інтеграції нових технологій та роботу з мультимедійним контентом.

Крім теоретичного дослідження, у роботі передбачається практична перевірка запропонованих підходів. Для цього планується реалізація програмної системи, яка буде використовуватися як приклад застосування досліджених методів. Дана система матиме функціональні можливості, характерні для сучасних музичних платформ, зокрема роботу з користувачами, управління контентом та забезпечення взаємодії між учасниками.

У межах реалізації програмної системи необхідно забезпечити виконання базових функціональних можливостей, серед яких:

- організація роботи користувачів системи, включаючи механізми реєстрації та авторизації;
- забезпечення можливості взаємодії з музичним контентом, зокрема його завантаження, перегляду та збереження;

- реалізація механізмів взаємодії між користувачами, таких як коментарі або інші форми зворотного зв'язку;

- управління доступом до функціоналу системи відповідно до ролей користувачів.

При цьому основний акцент у роботі робиться не на розробці конкретного програмного продукту, а на дослідженні та демонстрації ефективності запропонованих методів організації процесів розробки та управління ІТ-проєктами. Реалізована система виступає як інструмент для практичної апробації цих методів.

Таким чином, у результаті виконання роботи передбачається отримання узагальнених підходів до вдосконалення процесів розробки програмного забезпечення у сфері створення музичних композицій, що дозволить підвищити ефективність управління ІТ-проєктами та якість кінцевого програмного продукту.

2 МЕТОДИ ТА ПІДХОДИ ДО ОРГАНІЗАЦІЇ ПРОЦЕСІВ РОЗРОБКИ ТА УПРАВЛІННЯ ІТ-ПРОЄКТАМИ

2.1 Підходи до організації процесів розробки програмного забезпечення

Сучасна розробка програмного забезпечення базується на використанні різноманітних підходів до організації процесів, які визначають послідовність виконання робіт, способи взаємодії між учасниками проєкту та методи досягнення поставлених цілей. Вибір відповідного підходу до організації розробки безпосередньо впливає на швидкість створення програмного продукту, його якість, стабільність та можливість адаптації до змін вимог у процесі реалізації проєкту.

У загальному випадку процес розробки програмного забезпечення включає етапи аналізу вимог, проєктування системи, реалізації функціональності, тестування, розгортання та подальшого супроводу програмного продукту. Залежно від специфіки проєкту та обраної методології ці етапи можуть виконуватись послідовно або ітеративно.

Одним із традиційних підходів до організації процесів розробки є каскадна модель (Waterfall), яка передбачає послідовне виконання етапів життєвого циклу програмного забезпечення [11]. У межах такого підходу кожен наступний етап розпочинається лише після завершення попереднього. Перевагами даної моделі є чітка структура процесу, прогнозованість результатів та високий рівень документації. Водночас каскадна модель характеризується низькою гнучкістю щодо змін вимог, що ускладнює її використання у сучасних ІТ-проєктах, особливо у сфері цифрових музичних платформ та вебсервісів, де функціональні вимоги можуть змінюватися протягом усього процесу розробки.

У сучасних умовах більшого поширення набувають гнучкі підходи до організації розробки програмного забезпечення, об'єднані концепцією Agile. Agile-методології орієнтовані на ітеративну розробку, швидке впровадження

функціональності та постійну адаптацію до змін вимог. Такий підхід є особливо актуальним для проєктів у сфері музичних вебплатформ та цифрових сервісів, оскільки дозволяє оперативно реагувати на зміну потреб користувачів, розвиток технологій та появу нових вимог до функціональності системи.

Одними з найбільш поширених Agile-підходів є Scrum та Kanban. Методологія Scrum передбачає організацію процесу розробки у вигляді коротких ітерацій – спринтів, у межах яких команда реалізує визначений набір функціональних можливостей системи. Кожен спринт завершується отриманням окремого інкременту програмного продукту, що дозволяє регулярно оцінювати результати розробки та вносити необхідні зміни до подальшого планування.

Підхід Kanban орієнтований на безперервне управління потоком задач та візуалізацію процесу розробки. Основною особливістю Kanban є контроль поточного навантаження команди та забезпечення безперервного виконання задач без жорсткого поділу на ітерації. Використання Kanban дозволяє підвищити прозорість процесів розробки та ефективніше контролювати виконання задач у межах проєкту [12].

Важливою особливістю сучасних підходів до організації розробки є активне використання практик командної взаємодії, регулярного зворотного зв'язку та безперервного вдосконалення процесів. Це дозволяє своєчасно виявляти проблеми, покращувати архітектуру програмного забезпечення та підтримувати стабільність програмного продукту на всіх етапах життєвого циклу.

Для візуалізації циклічності та взаємозв'язку між окремими етапами менеджменту проєкту, на рисунку 2.1 наведено загальну схему ітеративного процесу розробки, що демонструє шлях від формування беклогу до випуску готового релізу та отримання зворотного зв'язку

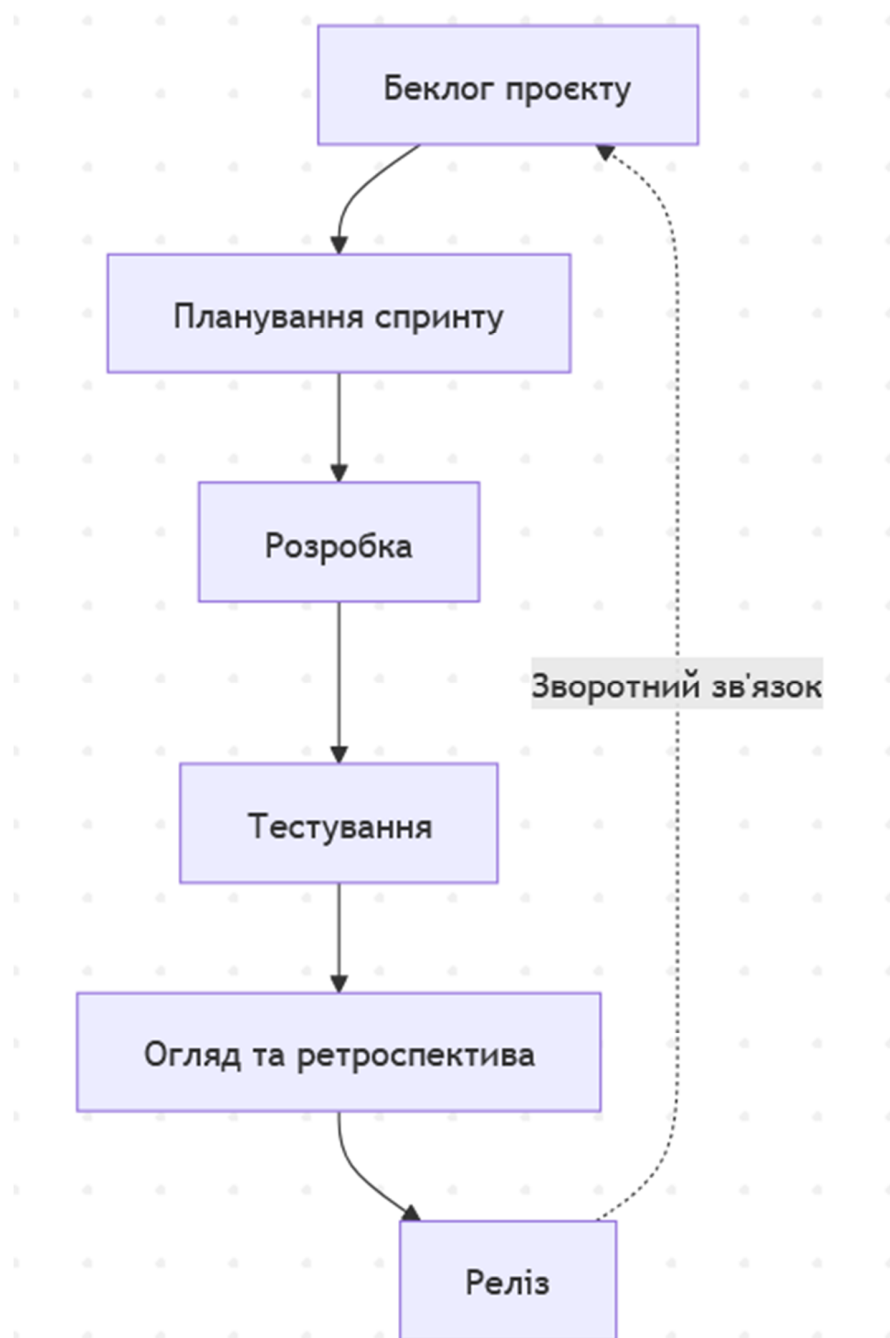


Рисунок 2.1 – Схема ітеративного процесу розробки за Agile-підходом

Окрім Agile-методологій, важливу роль у сучасній організації процесів розробки відіграє підхід DevOps, який поєднує процеси розробки програмного забезпечення, тестування, адміністрування інфраструктури та розгортання системи

в єдину автоматизовану модель життєвого циклу програмного продукту. Основною метою DevOps є підвищення швидкості та стабільності доставки нових версій програмного забезпечення шляхом автоматизації рутинних процесів та забезпечення тісної взаємодії між усіма учасниками процесу розробки.

Підхід DevOps передбачає використання практик CI/CD, автоматизованого тестування, контейнеризації середовища виконання та централізованого управління конфігураціями системи. Використання даних підходів дозволяє мінімізувати кількість помилок під час інтеграції змін, забезпечити стабільність середовища виконання застосунку та скоротити час між реалізацією функціональності та її появою у production-середовищі.

Особливого значення DevOps-підхід набуває у веборієнтованих програмних системах та цифрових платформах для роботи з музичним контентом, де важливими факторами є стабільність роботи сервісу, швидкість оновлення функціональності та безперервність доступу користувачів до системи.

Таким чином, сучасні підходи до організації процесів розробки програмного забезпечення орієнтовані на забезпечення гнучкості, автоматизації та безперервного вдосконалення програмного продукту. Поєднання Agile-методологій та DevOps-практик дозволяє ефективно організувати процес створення програмних систем, забезпечити стабільність процесів розробки та підвищити якість кінцевого програмного продукту.

2.2 Методи управління командною роботою в ІТ-проєктах

Ефективне управління командною роботою є одним із ключових факторів успішної реалізації сучасних ІТ-проєктів. У процесі розробки програмного забезпечення для роботи з музичним контентом та вебплатформами необхідно

забезпечити координацію дій між учасниками команди, контроль виконання задач, підтримку стабільності процесу розробки та своєчасне впровадження нової функціональності.

У сучасних ІТ-проєктах командна взаємодія базується на принципах гнучкого управління, що передбачають постійний обмін інформацією між учасниками розробки, швидке реагування на зміни вимог та безперервне вдосконалення процесів створення програмного забезпечення. Особливого значення такі підходи набувають у проєктах, пов'язаних із вебплатформами для роботи з музичним контентом, де вимоги до функціональності можуть змінюватися протягом усього життєвого циклу системи.

До складу команди розробки сучасної програмної системи можуть входити backend- та frontend-розробники, DevOps-інженери, тестувальники, адміністратори інфраструктури та інші спеціалісти, які спільно беруть участь у створенні та супроводі програмного продукту. Ефективна взаємодія між учасниками команди дозволяє забезпечити узгодженість архітектурних рішень, стабільність програмного забезпечення та контроль якості реалізації функціональності системи.

Одним із важливих елементів управління командною роботою є планування процесу розробки. У межах Agile-підходів планування здійснюється шляхом формування переліку задач (backlog), визначення пріоритетності функціональності та розподілу задач між учасниками команди. Такий підхід дозволяє ефективно організувати процес розробки та забезпечити контроль за виконанням робіт на кожному етапі реалізації проєкту.

Для організації командної взаємодії широко використовуються спеціалізовані системи управління задачами, зокрема Jira, Trello або GitHub Projects. Дані інструменти дозволяють централізовано управляти задачами, відстежувати стан їх виконання, контролювати строки реалізації функціональності та координувати взаємодію між учасниками проєкту. Використання подібних систем підвищує прозорість процесів розробки та спрощує управління змінами у програмному

продукті. На рисунку 2.2 наведено приклад дошки спринта з завданнями в Jira.

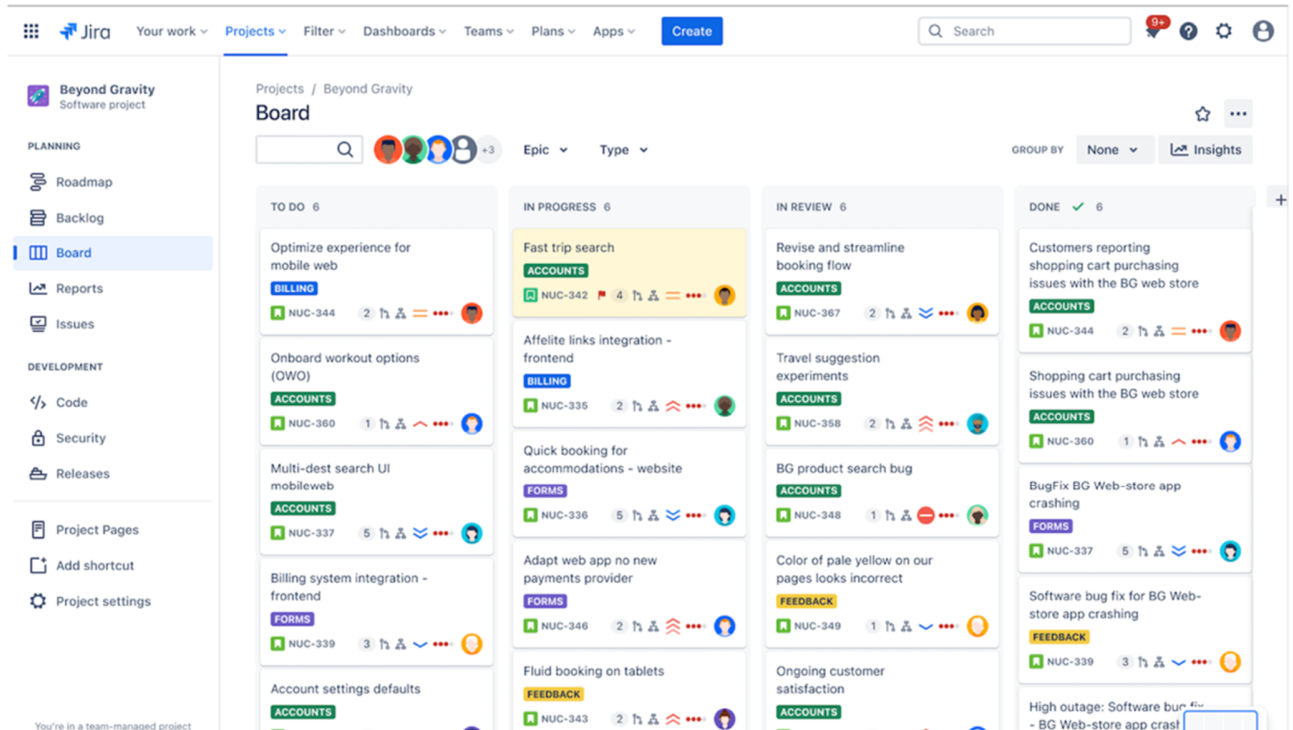


Рисунок 2.2 – Дошка завдань спринту у Jira

Важливою складовою сучасної командної роботи є використання механізмів Code Review та Pull Request. Перед інтеграцією змін до основної кодової бази програмний код проходить перевірку іншими учасниками команди, що дозволяє своєчасно виявляти помилки, контролювати відповідність програмного коду встановленим стандартам та забезпечувати архітектурну цілісність системи.

Процес рецензування коду також сприяє підвищенню якості програмного забезпечення, покращенню підтримуваності кодової бази та формуванню єдиного підходу до реалізації функціональності в межах проєкту. Особливо важливим це є у випадках використання спільної роботи над серверною логікою, інтеграцією баз даних та реалізацією Application Programming Interface (API) вебзастосунку.

Окрему роль у сучасних процесах управління командною роботою відіграє автоматизація перевірки програмного коду. Інтеграція систем безперервної

інтеграції дозволяє автоматично запускати тестування, перевірку стилу програмування та аналіз стабільності програмного забезпечення після внесення змін до репозиторію. Такий підхід дозволяє зменшити ризик інтеграції помилкового коду та забезпечити стабільність процесу розробки.

У процесі реалізації сучасних ІТ-проєктів важливим аспектом є також ефективна організація комунікації між учасниками команди. Для цього використовуються регулярні обговорення задач, планування ітерацій, демонстрації результатів роботи та аналіз виконаних етапів розробки. Це дозволяє своєчасно виявляти проблеми, коригувати процес реалізації функціональності та забезпечувати узгодженість дій між усіма учасниками проєкту.

Таким чином, використання сучасних методів управління командною роботою, гнучких підходів до організації процесів розробки та засобів автоматизації дозволяє підвищити ефективність реалізації ІТ-проєктів, забезпечити стабільність процесу створення програмного забезпечення та покращити якість кінцевого програмного продукту.

2.3 Методи управління змінами та контролю версій програмного коду

У процесі розробки програмного забезпечення зміни є невід'ємною складовою, оскільки вимоги до функціональності системи можуть уточнюватися протягом усього життєвого циклу проєкту. Це особливо характерно для програмних систем у сфері музики, де алгоритми обробки або генерації звуку можуть постійно вдосконалюватися.

Для ефективного управління змінами використовуються системи контролю версій, серед яких найбільш поширеною є Git [13]. Вона дозволяє відстежувати всі зміни у програмному коді, зберігати історію модифікацій та організовувати

паралельну роботу кількох розробників.

Для організації процесу розробки широко застосовуються стратегії розгалуження, зокрема Gitflow [14]. Вона передбачає використання окремих гілок для реалізації нових функцій, підготовки релізів та термінових виправлень, що дозволяє підтримувати стабільність основної гілки. А також дозволяє інтегрувати процеси автоматизованого тестування та CI/CD у загальну систему управління змінами програмного забезпечення.

Крім того, важливим елементом управління змінами є проведення перевірки коду та використання механізму Pull Request. Це забезпечує контроль якості коду, покращує його читабельність та сприяє узгодженню архітектурних рішень. Для запобігання внесенню нестабільних змін до основних гілок проєкту можуть використовуватись механізми `protected branches` та обов'язкове проходження автоматизованих перевірок перед інтеграцією змін.

Отже, використання систем контролю версій та відповідних стратегій управління змінами є необхідною умовою забезпечення стабільності, якості та прозорості процесу розробки програмного забезпечення.

2.4 Методи автоматизації процесів тестування та розгортання

Автоматизація процесів розробки програмного забезпечення є одним із ключових напрямів підвищення ефективності сучасних ІТ-проєктів. У сучасних умовах розробка веборієнтованих систем та цифрових платформ потребує швидкого впровадження нової функціональності, стабільності процесів оновлення програмного забезпечення та мінімізації кількості помилок, пов'язаних із ручним виконанням рутинних операцій. Для досягнення цих цілей використовуються підходи DevOps, автоматизоване тестування, контейнеризація та процеси

безперервної інтеграції й доставки програмного забезпечення.

Однією з основних складових автоматизації є автоматизоване тестування програмного забезпечення. Даний підхід дозволяє перевіряти працездатність системи після внесення змін до програмного коду та своєчасно виявляти помилки на ранніх етапах розробки. Автоматизація тестування підвищує стабільність програмного забезпечення та дозволяє зменшити ризик виникнення критичних помилок після розгортання нових версій системи.

У сучасних ІТ-проєктах використовуються різні види автоматизованого тестування, зокрема модульне, інтеграційне та функціональне тестування. Модульне тестування дозволяє перевіряти окремі компоненти програмної системи ізольовано один від одного, тоді як інтеграційне тестування забезпечує перевірку взаємодії між різними модулями та сервісами системи. Функціональне тестування використовується для перевірки коректності реалізації бізнес-логіки та роботи API програмного застосунку.

Для автоматизації тестування використовуються спеціалізовані фреймворки та інструменти, зокрема Pytest, unittest, Ruff, Bandit та інші засоби статичного аналізу програмного коду. Використання подібних інструментів дозволяє автоматизувати перевірку стилю програмування, виявлення потенційних помилок, аналіз безпеки залежностей та контроль якості кодової бази.

Важливу роль у сучасних процесах автоматизації відіграють практики CI/CD [15]. Підхід Continuous Integration (CI) передбачає автоматичне тестування та перевірку програмного коду після внесення змін до репозиторію. Це дозволяє своєчасно виявляти помилки інтеграції та забезпечувати стабільність основної кодової бази.

Підхід Continuous Delivery та Continuous Deployment (CD) орієнтований на автоматизацію процесів створення, оновлення та розгортання нових версій програмного забезпечення. Використання автоматизованих CD-процесів дозволяє значно скоротити час між реалізацією функціональності та її появою у production-

середовищі, а також мінімізувати вплив людського фактору на процес оновлення системи.

Для реалізації CI/CD-процесів широко використовуються спеціалізовані платформи автоматизації, зокрема GitHub Actions, GitLab CI/CD, Jenkins та інші інструменти автоматизованого розгортання [16]. Дані системи дозволяють створювати автоматизовані пайплайни, які виконують перевірку програмного коду, запуск тестування, збірку контейнерів, аналіз безпеки та автоматизоване розгортання програмного забезпечення. На рисунку 2.4 наведено загальну схему автоматизованого CI/CD-процесу.

Автоматизовані пайплайни можуть включати:

- перевірку стилю програмування;
- статичний аналіз програмного коду;
- запуск автоматизованих тестів;
- перевірку безпеки залежностей;
- збірку Docker-образів;
- публікацію контейнерів у registry;
- автоматизоване оновлення середовища виконання системи.

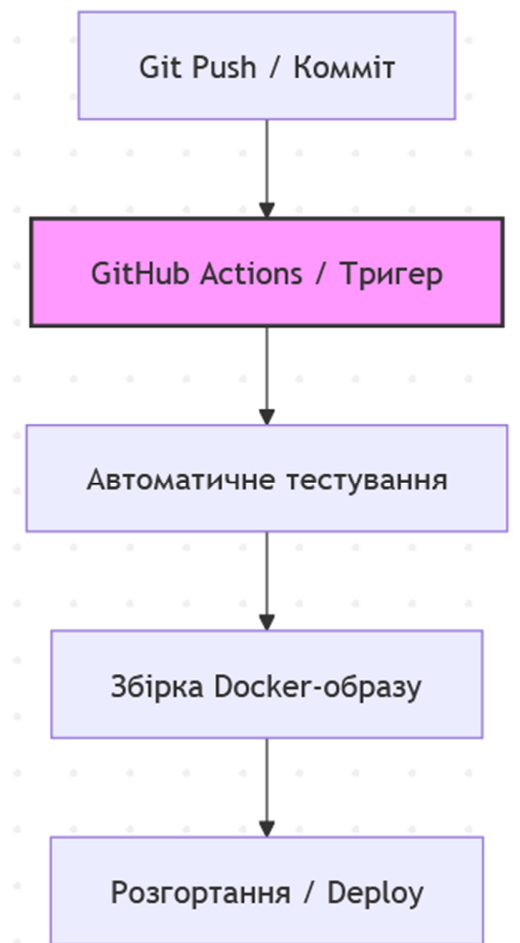


Рисунок 2.3 – Схема автоматизованого CI/CD-процесу

Важливим елементом автоматизації процесів розробки є контейнеризація програмного забезпечення. Однією з найбільш поширених технологій контейнеризації є Docker, який дозволяє стандартизувати середовище виконання застосунку та забезпечити однакову конфігурацію системи у середовищах розробки, тестування та production [17].

Використання Docker дозволяє ізолювати окремі сервіси програмної системи, централізовано управляти залежностями та спростити процес розгортання програмного забезпечення. Контейнеризація також забезпечує високу відтворюваність середовища виконання та дозволяє мінімізувати ризики виникнення помилок, пов'язаних із відмінностями конфігурацій між різними

серверами.

Для організації взаємодії між контейнерами можуть використовуватись системи оркестрації та управління сервісами, зокрема Docker Compose. Даний підхід дозволяє автоматизувати запуск багатокомпонентних систем, централізовано управляти контейнерами та забезпечувати стабільну взаємодію між окремими сервісами програмного застосунку.

Особливого значення автоматизація процесів тестування та розгортання набуває у вебплатформах для роботи з музичним контентом, де важливими факторами є стабільність роботи сервісу, безперервність доступу користувачів до системи та швидкість впровадження нової функціональності. Використання автоматизованих підходів дозволяє забезпечити контроль якості програмного забезпечення, скоротити час розгортання нових версій системи та підвищити надійність програмної платформи.

Таким чином, використання методів автоматизації тестування, контейнеризації та CI/CD-процесів дозволяє комплексно підвищити ефективність процесів розробки програмного забезпечення, забезпечити стабільність функціонування системи та інтегрувати процеси розробки й розгортання у єдину автоматизовану інфраструктуру управління ІТ-проєктом.

2.5 Обґрунтування вибору підходів до організації процесів розробки

На основі проведеного аналізу сучасних підходів до організації процесів розробки програмного забезпечення було визначено набір методів та інструментів, які є найбільш доцільними для реалізації ІТ-проєктів у сфері створення музичних композицій. Особливістю таких проєктів є необхідність забезпечення швидкої адаптації до змін функціональних вимог, стабільної роботи мультимедійних

компонентів, підтримки великої кількості користувацького контенту та безперервного розвитку програмної системи.

У межах дослідження встановлено, що використання гнучких методологій Agile є найбільш ефективним підходом для організації процесів розробки музичних вебплатформ. На відміну від каскадної моделі, Agile-підходи забезпечують можливість поступового розвитку функціональності системи, швидкого внесення змін та адаптації до нових вимог користувачів. Це є особливо важливим для платформ, пов'язаних із музичним контентом, оскільки функціональні можливості таких систем можуть змінюватися під впливом розвитку цифрових музичних сервісів, змін користувацьких потреб та появи нових технологій роботи з аудіо.

Для організації командної взаємодії доцільним є використання підходів Scrum та Kanban, які дозволяють забезпечити гнучке планування задач, контроль процесу розробки та прозорість виконання робіт. Ітеративна модель розробки дозволяє поступово реалізовувати окремі компоненти системи, зокрема модулі управління музичним контентом, механізми взаємодії між користувачами та засоби авторизації. Крім того, використання сучасних систем управління задачами забезпечує централізований контроль процесу розробки та покращує координацію між учасниками проєкту.

Важливу роль у забезпеченні стабільності процесу розробки відіграють системи контролю версій та підходи до управління змінами програмного коду. Використання Git та моделі GitFlow дозволяє організувати паралельну розробку окремих компонентів системи, мінімізувати ризики виникнення конфліктів та забезпечити стабільність production-версії застосунку. Для музичних платформ це є особливо важливим, оскільки одночасно можуть розроблятися різні функціональні модулі, пов'язані із завантаженням аудіофайлів, обробкою метаданих, взаємодією користувачів та адмініструванням системи.

Окрему увагу у межах дослідження приділено DevOps-підходам та автоматизації процесів розробки. Використання контейнеризації Docker, CI/CD-

пайплайнів та автоматизованого тестування дозволяє забезпечити стабільність середовищ виконання, прискорити процес інтеграції нової функціональності та мінімізувати ризики виникнення помилок під час розгортання системи. Для платформ, що працюють із мультимедійним контентом, це має критичне значення, оскільки навіть незначні конфігураційні помилки можуть впливати на стабільність обробки аудіоданих та доступність сервісу для користувачів.

Таким чином, поєднання Agile-методологій, сучасних підходів до управління змінами програмного коду та DevOps-практик дозволяє забезпечити ефективну організацію процесів розробки IT-проектів у сфері створення музичних композицій. Обрані підходи забезпечують гнучкість процесу розробки, стабільність програмної системи, ефективну командну взаємодію та можливість швидкого масштабування функціональності платформи відповідно до потреб користувачів.

3 ПРАКТИЧНЕ ДОСЛІДЖЕННЯ ТА РЕАЛІЗАЦІЯ МЕТОДІВ УПРАВЛІННЯ ПРИ РОЗРОБЦІ МУЗИЧНОЇ ПЛАТФОРМИ

3.1 Проєктування функціональної структури та архітектури системи

У межах практичного дослідження було спроектовано та реалізовано програмну систему «MicroMusicBlog» – вебплатформу для управління музичним контентом та взаємодії між користувачами. Основною метою розробки системи є практична демонстрація ефективності сучасних методів управління та розробки ІТ-проєктів, зокрема автоматизації процесів розробки, контейнеризації середовища виконання та впровадження механізмів безперервної інтеграції та доставки програмного забезпечення.

Архітектура системи побудована за принципом монолітної архітектури із розділенням на функціональні рівні (Layered Architecture). Такий підхід дозволяє забезпечити достатню гнучкість програмної системи, спростити підтримку програмного коду та підвищити ефективність процесів розробки й тестування. Крім того, монолітна архітектура є доцільною для невеликих та середніх проєктів, оскільки забезпечує швидшу реалізацію функціональності та спрощує процес контейнеризації застосунку.

Для реалізації серверної частини системи було використано мову програмування Python версії 3.13, вибір якої обумовлений високою швидкістю розробки, широкою екосистемою бібліотек та зручністю інтеграції із сучасними засобами автоматизації процесів розробки [18].

Основним фреймворком програмної системи обрано FastAPI, який забезпечує підтримку асинхронної обробки запитів, високу продуктивність та спрощує створення REST API. Використання FastAPI також дозволяє автоматизувати генерацію документації API та забезпечує зручну інтеграцію із системами тестування й контейнеризації.

Для збереження реляційних даних у системі використовується PostgreSQL [19]. Дана система керування базами даних забезпечує надійне збереження інформації про користувачів, музичний контент, ролі доступу та взаємодію між користувачами системи.

Для оптимізації процесів обробки даних та кешування часто використовуваної інформації у проєкті застосовується Redis. Крім кешування, Redis використовується як допоміжний компонент для організації асинхронної взаємодії між окремими процесами системи та управління чергами задач.

Відповідно до поставлених задач, програмна система включає декілька основних функціональних модулів:

- модуль аутентифікації та авторизації користувачів;
- модуль управління музичним контентом;
- модуль взаємодії між користувачами;
- модуль управління ролями та доступом до функціоналу системи.

Модуль аутентифікації та авторизації реалізовано із використанням JWT-токенів, що дозволяє забезпечити безпечне управління доступом до ресурсів системи та реалізувати розмежування прав між користувачами платформи, авторами музичного контенту та адміністраторами.

Модуль управління музичним контентом забезпечує завантаження, обробку та збереження музичних композицій, а також управління метаданими аудіофайлів, зокрема назвами треків, описами, жанрами та інформацією про авторів. Реалізація даного модуля дозволяє організувати централізовану систему роботи з музичним контентом у межах вебплатформи.

Для підтримки взаємодії між користувачами реалізовано механізми коментування, оцінювання композицій та зворотного зв'язку. Це створює елементи соціальної взаємодії між учасниками музичної платформи та спрощує обмін контентом між авторами й користувачами системи.

У межах проєктування системи також було розроблено структуру бази даних,

яка включає таблиці користувачів, музичних композицій та коментарів. Для управління змінами структури бази даних використовуються механізми міграцій, реалізовані за допомогою Alembic [20]. Це дозволяє централізовано контролювати зміни структури даних та інтегрувати процес оновлення бази даних у загальний процес управління версіями програмного забезпечення.

Загальна архітектура системи представлена на рис. 3.1. Архітектурне рішення базується на чіткому розділенні рівнів відповідальності. Рівень API на базі FastAPI забезпечує взаємодію з клієнтом, тоді як логіка управління контентом та користувачами винесена в окремі модулі. Використання Alembic для міграцій, інтегрованого в загальну архітектуру, дозволяє автоматизувати оновлення структури даних. Весь серверний стек ізольований у Docker-контейнері, що спрощує моніторинг інфраструктури та забезпечує стабільність роботи компонентів PostgreSQL та Redis у межах єдиного середовища розгортання [21].

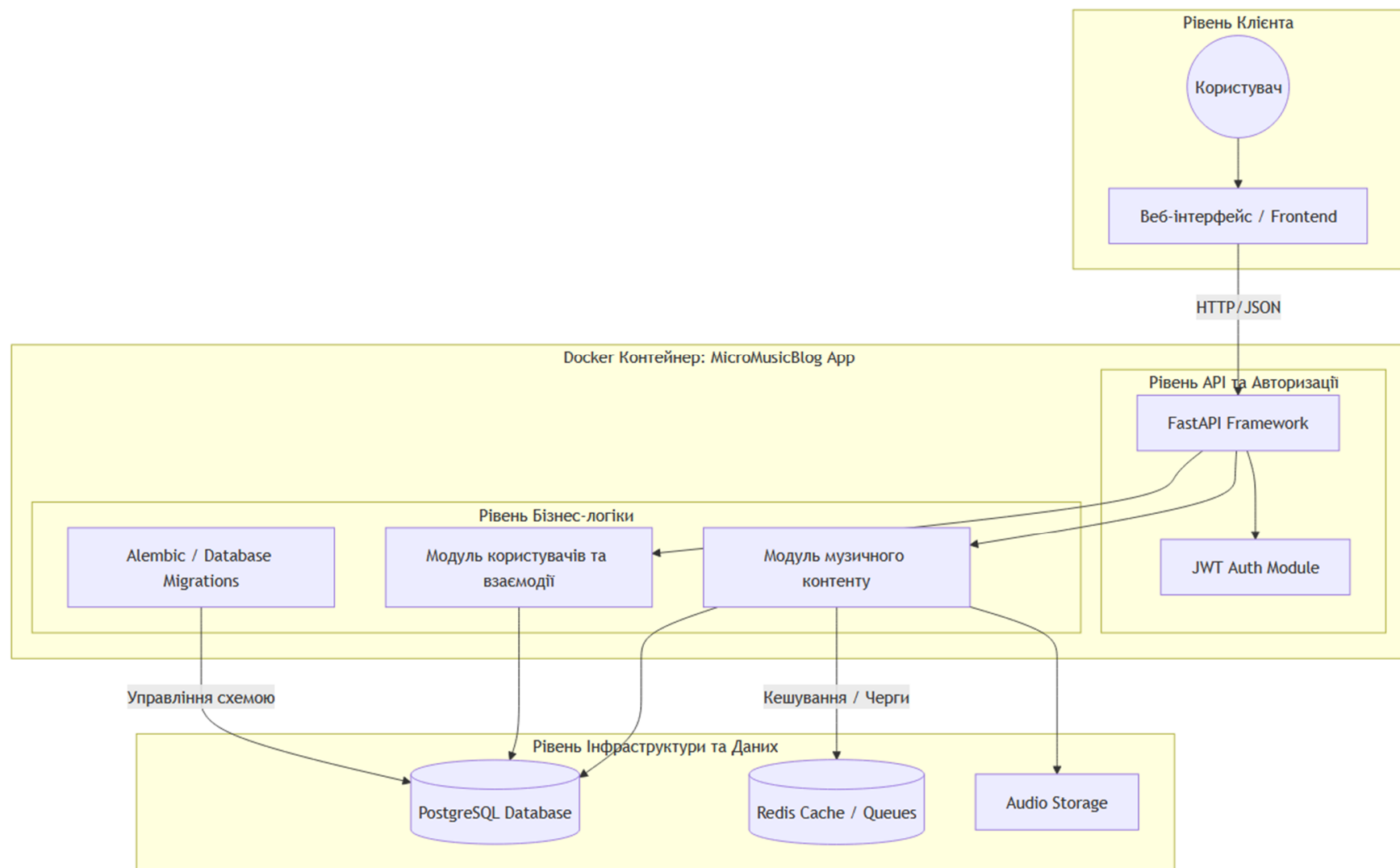


Рисунок 3.1 – Архітектура системи «MicroMusicBlog»

3.2 Організація контролю версій та управління потоками розробки

Ефективне управління процесами розробки програмного забезпечення потребує використання сучасних підходів до контролю версій, організації командної взаємодії та автоматизації інтеграції змін. У межах реалізації програмної системи «MicroMusicBlog» для управління вихідним кодом було використано децентралізовану систему контролю версій Git, яка забезпечує централізований контроль змін, підтримку командної розробки та інтеграцію із засобами автоматизації CI/CD.

Використання Git дозволило реалізувати структурований процес управління потоками розробки, мінімізувати ризики виникнення конфліктів під час інтеграції нової функціональності та забезпечити стабільність програмної системи на всіх етапах життєвого циклу проєкту.

Для організації процесу розробки було використано модифіковану модель GitFlow, адаптовану до сучасних DevOps-практик та процесів безперервної інтеграції. Даний підхід передбачає розділення стабільних, інтеграційних та функціональних гілок, що дозволяє ізолювати процес розробки окремих компонентів системи та забезпечити контроль якості програмного забезпечення перед інтеграцією змін.

У межах проєкту використовувались такі основні типи гілок:

- `main` – стабільна `production-ready` гілка, яка містить лише протестовані версії програмного забезпечення;
- `develop` – інтеграційна гілка для тестування та об'єднання нової функціональності;
- `feature`-гілки – окремі ізольовані гілки для реалізації нових модулів та функціональних можливостей системи.

Гілка `main` використовується для збереження стабільних версій застосунку,

готових до розгортання у production-середовищі. Інтеграція змін до даної гілки виконується лише після проходження всіх етапів автоматизованої перевірки та тестування. Для управління версіями програмного забезпечення також використовувалось тегування релізів, що дозволяє зберігати історію стабільних станів системи та спрощує процес повернення до попередніх версій у разі виникнення помилок. Саме у main-гілці зберігались стабільні версії музичної платформи, що включали перевірені механізми роботи з музичним контентом, авторизацією користувачів та взаємодією між учасниками системи.

Гілка develop виступає середовищем попередньої інтеграції та тестування нової функціональності. У даній гілці виконується перевірка сумісності змін між різними компонентами системи перед їх інтеграцією до production-версії застосунку. Зокрема, у develop-гілці проводилась перевірка взаємодії модулів завантаження музичних композицій, системи коментарів та механізмів кешування даних перед випуском нових версій платформи.

Для реалізації окремих функціональних можливостей створювались feature-гілки, що дозволяє ізолювати процес розробки окремих модулів та мінімізувати вплив незавершених змін на стабільність системи. Такий підхід спрощує процес паралельної розробки та забезпечує більш ефективне управління змінами програмного коду. Наприклад, у межах проєкту окремо використовувались feature-гілки для реалізації системи аутентифікації користувачів, модулю завантаження музичних композицій та механізмів взаємодії між користувачами. Це дозволило ізолювати процес розробки окремих компонентів системи та забезпечити паралельне виконання задач без впливу на стабільність основної кодової бази.

Важливою частиною організації процесу розробки стало використання механізму Pull Request. Інтеграція змін до основних гілок системи виконувалась виключно через Pull Request після проходження автоматизованих перевірок та рецензування програмного коду.

Використання Pull Request та Code Review дозволяє:

- контролювати відповідність програмного коду встановленим стандартам;
- виявляти помилки на ранніх етапах розробки;
- перевіряти коректність реалізації бізнес-логіки;
- забезпечувати архітектурну цілісність системи;
- підвищувати підтримуваність програмного забезпечення.

Для запобігання внесенню нестабільних змін до основних гілок використовувались механізми `protected branches`, які забороняють пряме внесення змін до гілок `main` та `develop` без проходження `Pull Request` і автоматизованих CI-перевірок. Такий підхід дозволяє мінімізувати ризик потрапляння помилкового або нестабільного коду до `production`-середовища.

У процесі рецензування програмного коду також використовувались засоби автоматичної перевірки стилю програмування та форматування коду відповідно до стандарту `PEP8`. Це дозволяє підтримувати єдиний стиль програмного коду та спрощує подальшу підтримку системи.

Окрім вихідного коду програмної системи, через `Git` здійснюється управління `Docker`-конфігураціями, файлами середовища розробки, конфігураціями CI/CD-пайплайнів та файлами міграцій бази даних. Такий підхід забезпечує централізоване управління всіма компонентами проєкту та дозволяє підтримувати синхронізацію середовищ розробки, тестування та розгортання.

Для контролю змін структури бази даних використовувався інструмент `Alembic`, який реалізує систему міграцій для `PostgreSQL`. Використання міграцій дозволяє інтегрувати процес оновлення структури бази даних у загальний процес управління версіями програмного забезпечення та забезпечує узгодженість структури даних між різними середовищами виконання системи.

Таким чином, використання `Git` у поєднанні з сучасними підходами до організації потоків розробки дозволило забезпечити стабільність процесу створення програмного забезпечення, підвищити ефективність командної взаємодії та інтегрувати процес управління версіями у загальну систему автоматизації

розробки та розгортання програмної системи.

На рисунку 3.2 можна побачити схему GitFlow, а на рисунку 3.3 – структуру репозиторію проєкта «MicroMusicBlog».

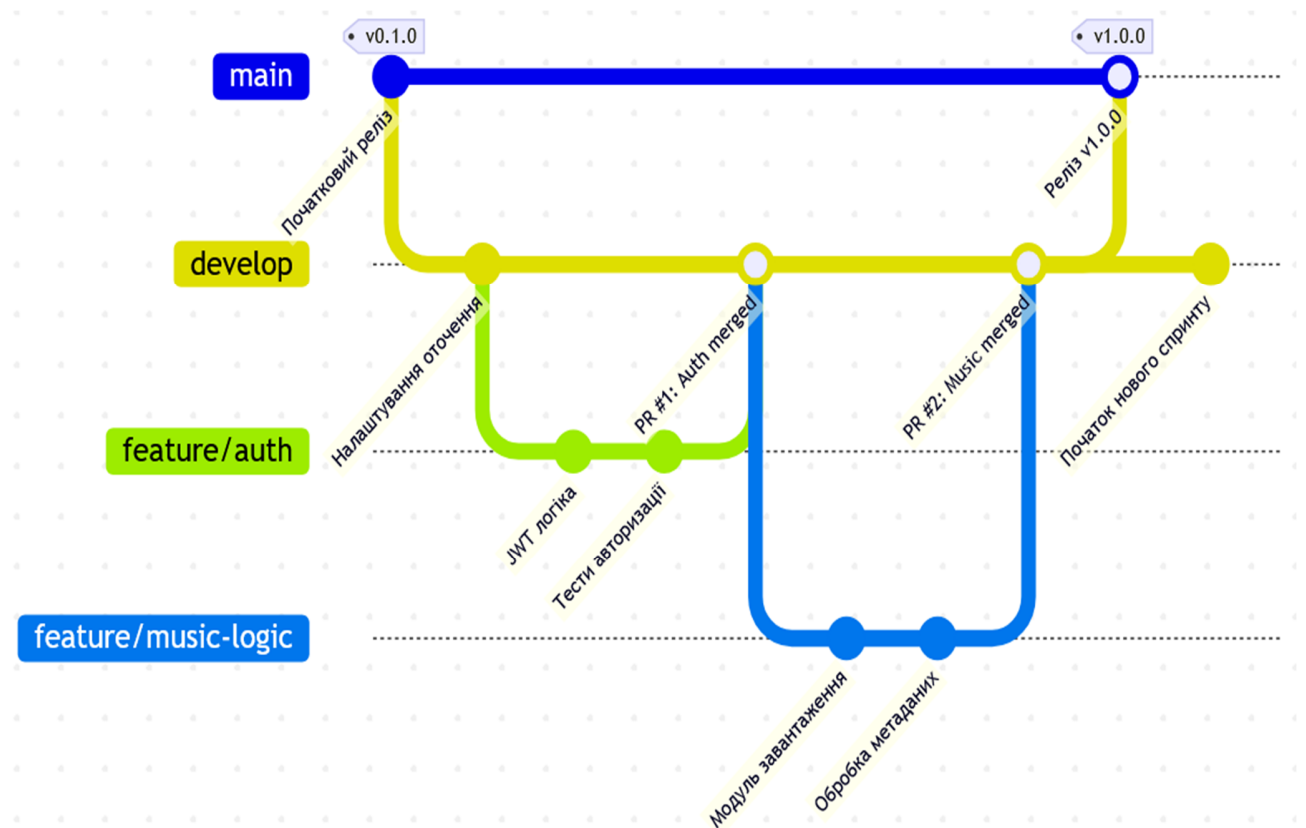


Рисунок 3.2 – Схема GitFlow проєкту

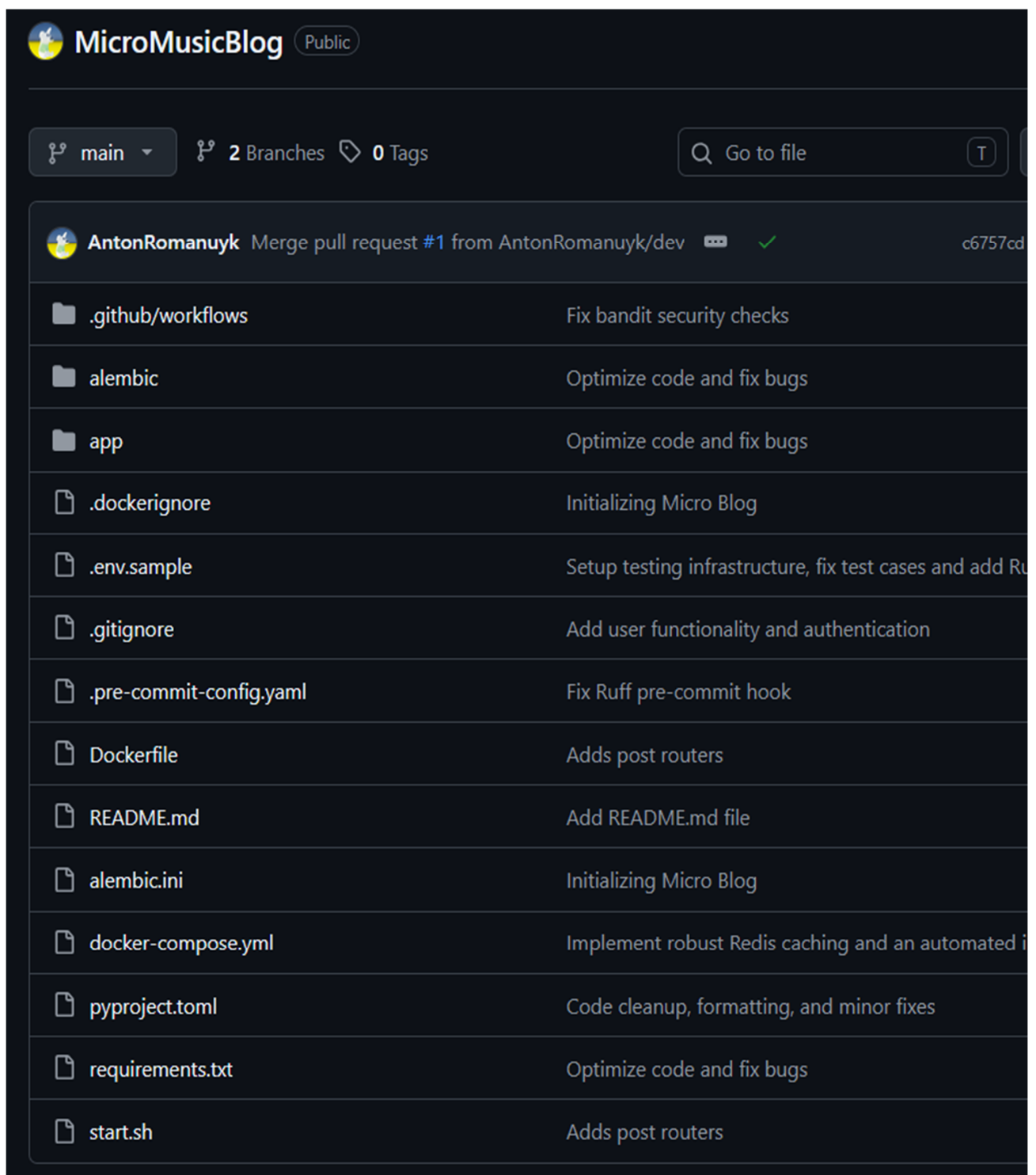


Рисунок 3.3 – Структура репозиторію

3.3 Контейнеризація середовища розробки та розгортання з використанням Docker

У сучасних процесах розробки програмного забезпечення важливим аспектом управління ІТ-проектами є забезпечення стабільності середовища виконання застосунку на всіх етапах життєвого циклу системи. Однією з основних проблем традиційного підходу до розгортання програмного забезпечення є відмінність конфігурацій між середовищами розробки, тестування та production, що може призводити до виникнення помилок сумісності та нестабільної роботи системи.

Для мінімізації подібних ризиків у межах реалізації програмної системи «MicroMusicBlog» було використано технологію контейнеризації на базі Docker. Використання Docker дозволило стандартизувати середовище виконання застосунку, спростити процес розгортання та забезпечити однакову поведінку системи незалежно від апаратного або програмного оточення.

Контейнеризація виступає не лише засобом запуску програмного забезпечення, а й важливим інструментом управління ІТ-інфраструктурою та автоматизації процесів розробки. Застосування Docker у проєкті дозволило забезпечити ізоляцію сервісів, централізоване управління залежностями та інтеграцію із процесами CI/CD.

У межах проєкту використання контейнеризації дозволило вирішити такі задачі:

- забезпечення однаковості середовищ розробки, тестування та production;
- ізоляція залежностей між окремими сервісами системи;
- спрощення процесу розгортання програмного забезпечення;
- зменшення ризику виникнення конфігураційних помилок;
- підвищення швидкості підготовки середовища для нових учасників

команди;

- спрощення масштабування окремих компонентів системи.

Для контейнеризації серверної частини системи було створено Dockerfile, який визначає процес автоматизованої збірки образу застосунку. У проєкті використовувався підхід multi-stage build, що дозволяє розділити етапи збірки та виконання програмного забезпечення. Такий підхід дозволяє зменшити розмір фінального Docker-образу, оптимізувати використання ресурсів та підвищити безпеку системи за рахунок виключення зайвих компонентів із production-середовища.

Лістинг 3.1 – Створення Dockerfile

```
FROM python:3.13
ENV PYTHONUNBUFFERED 1
WORKDIR /app
COPY requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt
COPY . /app
RUN chmod +x /app/start.sh
ENTRYPOINT ["/app/start.sh"]
```

У наведеному Dockerfile використовується офіційний Docker-образ Python версії 3.13 як базове середовище виконання застосунку. На початковому етапі за допомогою директиви WORKDIR визначається робоча директорія контейнера /app, у межах якої надалі виконуються всі операції.

Файл requirements.txt копіюється до контейнера окремо перед копіюванням вихідного коду, що дозволяє ефективніше використовувати механізм кешування шарів Docker під час повторної збірки образу. Після цього виконується автоматичне встановлення залежностей програмного забезпечення за допомогою пакетного менеджера `pip`.

Наступним етапом до контейнера копіюється вихідний код програмної

системи. Для автоматизації запуску застосунку використовується скрипт `start.sh`, якому надаються права виконання через команду `chmod +x`. Завершальним етапом є визначення точки входу контейнера (`ENTRYPOINT`), що забезпечує автоматичний запуск серверної частини системи після старту контейнера.

Використання `Dockerfile` дозволило стандартизувати процес створення середовища виконання застосунку та забезпечити відтворюваність процесу розгортання незалежно від платформи запуску.

Для організації взаємодії між контейнерами використовувався `Docker Compose`, який забезпечує централізоване управління сервісами системи та дозволяє автоматизувати запуск усіх компонентів програмної платформи.

Лістинг 3.2 – Створення `Docker Compose`

```
version: '4.46'

services:
  api:
    build:
      context: .
      dockerfile: Dockerfile
    command: sh start.sh
    volumes:
      - ../app
    ports:
      - 8000:8000
    environment:
      DATABASE_HOSTNAME: "${DATABASE_HOSTNAME}"
      DATABASE_PORT: "${DATABASE_PORT}"
      DATABASE_NAME: "${DATABASE_NAME}"
      DATABASE_USER: "${DATABASE_USER}"
      DATABASE_PASSWORD: "${DATABASE_PASSWORD}"
      SECRET_KEY: "${SECRET_KEY}"
    depends_on:
      - db
      - redis

  db:
    image: postgres:17.0
    environment:
```

```
POSTGRES_DB: "${DATABASE_NAME}"
POSTGRES_USER: "${DATABASE_USER}"
POSTGRES_PASSWORD: "${DATABASE_PASSWORD}"

redis:
  image: redis:latest
```

Для організації взаємодії між контейнерами використовувався Docker Compose, який забезпечує централізоване управління сервісами системи та автоматизує запуск усіх компонентів програмної платформи.

У межах конфігурації було визначено три основні сервіси:

- api – контейнер із FastAPI-застосунком;
- db – контейнер із системою керування базами даних PostgreSQL;
- redis – контейнер для кешування даних та підтримки асинхронної взаємодії між компонентами системи.

Сервіс api виконує збірку контейнера на основі створеного Dockerfile та забезпечує запуск серверної частини програмної системи. Для спрощення локальної розробки використовується механізм volumes, що дозволяє синхронізувати вихідний код між локальним середовищем та контейнером у режимі реального часу.

У конфігурації також визначено змінні оточення, через які передаються параметри підключення до бази даних, JWT-конфігурації та інші службові параметри системи. Такий підхід дозволяє уникнути зберігання конфіденційних даних безпосередньо у вихідному коді застосунку.

Сервіс db реалізовано на базі офіційного Docker-образу PostgreSQL версії 17.0. Для забезпечення збереження даних між перезапусками контейнерів використовується механізм Docker Volumes.

Сервіс redis використовує офіційний Docker-образ Redis та забезпечує кешування даних і підтримку асинхронної взаємодії між компонентами системи.

Використання Docker Compose дозволило організувати внутрішню мережеву взаємодію між контейнерами, централізувати управління залежностями сервісів та

автоматизувати процес запуску локальної інфраструктури проєкту.

Окрему увагу було приділено управлінню конфігураціями середовища. Для забезпечення безпеки та гнучкості розгортання чутливі параметри системи, зокрема паролі бази даних, JWT-ключі та службові конфігурації, були винесені до окремих файлів змінних оточення (.env). Такий підхід дозволяє уникнути зберігання конфіденційних даних у репозиторії та спрощує адаптацію системи до різних середовищ виконання.

Використання змінних оточення також забезпечує можливість використання однакових Docker-образів у середовищах розробки, тестування та production без необхідності зміни конфігурації застосунку.

Лістинг 3.3 – Створення .dockerignore

```
__pycache__  
*.pyc  
.env  
venv/  
.idea/  
.vscode/  
db.sqlite3  
media
```

Для оптимізації процесу збірки Docker-образів у проєкті використовувався файл .dockerignore, який визначає перелік файлів і директорій, що не повинні включатися до контексту збірки контейнера.

До таких файлів належать службові директорії середовищ розробки, кеш Python, локальні віртуальні середовища, конфігураційні файли редакторів коду, файли баз даних та змінні оточення. Використання .dockerignore дозволяє зменшити розмір контексту збірки, прискорити створення Docker-образів та підвищити безпеку системи шляхом виключення конфіденційних даних із контейнерів.

Важливою перевагою контейнеризації стало скорочення часу розгортання

системи. Завдяки автоматизації процесу збірки та запуску контейнерів підготовка середовища виконується значно швидше порівняно з ручним встановленням залежностей та налаштуванням сервісів. Це особливо важливо в умовах командної розробки та безперервної інтеграції програмного забезпечення.

Крім того, використання Docker дозволило спростити інтеграцію програмної системи з CI/CD-пайплайнами, оскільки всі середовища виконання використовують однакову конфігурацію контейнерів. Це забезпечує стабільність автоматизованого тестування та зменшує ризик виникнення помилок під час розгортання нових версій застосунку.

Таким чином, використання Docker та контейнеризації у межах проєкту «MicroMusicBlog» дозволило стандартизувати середовище розробки та розгортання, підвищити стабільність програмної системи, спростити процес управління інфраструктурою та інтегрувати процеси розробки із сучасними DevOps-практиками.

3.4 Автоматизація процесів безперервної інтеграції та доставки програмного забезпечення

3.4.1 Обґрунтування архітектури та інструментарію автоматизованих процесів CI/CD

Одним із ключових елементів сучасних підходів до управління ІТ-проєктами є автоматизація процесів інтеграції, тестування та розгортання програмного забезпечення. У межах реалізації програмної системи «MicroMusicBlog» було впроваджено процеси безперервної інтеграції та безперервної доставки (CI/CD), що дозволило автоматизувати основні етапи життєвого циклу програмного забезпечення та підвищити стабільність процесу розробки.

Основною метою впровадження CI/CD є мінімізація ручних операцій, зменшення кількості помилок під час інтеграції змін та забезпечення швидкого й контрольованого розгортання нових версій застосунку. Автоматизація даних процесів також дозволяє підвищити прозорість розробки та забезпечити постійний контроль якості програмного забезпечення.

Для реалізації процесів CI/CD у проєкті використовувалась платформа GitHub Actions, яка забезпечує інтеграцію автоматизованих сценаріїв безпосередньо з Git-репозиторієм та дозволяє запускати процеси перевірки й розгортання після внесення змін до програмного коду. Приклад роботи платформи наведено на рисунку 3.4.

Загальна структура CI/CD-процесу включає:

- автоматичну перевірку програмного коду;
- запуск тестування;
- перевірку стабільності Docker-збірки;
- автоматичне створення Docker-образів;
- публікацію образів у container registry;
- автоматизоване оновлення середовища розгортання.

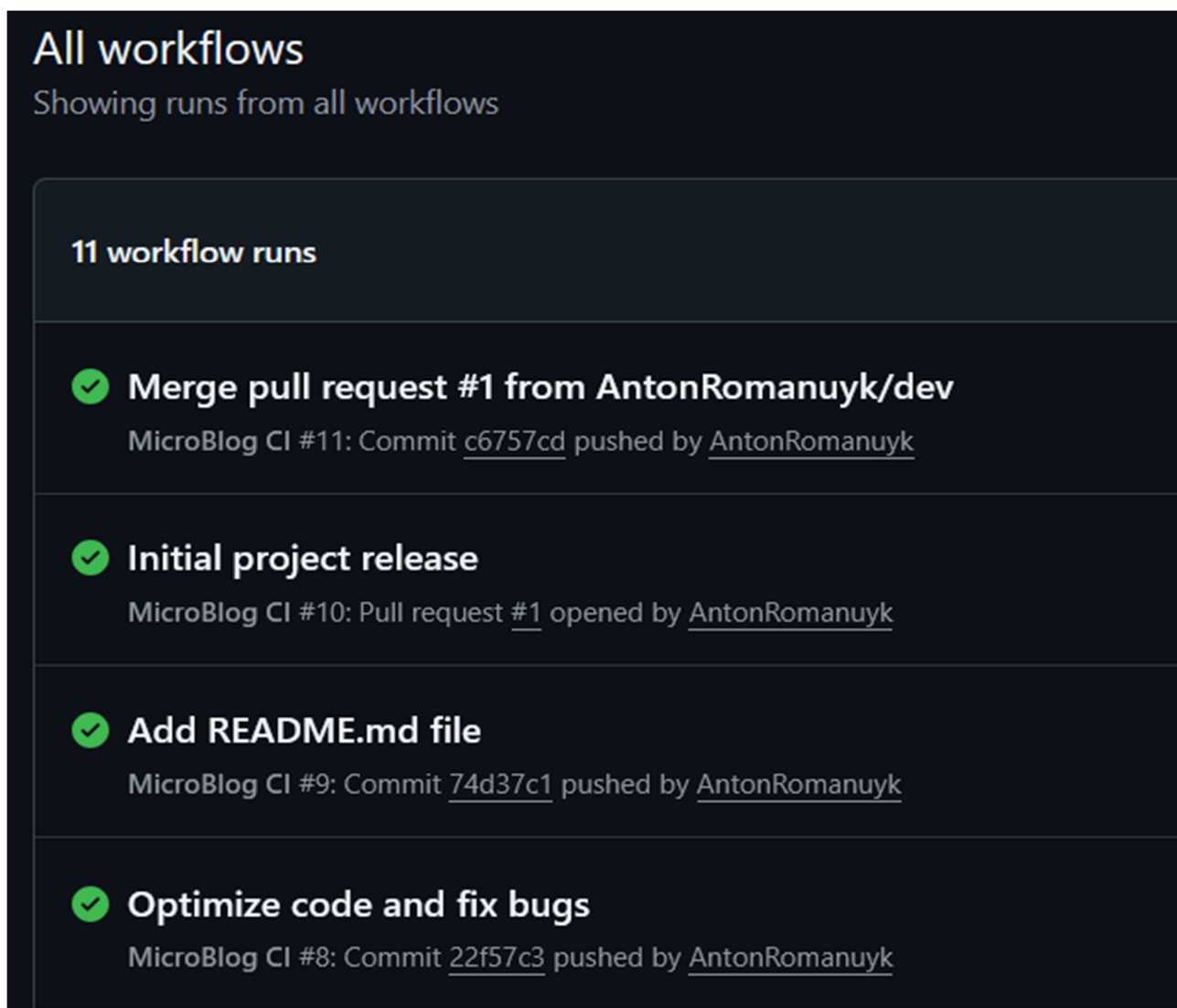


Рисунок 3.4 – Журнал виконання автоматизованих сценаріїв CI/CD у середовищі GitHub Actions

3.4.2 Автоматизація безперервної інтеграції (CI)

Процеси безперервної інтеграції були налаштовані таким чином, щоб кожне внесення змін до репозиторію або створення Pull Request автоматично запускало процедури перевірки якості програмного коду. Такий підхід дозволяє виявляти помилки на ранніх етапах розробки та запобігати інтеграції нестабільних змін до

основних гілок проєкту.

У межах реалізації CI-пайплайна було автоматизовано такі процеси:

- статичний аналіз програмного коду;
- перевірка форматування та стилю програмування;
- запуск автоматизованих тестів;
- перевірка збірки Docker-контейнерів;
- базовий аналіз безпеки залежностей.

Лістинг 3.4 – Створення фрагмента конфігурації CI-пайплайна GitHub Actions

```
name: MicroBlog CI

on:
  push: {}
  pull_request: {}

jobs:
  MicroMusicBlogCI:
    runs-on: ubuntu-latest

    services:
      db:
        # конфігурація PostgreSQL

      redis:
        # конфігурація Redis

    steps:
      - name: Checkout code
        uses: actions/checkout@v6

      - name: Set up Python
        uses: actions/setup-python@v6
        with:
          python-version: "3.13"
          cache: "pip"

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
```

```

- name: Wait for PostgreSQL
  run: |
    until pg_isready -h localhost -p 5432; do
      echo "Waiting for PostgreSQL..."
      sleep 2
    done

- name: Run Ruff (linter)
  run: |
    ruff check .
    ruff format --check

- name: Apply Migrations
  env:
    # конфігурація бази даних
  run: |
    alembic upgrade head

- name: Security Checks
  run:
    bandit -r app -x app/tests -ll

- name: Run pytest
  env:
    # конфігурація тестового середовища
  run: pytest

```

У межах проєкту процес безперервної інтеграції реалізовано за допомогою GitHub Actions. Автоматизований CI-пайплайн запускається при кожному внесенні змін до репозиторію або створенні Pull Request, що дозволяє забезпечити постійний контроль якості програмного забезпечення.

У конфігурації пайплайна визначено автоматичне розгортання служб PostgreSQL та Redis, необхідних для тестування програмної системи. Після підготовки середовища виконується встановлення залежностей проєкту та запуск автоматизованих перевірок.

Для контролю якості програмного коду використовуються інструменти Ruff та Bandit. Ruff забезпечує статичний аналіз і перевірку форматування коду відповідно до стандартів Python, а Bandit виконує базовий аналіз безпеки програмного забезпечення.

Окремим етапом пайплайна є автоматичне застосування міграцій бази даних за допомогою Alembic та запуск тестування із використанням pytest. Такий підхід дозволяє виявляти помилки на ранніх етапах розробки та запобігати інтеграції нестабільних змін до основних гілок проєкту.

Для перевірки відповідності програмного коду стандартам програмування використовувались інструменти статичного аналізу та форматування коду. Автоматична перевірка дозволяє контролювати дотримання стандартів PEP8, підтримувати єдиний стиль кодової бази та покращувати підтримуваність програмного забезпечення.

Автоматизоване тестування реалізовано із використанням фреймворку pytest. Тестування виконується у тимчасових Docker-контейнерах, що дозволяє забезпечити однакові умови виконання тестів незалежно від середовища запуску. У процесі тестування перевіряється коректність роботи API-ендпоінтів, взаємодія з базою даних PostgreSQL та стабільність асинхронної логіки FastAPI-застосунку.

Окрему увагу було приділено перевірці Docker-конфігурацій та працездатності контейнеризованого середовища. Це дозволяє виявляти помилки конфігурації ще до етапу розгортання системи.

Для підвищення рівня безпеки програмного забезпечення у CI-процес також інтегровано базову перевірку залежностей на наявність відомих вразливостей. Такий підхід дозволяє зменшити ризик використання небезпечних або застарілих компонентів програмного забезпечення.

Інтеграція змін до основних гілок проєкту дозволялась лише після успішного проходження всіх автоматизованих перевірок, що забезпечує стабільність програмної системи та мінімізує ризик потрапляння помилкового коду до production-середовища.

3.4.3 Конвеєр безперервної доставки та розгортання (CD)

Після успішного завершення етапу безперервної інтеграції система автоматично переходить до процесів безперервної доставки та розгортання програмного забезпечення.

Основною задачею CD-процесу є автоматизація створення та оновлення production-ready версій застосунку. Це дозволяє скоротити час доставки нової функціональності та забезпечити передбачуваність процесу розгортання.

У межах реалізації CD-пайплайна було автоматизовано такі етапи:

- збірка Docker-образів;
- маркування версій образів;
- публікація образів у container registry;
- оновлення контейнерів на сервері;
- виконання міграцій бази даних;
- перезапуск сервісів після оновлення.

Лістинг 3.5 – Створення фрагмента конфігурації CD-пайплайна

```
name: MicroMusicBlog CD

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v6

      - name: Build Docker image
```

```
run: |
    docker build -t micromusicblog:latest .

- name: Tag Docker image
  run: |
    docker tag micromusicblog:latest micromusicblog:v1.0

- name: Run database migrations
  run: |
    alembic upgrade head

- name: Restart containers
  run: |
    docker compose down
    docker compose up -d
```

Для автоматизації процесу безперервної доставки було розроблено CD-пайплайн, який забезпечує повний цикл оновлення програмної системи після внесення змін до гілки `main`. У межах цього процесу реалізовано автоматичну збірку Docker-образу застосунку, присвоєння йому версії та подальшу публікацію у container registry.

Після завершення етапу збірки виконуються автоматизовані сценарії розгортання, що включають оновлення контейнеризованого середовища та застосування міграцій бази даних за допомогою Alembic. Завершальним етапом є перезапуск сервісів із використанням оновлених образів через Docker Compose.

Використання CD-процесу дозволяє мінімізувати кількість ручних операцій, скоротити час доставки нових версій системи та підвищити стабільність процесу розгортання. Додатково контейнеризований підхід забезпечує однаковість середовищ виконання на різних інфраструктурах та зменшує ризики, пов'язані з конфігураційними відмінностями.

Також реалізовано можливість швидкого відкату (rollback) до попередньої стабільної версії у разі виникнення помилок після оновлення, що підвищує надійність експлуатації системи та зменшує ймовірність простою сервісу.

3.5 Засоби моніторингу та супроводу програмної системи в процесі експлуатації

Ефективна експлуатація сучасних веб-застосунків потребує впровадження механізмів моніторингу стану системи, збору логів та контролю працездатності сервісів. У межах програмної системи «MicroMusicBlog» були застосовані підходи, що дозволяють забезпечити спостережуваність та оперативне виявлення помилок у процесі роботи застосунку.

Одним із базових механізмів супроводу системи є логування подій. Серверна частина застосунку, реалізована на базі FastAPI, генерує структуровані логи, які містять інформацію про обробку HTTP-запитів (Hypertext Transfer Protocol), помилки виконання, взаємодію з базою даних та інші критичні події системи. У контейнеризованому середовищі Docker забезпечує централізований доступ до логів кожного сервісу за допомогою механізму docker logs, що дозволяє оперативно аналізувати стан окремих компонентів системи.

Такий підхід до логування дозволяє ефективно реалізувати управління помилками (Error Management), оскільки розробник або адміністратор системи має можливість швидко визначити джерело збою та проаналізувати послідовність подій, що призвели до виникнення проблеми.

Окрему роль у забезпеченні стабільності системи відіграють механізми перевірки працездатності сервісів (Health Checks). У межах Docker-контейнерів було налаштовано періодичні перевірки стану ключових компонентів системи, зокрема серверної частини FastAPI, бази даних PostgreSQL та кеш-сервера Redis.

Наявність Health Check механізмів дозволяє автоматично визначати стан кожного сервісу та фіксувати випадки його недоступності. У разі виникнення збоїв контейнерна інфраструктура може реагувати на такі події шляхом перезапуску сервісів або повідомлення про критичний стан системи. Це підвищує надійність

програмної платформи та зменшує час простою системи.

Важливим аспектом супроводу є також контроль використання системних ресурсів. Контейнеризована архітектура дозволяє відстежувати споживання пам'яті, процесорного часу та мережевих ресурсів кожним сервісом окремо. Це є критично важливим для забезпечення стабільної роботи системи при високому навантаженні та досягненні заявленої продуктивності обробки запитів.

Завдяки ізоляції контейнерів можна точно визначити, який компонент системи створює навантаження, та оптимізувати його роботу без впливу на інші частини застосунку. Такий підхід підвищує масштабованість системи та дозволяє ефективно планувати використання інфраструктурних ресурсів.

Окремим елементом моніторингу є автоматично згенерована документація API, яка реалізована у FastAPI за допомогою Swagger UI. Даний інструмент виступає одночасно як засіб документації та інтерфейс тестування API, що дозволяє розробникам і тестувальникам перевіряти роботу ендпоінтів без використання додаткових інструментів.

Swagger UI забезпечує: перегляд структури API-запитів; тестування ендпоінтів у реальному часі; перевірку відповідей сервера; аналіз форматів даних та помилок.

Таким чином, система має вбудований інструмент самодокументації, що значно спрощує процес супроводу та розвитку програмного забезпечення.

У сукупності використані механізми логування, health checks, моніторингу ресурсів та інтерактивної документації API забезпечують комплексний підхід до спостережуваності системи. Це дозволяє своєчасно виявляти та усувати проблеми, підтримувати стабільність роботи платформи та забезпечувати її готовність до експлуатації в умовах реального навантаження.

Загалом, впроваджені засоби моніторингу та супроводу є важливою складовою DevOps-підходу та дозволяють забезпечити контроль над усіма етапами функціонування програмної системи «MicroMusicBlog» після її розгортання.

4 АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ВПРОВАДЖЕНИХ МЕТОДІВ

4.1 Критерії оцінки ефективності процесів розробки та управління версіями

Оцінка ефективності сучасних методів управління ІТ-проектами потребує визначення критеріїв, які дозволяють аналізувати не лише функціональні характеристики програмного забезпечення, але й якість організації процесів розробки, інтеграції змін та супроводу системи. У межах даного дослідження основна увага приділяється оцінці ефективності процесів управління розробкою програмної системи «MicroMusicBlog» із використанням Git, Docker та CI/CD-підходів.

На відміну від традиційного підходу до оцінювання програмних систем, де основними показниками виступають лише продуктивність або швидкодія застосунку, сучасні DevOps-практики передбачають комплексний аналіз стабільності процесів розробки, керованості змін, швидкості інтеграції нової функціональності та рівня автоматизації процесів.

Для систем, орієнтованих на поширення, створення та обробку музичного контенту, зазначені критерії набувають особливого значення, оскільки такі платформи працюють із мультимедійними даними, потоковою передачею контенту, асинхронною обробкою аудіофайлів та високою інтенсивністю змін функціональності. У подібних проєктах навіть незначні помилки конфігурації або нестабільність процесів інтеграції можуть призводити до порушення роботи сервісів зберігання, обробки або публікації музичних композицій.

У межах дослідження ефективність впроваджених методів оцінювалась за такими основними критеріями:

- стабільність програмного коду;
- ефективність управління змінами;

- швидкість інтеграції нової функціональності;
- контроль якості програмного забезпечення;
- швидкість розгортання системи;
- відтворюваність середовища виконання;
- рівень автоматизації процесів розробки;
- можливість масштабування процесів командної роботи.

Одним із ключових критеріїв є стабільність кодової бази. Використання структурованої моделі розгалуження GitFlow дозволяє ізолювати процес розробки нових функціональних можливостей від стабільної версії застосунку. Це зменшує ризик потрапляння нестабільного коду до production-середовища та забезпечує більш контрольований процес інтеграції змін.

Важливим показником ефективності також є швидкість локалізації та усунення помилок. Завдяки використанню системи контролю версій Git кожна зміна програмного коду має власну історію, що дозволяє швидко визначати джерело виникнення помилок та виконувати повернення до попередніх стабільних версій системи у разі необхідності.

Окрему роль у забезпеченні ефективності процесів розробки відіграє контроль конфліктів під час інтеграції змін. Використання feature-гілок та механізму Pull Request дозволяє ізолювати зміни між різними модулями системи та зменшує кількість конфліктів при злитті програмного коду. Це особливо важливо в умовах паралельної розробки декількох компонентів системи.

Ще одним важливим критерієм є швидкість інтеграції та доставки нової функціональності. Автоматизація процесів тестування та розгортання за допомогою CI/CD дозволяє значно скоротити час між завершенням розробки функціональності та її появою у production-середовищі. Це позитивно впливає на загальну ефективність управління проєктом та дозволяє швидше адаптувати систему до нових вимог.

Для оцінки стабільності інфраструктури важливим показником виступає

відтворюваність середовища виконання. Використання Docker-контейнерів дозволяє забезпечити однаковість середовищ розробки, тестування та розгортання, що мінімізує ризики виникнення помилок, пов'язаних із відмінностями конфігурацій системи.

Окремим критерієм оцінки є рівень автоматизації процесів розробки. Автоматичне тестування, перевірка якості коду, збірка Docker-образів та автоматизоване розгортання дозволяють зменшити кількість ручних операцій та мінімізувати вплив людського фактору на стабільність програмної системи.

Важливим аспектом ефективності також є масштабованість процесів командної роботи. Використання структурованої моделі управління версіями дозволяє організувати паралельну розробку декількох функціональних модулів без порушення стабільності основної кодової бази. Це створює передумови для ефективного масштабування команди розробки та спрощує координацію між учасниками проєкту.

Таким чином, оцінка ефективності впроваджених методів управління процесами розробки базується не лише на технічних характеристиках програмної системи, але й на показниках стабільності, керованості, автоматизації та організації командної взаємодії. Саме комплексне поєднання цих критеріїв дозволяє оцінити практичну ефективність сучасних DevOps-підходів у процесі створення програмної системи «MicroMusicBlog».

Варто підкреслити, що з позиції управління IT-проєктами системна реалізація зазначених критеріїв трансформується у стратегічну перевагу при використанні людських ресурсів. Комплексне впровадження цих показників дозволяє кардинально змінити підхід до менеджменту: замість постійного операційного контролю рутинних операцій, керівник проєкту отримує можливість фокусувати зусилля команди на створенні інтелектуальної цінності.

Високий рівень автоматизації та стабілізація процесів вивільняють часовий ресурс розробників, який раніше витрачався на усунення інфраструктурних

помилки або вирішення конфліктів інтеграції. Це створює умови для більш гнучкого управління проєктом, де менеджер може ефективніше розподіляти навантаження та оперативно реагувати на нові вимоги ринку, оскільки технічний фундамент системи стає передбачуваним та надійним.

4.2 Практичний аналіз організації процесів розробки та інтеграції змін

У процесі дослідження було проведено практичний аналіз впливу організованого підходу до управління розробкою на стабільність та керованість програмної системи «MicroMusicBlog». Основною задачею даного етапу стало порівняння традиційного неструктурованого підходу до розробки програмного забезпечення із підходом, що базується на використанні GitFlow, контейнеризації та автоматизованих CI/CD-процесів.

У традиційній моделі розробки, де всі зміни вносяться безпосередньо до однієї основної гілки, процес інтеграції нової функціональності супроводжується підвищеним ризиком виникнення конфліктів, нестабільності програмного коду та порушення працездатності системи. Відсутність ізоляції змін ускладнює паралельну розробку декількох функціональних модулів та підвищує ймовірність потрапляння помилок до production-середовища. Крім того, неструктурований підхід до управління змінами ускладнює процес тестування та локалізації помилок, оскільки різні зміни інтегруються одночасно без чіткого розділення функціональності. Це призводить до збільшення часу на виявлення причин помилок та підвищує складність супроводу програмного забезпечення.

У межах реалізованого проєкту було використано структурований підхід до організації розробки, який базується на модифікованій моделі GitFlow. Використання feature-гілок дозволило ізолювати процес реалізації окремих

компонентів системи та забезпечити незалежну розробку різних функціональних модулів.

Такий підхід дозволив організувати паралельну розробку декількох частин системи без впливу на стабільність основної кодової бази. Наприклад, реалізація механізмів авторизації користувачів могла виконуватись одночасно із розробкою модулів роботи з музичним контентом без ризику порушення працездатності production-версії застосунку.

У контексті музичних платформ це має особливе значення, оскільки різні модулі системи можуть відповідати за незалежні процеси: роботу з аудіофайлами, обробку метаданих композицій, систему коментарів, авторизацію користувачів або механізми публікації контенту. Ізоляція змін дозволяє розробляти та тестувати такі компоненти незалежно один від одного без ризику порушення стабільності всієї системи.

Для керівника проєкту використання GitFlow є інструментом декомпозиції та паралельного планування. Це дає змогу чітко розмежовувати відповідальність у команді: поки один розробник працює над критичним виправленням у hotfix-гілці, інша частина команди продовжує роботу над запланованим спринтом у feature-гілках, не зупиняючи процес підготовки релізу.

Використання Pull Request та механізмів Code Review забезпечило додатковий рівень контролю якості програмного забезпечення. Перед інтеграцією змін до основних гілок проєкту програмний код проходив перевірку на відповідність архітектурним вимогам, стандартам програмування та правилам організації бізнес-логіки.

Практичне використання CI/CD-процесів також дозволило суттєво підвищити стабільність інтеграції змін. Усі Pull Request автоматично проходили перевірку за допомогою CI-пайплайна, який виконував:

- перевірку стилю програмування;
- запуск автоматизованих тестів;

- перевірку Docker-збірки;
- базовий аналіз безпеки залежностей.

Такий підхід дозволяє виявляти помилки ще до етапу інтеграції змін до основних гілок системи та значно зменшує ризик порушення стабільності застосунку після оновлення.

Важливою перевагою структурованого процесу розробки стала можливість підтримки чистоти релізів програмного забезпечення. Кожна стабільна версія системи формувалась як окремий перевірений Docker-образ, що проходив усі етапи автоматизованого тестування та збірки. Це дозволяє забезпечити контрольованість процесу розгортання та гарантує відтворюваність середовища виконання.

У процесі практичного дослідження також було встановлено, що використання контейнеризації спрощує інтеграцію змін між різними середовищами виконання системи. Завдяки Docker забезпечується однакова конфігурація середовищ розробки, тестування та production, що мінімізує кількість помилок, пов'язаних із несумісністю залежностей або відмінностями конфігурацій.

Окрему роль у підвищенні ефективності процесів розробки відіграла автоматизація розгортання системи. Використання CI/CD дозволило значно скоротити час між завершенням реалізації функціональності та її появою у production-середовищі. Автоматизоване створення Docker-образів і оновлення контейнерів забезпечило передбачуваність та стабільність процесу доставки програмного забезпечення.

Для оцінки ефективності впроваджених методів управління розробкою було використано підходи, що застосовуються у сучасних дослідженнях DevOps-практик, зокрема методологічні принципи DORA (DevOps Research and Assessment), представлені у роботах Н. Форсгрена, Дж. Хамбла та Дж. Кіма [22]. У дослідженні «Accelerate» автори класифікують ІТ-команди за рівнем зрілості процесів автоматизації та безперервної доставки програмного забезпечення. Зокрема, команди з низьким рівнем автоматизації (Low Performers)

характеризуються значними часовими витратами на інтеграцію змін та ручне розгортання систем, тоді як високопродуктивні команди (High та Elite Performers), що використовують CI/CD та контейнеризацію, демонструють суттєве скорочення часу доставки програмного забезпечення та підвищення стабільності релізів. У межах даного дослідження методологічний підхід DORA було використано для оцінки ефективності процесів розробки системи «MicroMusicBlog». Отримані результати порівнювались із типовими характеристиками ручного та автоматизованого підходів до управління розробкою ПЗ [23].

Показники традиційного підходу, наведені у таблиці 4.1, були отримані шляхом хронометражу ручного виконання основних операцій під час розробки та розгортання системи. Показники автоматизованого підходу визначались на основі статистики виконання GitHub Actions, логів Docker-контейнерів та часу виконання CI/CD-пайплайнів.

Отримані результати демонструють, що найбільший ефект від впровадження автоматизованих підходів спостерігається у скороченні часу розгортання та спрощенні процесу підготовки середовища розробки. Завдяки контейнеризації та автоматизації CI/CD більшість рутинних операцій була виключена з процесу ручного управління. Це узгоджується із тенденціями, описаними у звітах State of DevOps Report, де автоматизація процесів інтеграції та доставки програмного забезпечення розглядається як один із ключових факторів підвищення продуктивності ІТ-команд [24]. Зокрема, скорочення часу розгортання та підвищення стабільності релізів, зафіксовані у межах практичного дослідження, відповідають загальним закономірностям переходу від ручних процесів до автоматизованих DevOps-практик.

Таблиця 4.1 – Порівняльний аналіз показників ефективності процесів розробки

Метрика	Традиційний підхід	Автоматизований підхід	Ефект
Час розгортання системи	40–60 хв	3–5 хв	≈ 90% скорочення
Час налаштування середовища	4–8 год	10–15 хв	≈ 95% скорочення
Виявлення помилок до релізу	Переважно ручне тестування	Автоматизоване CI-тестування	Підвищення ефективності
Ймовірність конфліктів при злитті	90%	20%	≈ 80% зменшення
Стабільність релізів	Нестабільна	Контрольована	Підвищення надійності
Продуктивність системи	Невизначена	5000+ RPS	Стабільна робота
Пропускна здатність команди	Обмежена конфліктами	Висока (паралельна розробка)	Зростання продуктивності

Для додаткової оцінки ефективності було проведено розрахунок загального коефіцієнта підвищення ефективності процесів розробки. Основою для розрахунку виступає порівняння часових витрат на один цикл інтеграції змін – від моменту завершення розробки функціональності до її розгортання у середовищі виконання.

Сумарний час виконання циклу при традиційному підході визначається за формулою:

$$T_m = t_{config} + t_{test} + t_{build} + t_{deploy}$$

де t_{config} – час налаштування середовища;

t_{test} — час тестування;

t_{build} — час збірки застосунку;

t_{deploy} — час розгортання системи.

Для традиційного підходу було отримано такі значення:

– налаштування середовища – близько 30 хв;

– ручне тестування – близько 40 хв;

– ручне розгортання – близько 20 хв.

Таким чином:

$$T_m = 90 \text{ хв.}$$

У випадку використання автоматизованого підходу із Docker та CI/CD часові витрати значно скорочуються за рахунок автоматизації основних процесів:

– автоматизоване створення середовища Docker – близько 2 хв;

– автоматичне тестування та розгортання через CI/CD – близько 5 хв.

Відповідно:

$$T_a = 7 \text{ хв.}$$

Для оцінки загального приросту ефективності використовується формула:

$$E_{total} = \left(\frac{T_m - T_a}{T_m} \right) \times 100\%$$

Після підстановки отриманих значень:

$$E_{total} = \left(\frac{90 - 7}{90} \right) \times 100\% \approx 92,2\%$$

Отриманий результат свідчить про те, що впровадження автоматизованих методів управління розробкою на базі GitFlow, Docker та CI/CD дозволило підвищити ефективність операційних процесів приблизно на 92%. Для IT-проектів у музичній сфері це має критичне значення, оскільки подібні системи часто потребують безперервного оновлення функціональності, швидкого виправлення помилок та підтримки стабільного доступу користувачів до мультимедійного контенту.

4.3 Оцінка впливу GitFlow, Docker та CI/CD на стабільність і надійність системи

У процесі практичного дослідження було проведено оцінку впливу впроваджених методів управління розробкою на стабільність, надійність та безпечність програмної системи «MicroMusicBlog». Основна увага приділялась аналізу ефективності використання моделі GitFlow, контейнеризації Docker та автоматизованих CI/CD-процесів у контексті забезпечення контрольованого життєвого циклу програмного забезпечення. Для систем створення та публікації музичних композицій питання стабільності є особливо важливим через необхідність забезпечення безперервного доступу до аудіоконтенту, коректної роботи механізмів завантаження файлів та стабільної взаємодії між сервісами обробки мультимедійних даних.

Для оцінки результатів дослідження було використано підходи та принципи, описані у праці Б. Бейєра, К. Джонса, Дж. Петоффа та Н. Мерфі «Site Reliability Engineering: How Google Runs Production Systems», а також рекомендації Google Cloud Architecture Framework: Reliability [25-26].

У зазначених джерелах надійність програмної системи визначається як здатність інфраструктури підтримувати стабільну та передбачувану роботу в умовах постійних змін програмного коду, оновлення залежностей та зростання навантаження. Одними з основних факторів забезпечення стабільності сучасних ІТ-систем визначаються:

- автоматизація процесів інтеграції та розгортання;
- стандартизація середовищ виконання;
- контроль змін через системи керування версіями;
- мінімізація ручних операцій;
- швидке відновлення системи після збоїв.

Згідно з підходами Site Reliability Engineering, ручне управління процесами інтеграції та розгортання програмного забезпечення створює значну кількість операційних ризиків, серед яких:

- нестабільність production-середовища;
- помилки конфігурації;
- залежність від людського фактору;
- складність відновлення системи після збоїв;
- відсутність відтворюваності інфраструктури;
- висока ймовірність критичних помилок після оновлення системи.

У межах практичної частини дослідження було проведено порівняння традиційного підходу до розробки програмного забезпечення та автоматизованої моделі, реалізованої у системі «MicroMusicBlog» на базі GitFlow, Docker та CI/CD.

Однією з ключових переваг використання GitFlow стало підвищення стабільності кодової бази. Завдяки чіткому розділенню гілок за функціональним призначенням було забезпечено ізоляцію процесів розробки нових функцій, виправлення помилок та підготовки релізів. Це дозволило мінімізувати ризик інтеграції нестабільного коду до production-середовища та спростити управління змінами у межах проєкту.

Використання feature-гілок забезпечило можливість незалежної реалізації окремих компонентів системи без порушення працездатності основної версії застосунку. У свою чергу, release- та hotfix-гілки дозволили реалізувати контрольований процес підготовки релізів та оперативного виправлення критичних помилок.

Важливим фактором підвищення надійності системи стало впровадження механізму Pull Request та обов'язкового Code Review. Кожна зміна програмного коду проходила перевірку перед інтеграцією до основних гілок проєкту, що дозволило:

- зменшити кількість архітектурних помилок;
- забезпечити дотримання стандартів програмування;
- підвищити якість реалізації бізнес-логіки;
- мінімізувати ризик потрапляння нестабільного коду до production-середовища.

Застосування CI/CD-процесів підвищило контроль над інтеграцією та розгортанням системи. Автоматичний запуск тестів при створенні Pull Request забезпечив раннє виявлення помилок до етапу інтеграції змін.

Автоматизація збірки та доставки програмного забезпечення зменшила вплив людського фактору та забезпечила відтворюваність середовища завдяки єдиному процесу формування Docker-образів для всіх етапів життєвого циклу системи.

Використання Code Review та CI-перевірок дозволило перенести контроль якості на ранні етапи розробки, що зменшило кількість критичних помилок перед релізом і підвищило передбачуваність процесу розгортання.

Контейнеризація на базі Docker забезпечила ізоляцію сервісів та уніфікацію середовищ виконання, що мінімізувало проблеми сумісності та різниці конфігурацій між розробкою, тестуванням і production.

Завдяки використанню однакових Docker-образів у всіх середовищах знизився ризик помилок після деплою. Для порівняльного аналізу стабільності

системи використано таблицю 4.2.

Таблиця 4.2 – Порівняльний аналіз показників ефективності процесів розробки

Показник	Традиційний підхід	GitFlow + Docker + CI/CD
Відтворюваність середовища	Низька	Висока
Ризик помилок конфігурації	Високий	Мінімізований
Контроль інтеграції змін	Частковий	Автоматизований
Швидкість rollback	Низька	Висока
Стабільність релізів	Нестабільна	Контрольована
Імовірність критичних помилок після деплою	Висока	Значно знижена
Залежність від ручних операцій	Висока	Мінімальна
Контроль версій інфраструктури	Відсутній	Повний

Отримані результати демонструють, що використання автоматизованих DevOps-практик значно підвищує стабільність та передбачуваність роботи програмної системи. Особливо важливим фактором є мінімізація ручних операцій, які у традиційному підході виступають одним із головних джерел помилок та нестабільності production-середовища.

Використання GitFlow у поєднанні з контейнеризацією Docker забезпечило можливість реалізації контрольованого rollback-механізму, що відповідає рекомендаціям Site Reliability Engineering щодо мінімізації часу відновлення системи після збоїв Mean Time To Recovery (MTTR).

У традиційному підході відновлення системи після помилки часто потребує ручного втручання та повторної конфігурації середовища. У реалізованій системі «MicroMusicBlog» повернення до попередньої стабільної версії здійснюється шляхом використання Git-тегів та версіонованих Docker-образів, що дозволяє

значно скоротити час відновлення працездатності системи.

У процесі практичного тестування було встановлено, що використання автоматизованого підходу дозволяє скоротити середній час відновлення системи після помилки з 20–30 хвилин при ручному розгортанні до 2–5 хвилин при використанні Docker та CI/CD. Це відповідає підходам reliability engineering, де швидкість відновлення системи є одним із ключових показників стабільності інфраструктури.

Використання Docker-контейнеризації також суттєво покращує управління ресурсами команди. Час введення нового розробника в проєкт скорочується з декількох робочих днів до 15 хвилин, оскільки налаштування робочого оточення повністю автоматизоване. Це мінімізує витрати проєкту на адаптацію персоналу.

Важливою перевагою інтеграції GitFlow із CI/CD стала автоматизація процесів контролю життєвого циклу програмного забезпечення. Зокрема:

- внесення змін до develop-гілки автоматично ініціює запуск тестування та збірки staging-версії системи;
- інтеграція змін до main-гілки запускає процес формування production-релізу та автоматизованого розгортання;
- створення тегів релізів дозволяє фіксувати стабільні версії програмного забезпечення для подальшого використання або відновлення.

З точки зору безпеки проєкту впроваджені підходи також забезпечили додатковий рівень захисту:

- контроль змін через Pull Request мінімізує ризик внесення небезпечного коду;
- автоматизована перевірка залежностей дозволяє виявляти потенційно вразливі компоненти;
- ізоляція контейнерів зменшує ризик поширення помилок між сервісами;
- централізоване управління конфігураціями через змінні середовища підвищує безпечність роботи із секретними даними.

Отримані результати узгоджуються з принципами Site Reliability Engineering та рекомендаціями Google Cloud Architecture Framework щодо побудови надійних production-систем. Проведене дослідження підтверджує, що комплексне використання GitFlow, Docker та CI/CD дозволяє значно підвищити стабільність, надійність та керованість програмної системи шляхом автоматизації процесів інтеграції, стандартизації середовищ виконання та мінімізації ризиків, пов'язаних із людським фактором.

4.4 Рекомендації щодо застосування запропонованого підходу та можливості подальшого розвитку системи

Результати проведеного дослідження підтвердили ефективність комплексного використання GitFlow, Docker та CI/CD для управління процесами розробки сучасних ІТ-проектів. Запропонований підхід дозволяє забезпечити контрольованість життєвого циклу програмного забезпечення, підвищити стабільність системи та мінімізувати ризики, пов'язані з ручним управлінням інфраструктурою.

Отримані результати свідчать про доцільність використання впроваджених методів не лише у межах системи «MicroMusicBlog», але й під час розробки інших веборієнтованих платформ із високою інтенсивністю змін та потребою у швидкому випуску нових версій програмного забезпечення [27].

Однією з головних переваг запропонованого підходу є можливість масштабування процесів розробки при збільшенні кількості учасників команди. У невеликих проєктах відсутність структурованого управління версіями може не створювати критичних проблем, однак зі збільшенням кількості розробників значно підвищується ризик конфліктів при інтеграції змін, дублювання функціоналу та

порушення стабільності кодової бази.

Використання моделі GitFlow дозволяє уникнути так званого «merge hell» – ситуації, за якої велика кількість паралельних змін призводить до складних конфліктів злиття та втрати контролю над станом проєкту. Завдяки чіткому розподілу гілок за функціональним призначенням забезпечується ізоляція процесів розробки та передбачуваність інтеграції нового функціоналу. А також сприяє формуванню DevOps-культури всередині команди, де відповідальність за якість продукту розподіляється між розробниками та операційними фахівцями. Це спрощує комунікацію та усуває організаційні бар'єри, що є типовою проблемою при управлінні великими ІТ-проєктами

Практика використання Pull Request та Code Review також має важливе значення для командної розробки. Як зазначається у праці Accelerate, регулярна перевірка змін та автоматизація процесів інтеграції є одними з ключових факторів підвищення продуктивності та стабільності сучасних ІТ-команд. У межах системи «MicroMusicBlog» це дозволило забезпечити додатковий контроль якості програмного коду та зменшити ризик інтеграції нестабільних змін до production-середовища.

Важливим напрямом подальшого розвитку системи є вдосконалення процесів автоматизації інфраструктури. У межах проведеного дослідження Docker використовувався переважно як інструмент контейнеризації локального та production-середовища. Проте у масштабних distributed-системах доцільним є впровадження систем оркестрації контейнерів, таких як Kubernetes, що дозволяють автоматизувати масштабування сервісів, балансування навантаження та управління відмовостійкістю інфраструктури.

Крім того, перспективним напрямом розвитку є впровадження підходів Infrastructure as Code (IaC), які передбачають управління інфраструктурою через програмний код. Зокрема, використання таких інструментів, як Terraform або Ansible, дозволяє автоматизувати процес створення серверної інфраструктури та

забезпечити повну відтворюваність середовищ виконання.

Відповідно до рекомендацій, наведених у праці Site Reliability Engineering: How Google Runs Production Systems, подальше підвищення стабільності системи може бути досягнуте шляхом розширення засобів моніторингу та автоматичного реагування на збої. Для цього доцільним є використання централізованих систем логування та моніторингу, зокрема:

- Prometheus для збору метрик;
- Grafana для візуалізації показників;
- Elastic Search, LogStash, Kibana (ELK) Stack для централізованого аналізу логів.

Такі інструменти дозволяють реалізувати принципи observability, які є важливою складовою сучасних DevOps- та SRE-підходів.

Окрему увагу в подальших дослідженнях доцільно приділити питанням безпеки CI/CD-процесів. У сучасних умовах автоматизовані конвеєри розгортання стають однією з потенційних точок атаки, тому важливими напрямками розвитку є:

- автоматичне сканування залежностей на наявність вразливостей;
- використання секрет-менеджерів;
- впровадження Zero Trust-підходів;
- контроль безпеки Docker-образів;
- автоматизована перевірка політик доступу.

Перспективним напрямом також є інтеграція практик DevSecOps, які поєднують процеси розробки, експлуатації та забезпечення безпеки в єдиному автоматизованому середовищі [28].

Отже, результати дослідження підтверджують, що використання комплексного підходу до управління розробкою на базі GitFlow, Docker та CI/CD є ефективним методом удосконалення процесів розробки та управління ІТ-проєктами для створення сучасних програмних систем із високими вимогами до стабільності, масштабованості та швидкості доставки програмного забезпечення.

Запропонований підхід може бути використаний як основа для побудови масштабованих музичних платформ, систем потокової обробки аудіоконтенту, сервісів спільного створення музичних композицій та інших web-oriented систем із високими вимогами до стабільності, швидкості доставки функціональності та безперервної доступності сервісів [29-30].

ВИСНОВКИ

У результаті проведеного дослідження методів удосконалення процесів розробки та управління ІТ-проєктами для створення музичних композицій було вирішено ряд важливих завдань та отримано практичні результати, що підтверджують ефективність сучасних підходів до автоматизації процесів розробки програмного забезпечення.

У ході роботи було проведено аналіз сучасних методів організації процесів розробки програмних систем у музичній сфері. Досліджено особливості використання гнучких методологій управління проєктами, зокрема Scrum і Kanban, а також сучасних DevOps-практик.

За результатами дослідження встановлено, що для проєктів із високою інтенсивністю змін найбільш ефективним є інтегрований підхід, який поєднує принципи Agile та DevOps. Такий підхід дозволяє забезпечити гнучкість процесів розробки, підвищити стабільність програмної системи та мінімізувати ризики, пов'язані з ручним управлінням інфраструктурою.

У межах практичної частини роботи було спроектовано та реалізовано програмну систему «MicroMusicBlog» на базі FastAPI, PostgreSQL та Redis із використанням контейнеризації Docker та CI/CD-процесів. Для управління змінами було впроваджено модель контролю версій GitFlow, а для автоматизації процесів інтеграції та доставки програмного забезпечення – GitHub Actions.

Практичне тестування показало, що впровадження Docker та CI/CD дозволило суттєво скоротити час розгортання системи, зменшити кількість помилок конфігурації та підвищити стабільність релізів. Порівняльний аналіз традиційного та автоматизованого підходів продемонстрував скорочення

тривалості циклу інтеграції та розгортання приблизно на 92% у порівнянні з ручним підходом.

Окремо встановлено, що впровадження автоматизованих інструментів позитивно впливає не лише на технічну сторону розробки, а й на процеси управління ІТ-проєктом загалом. Автоматизація тестування, збірки та доставки програмного забезпечення дозволяє мінімізувати кількість рутинних операцій, зменшити потребу у ручному контролі та підвищити прозорість процесів розробки.

У роботі також було досліджено вплив автоматизації на стабільність і надійність програмної системи. Встановлено, що використання GitFlow, Docker та CI/CD дозволяє скоротити час відновлення системи після збоїв, підвищити контрольованість процесів інтеграції змін та забезпечити стабільність production-середовища.

За результатами дослідження визначено основні рекомендації щодо використання сучасних DevOps-практик у процесі розробки ІТ-проєктів, які включають використання систем контролю версій із чіткою моделлю розгалуження, контейнеризацію середовищ виконання, автоматизацію тестування та розгортання, а також впровадження засобів моніторингу та контролю стабільності системи.

Наукова новизна дослідження полягає у розробці комплексного підходу до організації процесів розробки та управління ІТ-проєктами у сфері створення музичних композицій, який поєднує методології Agile, DevOps-практики, контейнеризацію та автоматизовані CI/CD-процеси.

Практична цінність роботи визначається можливістю використання запропонованих підходів для підвищення ефективності процесів розробки сучасних веборієнтованих систем та зменшення операційних ризиків.

Таким чином, проведене дослідження підтвердило ефективність комплексного використання GitFlow, Docker та CI/CD як інструментів оптимізації процесів розробки та управління сучасними ІТ-проєктами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Романюк А. С. Дослідження методів удосконалення процесів розробки та управління ІТ-проєктами для створення музичних композицій. 30-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у ХХІ столітті» : зб. матеріалів форуму. Харків: ХНУРЕ, 2026. С. 221–222.
2. Актуальність веб-застосунку : вебсайт. URL: <https://istec.com.ua/uk/yak-viznachiti-aktualnist-veb-sajtu-internet-magazina/> (дата звернення: 06.04.2026).
3. FL Studio : офіційний вебсайт. URL: <https://www.image-line.com/> (дата звернення: 08.04.2026).
4. Soundcloud : офіційний вебсайт. URL: <https://soundcloud.com/> (дата звернення: 08.04.2026).
5. Suno AI : офіційний вебсайт. URL: <https://suno.com/> (дата звернення: 09.04.2026).
6. Організаційні компоненти інформаційних систем : вебсайт. URL: https://pidru4niki.com/10610928/informatika/komponenti_sistemi_opratsyuvannya_dani_h/ (дата звернення: 11.04.2026).
7. Redis Documentation : вебсайт. URL: <https://redis.io/documentation> (дата звернення: 15.04.2026).
8. FastAPI Documentation : вебсайт. URL: <https://fastapi.tiangolo.com/> (дата звернення: 17.04.2026).
9. Docker Documentation : вебсайт. URL: <https://docs.docker.com/> (дата звернення: 19.04.2026).
10. A successful Git branching model : вебсайт. URL: <https://nvie.com/posts/a-successful-git-branching-model/> (дата звернення: 25.04.2026).
11. Project Management Institute. A Guide to the Project Management Body of Knowledge (PMBOK Guide). 7th Edition. Newtown Square, Pennsylvania. Project

Management Institute, Inc., 2021. P. 370–374.

12. Skarin H. Scaling Agile with Platform Engineering: Managing complexity in large-scale IT projects. Journal of Software Engineering and Management. 2023. Vol. 14, No. 2. P. 45–58.

13. Chacon S., Straub J. Pro Git. 2nd Edition. New York. Apress, 2014. P. 540–544.

14. What is a Git workflow : вебсайт. URL: <https://about.gitlab.com/topics/version-control/what-is-git-workflow/> (дата звернення: 28.04.2026).

15. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston. Addison-Wesley Professional, 2010. P. 510–514.

16. GitHub Actions Documentation : вебсайт. URL: <https://docs.github.com/en/actions> (дата звернення: 29.04.2026).

17. What is a container : вебсайт. URL: <https://www.docker.com/resources/what-container/> (дата звернення: 01.05.2026).

18. Antao T. Professional FastAPI: A recipe-based guide to building high-performance REST APIs with FastAPI and Python 3.11. Birmingham. Packt Publishing, 2022. P. 340–349.

19. PostgreSQL Documentation : вебсайт. URL: <https://www.postgresql.org/docs/> (дата звернення: 03.05.2026).

20. Alembic Documentation: Database Migrations for SQLAlchemy : вебсайт. URL: <https://alembic.sqlalchemy.org/en/latest/> (дата звернення: 05.05.2026).

21. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. Sebastopol, CA. O'Reilly Media, 2022. P. 445–453.

22. Forsgren N., Humble J., Kim G. Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. Portland, Oregon. IT Revolution Press, 2018. P. 280–290.

23. DORA. 2024 State of DevOps Report: Focusing on User-Centricity and Platform Engineering. URL: <https://cloud.google.com/devops/state-of-devops/> (дата звернення: 09.05.2026).

24. DORA. 2023 State of DevOps Report. URL: <https://cloud.google.com/devops/state-of-devops/> (дата звернення: 10.05.2026).

25. Beyer B., Jones C., Petoff J., Murphy N. Site Reliability Engineering: How Google Runs Production Systems. Sebastopol, CA. O'Reilly Media, 2016. P. 550–554.

26. Google Cloud Architecture Framework: Reliability : вебсайт. URL: <https://cloud.google.com/architecture/framework/reliability> (дата звернення: 25.04.2026).

27. Managing Digital Projects in the Music Industry: Methodologies and Tools : вебсайт. URL: <https://www.berklee.edu/focused/music-business/digital-project-management> (дата звернення: 12.05.2026).

28. DevSecOps: Integrating Security into the CI/CD Pipeline : вебсайт. URL: <https://www.redhat.com/en/topics/devops/what-is-devsecops> (дата звернення: 13.05.2026).

29. OWASP Top 10:2021 The Next Generation of Application Security : вебсайт. URL: <https://owasp.org/www-project-top-ten/> (дата звернення: 14.05.2026).

30. State of Software Supply Chain 2025: Security and Dependency Management Report. Sonatype, 2025. P. 75–89.