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

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

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Дослідження процесів оптимізації розробки full-stack застосунків для підвищення характеристик ефективності та масштабованості

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

_____ Болож О.В.
(підпис) (прізвище та ініціали)

Керівник _____ Боднарчук І.О.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Готович В.А.
(підпис) (прізвище та ініціали)

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

Рецензент _____ Кульчицький Т.Р.
(підпис) (прізвище та ініціали)

Тернопіль
2024

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

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

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«____»_____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

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

Студенту Боложу Олександру Васильовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження процесів оптимізації розробки full-stack застосунків для підвищення характеристик ефективності та масштабованості

Керівник роботи Боднарчук Ігор Орестович, к.т.н., доцент кафедри КН
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від « 24 » листопада 2023 року № 4/7-1100

2. Термін подання студентом завершеної роботи 29 травня 2024р.

3. Вихідні дані до роботи Наукові публікації про методи оптимізації розробки з метою підвищення ефективності та масштабованості, технічна документація, інтернет джерела

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1 Аналітичний огляд існуючих концепцій та методологій оптимізації розробки full-stack застосунків. 2 Дослідження моделі безсерверних обчислень, концепції CI/CD конвеєрів та методології тестування продуктивності. 3 Впровадження процесів оптимізації на прикладі музичної платформи. 4 Охорона праці та безпека в надзвичайних ситуаціях. Висновки. Додатки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1 Тема роботи. 2 Мета дослідження. 3 Завдання дослідження. 4 Актуальність дослідження. 4 Актуальність дослідження. 5 Full-Stack розробка. 6. Ефективність full-stack застосунків. 7 Масштабованість full-stack застосунків. 8 Методи оптимізації розробки для підвищення ефективності та масштабованості full-stack застосунків. 9 Безсерверні обчислення. 10 Тестування продуктивності. 11 Конвеєри CI/CD. 12 Вимоги до експериментальної системи 13 Архітектура системи з впровадженням процесів оптимізації. 14 Сценарій навантажувального тестування. 15 Результати тестування з застосуванням першого робочого навантаження. 16 Результати тестування з застосуванням другого робочого навантаження. 17 Результати тестування з застосуванням третього робочого навантаження. 18 Результати впровадження CI/CD конвеєру. 19 Висновки. 20 Завершальний слайд.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Сенчишин В.С., доцент		
Безпека в надзвичайних ситуаціях	Клепчик В.М., ст. викладач		

7. Дата видачі завдання 24 листопада 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	04.12.2023	*Виконано*
2.	Підбір наукових джерел про процеси оптимізації	15.12.2023-31.11.2023	*Виконано*
	розробки full-stack застосунків з метою підвищення		
	характеристик ефективності та масштабованості		
3.	Опрацювання наукових публікацій та збір даних по	15.01.2024-25.02.2024	*Виконано*
	темі роботи		
4.	Виконання дослідження згідно мети кваліфікаційної	26.02.2024-07.04.2024	*Виконано*
	роботи		
5.	Оформлення розділу «Аналітичний огляд існуючих	15.04.2024-18.04.2024	*Виконано*
	концепцій та методологій оптимізації розробки»		
6.	Оформлення розділу «Дослідження моделі	19.04.2024-25.04.2024	*Виконано*
	безсерверних обчислень, концепції CI/CD конвеєрів та		
	методології тестування продуктивності»		
7.	Оформлення розділу «Впровадження процесів	26.04.2024-02.05.2024	*Виконано*
	оптимізації на прикладі музичної платформи»		
8.	Виконання завдання до підрозділу «Охорона праці»	03.05.2024-07.05.2024	*Виконано*
9.	Виконання завдання до підрозділу «Безпека в	08.05.2024-10.05.2024	*Виконано*
	надзвичайних ситуаціях»		
10.	Оформлення кваліфікаційної роботи	11.05.2024-14.05.2024	*Виконано*
11.	Нормоконтроль	15.05.2024-16.05.2024	*Виконано*
12.	Перевірка на плагіат		*Виконано*
13.	Попередній захист кваліфікаційної роботи		*Виконано*
14.	Захист кваліфікаційної роботи	29.05.2024	

Студент _____ Болож О.В.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Боднарчук І.О.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дослідження процесів оптимізації розробки full-stack застосунків для підвищення характеристик ефективності та масштабованості // Кваліфікаційна робота освітнього рівня «Магістр» // Болож Олександр Васильович// Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СНнм-61 // Тернопіль, 2024 // С. 75, рис. – 25, табл. – 3, кресл. – 20, додат. – 2, бібліогр. – 48.

Ключові слова: full-stack розробка, безсерверні обчислення, CI/CD, кешування, ефективність, масштабованість

Кваліфікаційна робота присвячена дослідженню процесів оптимізації розробки full-stack застосунків для підвищення характеристик ефективності та масштабованості та їх впровадженню.

У першому розділі кваліфікаційної роботи розглядається концепція full-stack розробки, аналізуються ключові аспекти ефективності та масштабованості full-stack застосунків, а також досліджуються сучасні підходи до оптимізації розробки задля підвищення зазначених характеристик.

Другий розділ присвячений дослідженню безсерверної моделі обчислень як одного з методів оптимізації. Також розглядається методологія навантажувального тестування як метод оцінки ефективності та масштабованості full-stack застосунків та описується концепція конвеєрів безперервної інтеграції та безперервного розгортання (CI/CD).

У третьому розділі здійснюється впровадження методів оптимізації, зокрема безсерверної моделі обчислень, кешування та CI/CD, проводиться навантажувальне тестування та здійснюється оцінка ефективності та масштабованості на основі результатів.

В четвертому розділі кваліфікаційної роботи розглядаються питання охорони праці та безпеки в надзвичайних ситуаціях, зокрема фізіологічні та психологічні аспекти праці, вимоги до режимів праці та відпочинку під час

роботи з комп'ютерними системами та підвищення стійкості функціонування підприємств приладобудівної галузі в умовах воєнного часу.

ANNOTATION

Research on full-stack development optimization processes to increase efficiency and scalability characteristics // The educational level "Master" qualification work // Bolozh Oleksandr // Ternopil Ivan Pulyuy National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Computer Science, SNnm-61 group // Ternopil, 2024 // P. 75, fig. - 25, tables - 3, posters - 20, annexes - 2, ref. - 48.

Key words: full-stack development, serverless computing, CI/CD, caching, efficiency, scalability

The qualification work is devoted to the study of full-stack development optimization processes to increase efficiency and scalability characteristics and their implementation.

The first chapter of the qualification work discusses the concept of full-stack development, analyzes the key aspects of efficiency and scalability of full-stack applications, and explores modern approaches to optimizing development to improve these characteristics.

The second section is devoted to the study of the serverless computing model as one of the optimization methods. It also considers the load testing methodology as a method for evaluating the efficiency and scalability of full-stack applications and describes the concept of continuous integration and continuous deployment (CI/CD) pipelines.

The third section introduces optimization methods, including serverless computing, caching, and CI/CD, conducts load testing, and evaluates the efficiency and scalability based on the results.

The fourth section focuses on occupational health and safety in emergencies, including the physiological and psychological aspects of work, requirements for work and rest regimes when working with computer systems, and improving the resilience of the functioning of instrument-making enterprises in wartime.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД (База даних) – це організована колекція даних, які зберігаються і обробляються за допомогою системи керування базами даних (СКБД).

Безсерверні обчислення (Serverless computing) – парадигма хмарних обчислень, де розробники можуть створювати і запускати застосунки без необхідності керування серверами. Обчислювальні ресурси динамічно розподіляються за потребою, а плата стягується лише за використані ресурси.

API (Application Programming Interface) – набір правил, протоколів та інструментів для створення програмного забезпечення для спілкування між різними компонентами. API визначає, як різні програмні компоненти повинні взаємодіяти один з одним.

AWS (Amazon Web Services) – хмарна платформа, що пропонує широкий спектр хмарних послуг, включаючи обчислювальні потужності, сховища даних, мережеві ресурси та багато іншого.

BaaS (Backend-as-a-Service) – модель хмарних сервісів, де постачальник хмарних послуг керує всією серверною інфраструктурою та забезпечує функції, необхідні для роботи мобільних та веб-додатків, такі як управління даними, автентифікація користувачів, push-сповіщення тощо.

CD (Continuous Delivery) – практика методології DevOps, яка передбачає автоматизоване створення, тестування та підготовку коду до випуску в будь-який момент часу з мінімальними зусиллями.

CDN (Content Delivery Network) – розподілена мережа серверів, що забезпечує ефективну доставку веб-контенту користувачам на основі їх географічного розташування, зменшуючи затримку та збільшуючи пропускну здатність.

CI (Continuous Integration) – практика методології DevOps, яка передбачає часте об'єднання коду в єдину основну гілку репозиторію з автоматизованими збірками та тестами.

CRUD – акронім, що позначає чотири основні операції з базами даних: створення (Create), читання (Read), оновлення (Update) та видалення (Delete) записів.

FaaS (Function-as-a-Service) – модель хмарних обчислень, де розробники можуть створювати та розгортати окремі функції без необхідності управління серверами. Ресурси динамічно виділяються за потребою, а обчислювальна потужність оплачується відповідно до використання.

ЗМІСТ

ВСТУП

Актуальність теми. Реалії сучасного цифрового світу висувають все вищі вимоги до ефективності та масштабованості веб-, чи мобільних застосунків. Користувачі очікують миттєвого відгуку системи, безперебійної роботи навіть за умов пікових навантажень, а компанії прагнуть максимально оптимізувати витрати на інфраструктуру. У цьому контексті питання оптимізації виходить на перший план, особливо в галузі full-stack розробки, коли необхідно забезпечити ефективність та масштабованість усіх аспектів розроблюваного застосунку.

Мета і задачі дослідження. Метою даної кваліфікаційної роботи освітнього рівня «Магістр» є дослідження процесів оптимізації розробки full-stack застосунків з метою підвищення характеристик ефективності та масштабованості. Для досягнення поставленої мети потрібно виконати ряд завдань, зокрема:

- Провести ґрунтовний аналіз характеристик ефективності та масштабованості full-stack застосунків для ідентифікації аспектів для оптимізації.
- Дослідити сучасні методології та підходи до оптимізації процесів розробки, спрямовані на підвищення характеристик ефективності та масштабованості.
- Впровадити досліджені методи оптимізації та надати об'єктивну оцінку їх впливу на показники ефективності та масштабованості.

Об’єкт дослідження: характеристики ефективності та масштабованості full-stack застосунків.

Предмет дослідження: процеси оптимізації розробки full-stack застосунків для підвищення характеристик ефективності та масштабованості.

Наукова новизна одержаних результатів кваліфікаційної роботи полягає у виявленні переваг та обмежень сучасних методів оптимізації розробки full-stack застосунків, серед яких методології безсерверних обчислень, безперервної інтеграції та доставки (CI/CD). Отримані результати можуть бути використані для подальших досліджень в цьому напрямку.

Практичне значення одержаних результатів. Проведено імплементацію досліджуваних методів оптимізації у мобільний застосунок музичної платформи. З метою оцінки ефективності та масштабованості системи було здійснено низку навантажувальних тестів. Отримані результати дозволили провести всебічний аналіз впливу застосованих методів оптимізації в умовах різного рівня навантаження та виявити переваги та обмеження системи.

Апробація результатів магістерської роботи. Основні результати проведених досліджень обговорювались на VI міжнародній студентській науково-технічній конференції «Природничі та гуманітарні науки. Актуальні питання» Тернопільського національного технічного університету імені Івана Пулюя (м. Тернопіль, 2022 р.) та VII міжнародній студентській науково-технічній конференції «Природничі та гуманітарні науки. Актуальні питання» Тернопільського національного технічного університету імені Івана Пулюя (м. Тернопіль, 2024 р.)

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

Структура й обсяг кваліфікаційної роботи. Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку літератури з 48 найменувань та 2 додатків. Загальний обсяг кваліфікаційної роботи складає 75 сторінок, з них 50 сторінок основного тексту, який містить 25 рисунків та 3 таблиці.

1 АНАЛІТИЧНИЙ ОГЛЯД ІСНУЮЧИХ КОНЦЕПЦІЙ ТА МЕТОДОЛОГІЙ ОПТИМІЗАЦІЇ РОЗРОБКИ FULL-STACK ЗАСТОСУНКІВ

У першому розділі кваліфікаційної роботи досліджується концепція full-stack розробки, аналізуються ключові аспекти ефективності та масштабованості full-stack застосунків, а також розглядаються підходи до оптимізації розробки задля підвищення зазначених характеристик.

1.1 Огляд концепції full-stack розробки

Full-stack розробка – це процес створення повноцінних програмних застосунків, які поєднують клієнтську та серверну частини в єдиній кодовій базі [48, 22]. Цей підхід охоплює розробку як користувацького інтерфейсу (фронтенду), так і бекенду (бази даних та бізнес-логіки) (рисунок 1.1). Фронтенд відповідає за візуальне представлення інформації та взаємодію з користувачем, тоді як бекенд забезпечує основну функціональність застосунку, обробку даних та інтеграцію з іншими системами.

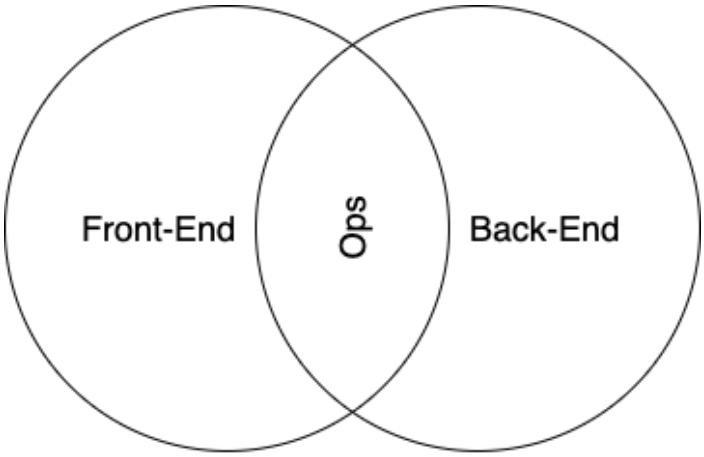

Рисунок 1.1 – Концепція Full-Stack розробки

Для розробки фронтенду зазвичай використовуються мови розмітки (HTML), стилізації (CSS) та JavaScript для створення динамічного та інтерактивного користувацького інтерфейсу. Бекенд реалізується за допомогою

серверних мов програмування, таких як Node.js, PHP, Ruby, Java або Python, і часто складається з трьох основних рівнів: рівня API (взаємодія з фронтендом), рівня зберігання даних (робота з базами даних) та рівня бізнес-логіки (обробка даних та реалізація основної функціональності).

Для прискорення та спрощення процесу розробки використовуються фреймворки – набори програмних компонентів та бібліотек, що забезпечують типові функції та дотримуються певних архітектурних принципів. Популярними фреймворками для full-stack розробки є Ruby on Rails, Django, Spring Boot, Laravel та Next.js.

Для ефективної роботи над проєктом застосунку часто використовуються програмні стеки – комбінації різних технологій, таких як мови програмування, бази даних, фреймворки, та API. Найпоширенішими стеками є LAMP (Linux, Apache, MySQL, PHP), MEAN (MongoDB, Express.js, Angular, Node.js) та LEMP (Linux, Nginx, MySQL, PHP).

До переваг даної концепції відносять [48, 22]:

- кращий контроль та розуміння проєкту;
- економія часу та коштів, краща продуктивність;
- швидше виправлення помилок завдяки цілісному розумінню системи.

Перейдемо до аналізу характеристик ефективності та масштабованості.

1.2 Аналіз ефективності та масштабованості full-stack застосунків

Ефективність full-stack застосунку є комплексною характеристикою, що визначає здатність виконувати необхідний функціонал з мінімальними витратами ресурсів і часу [20]. Вона залежить від багатьох чинників: вибору оптимальної архітектури, використання сучасних фреймворків і бібліотек, продуктивності команди розробників, якості коду, можливостей масштабування та розширення функціоналу.

До ключових кількісних показників ефективності відносять:

- швидкість роботи системи та окремих сервісів час виконання запитів, час завантаження;

- час відгуку на запити користувачів показує, наскільки швидко система реагує на дії користувача;
- пропускна здатність при високих навантаженнях кількість одночасних користувачів чи запитів, які система може обробити;
- вартість розробки та підтримки включає витрати на розробку, тестування, впровадження, а також поточні витрати на підтримку;
- собівартість внесення змін та додавання нового функціоналу відображає зусилля, необхідні для модифікації системи.

Серед якісних показників ефективності:

- зручність використання з точки зору кінцевих користувачів – інтуїтивний UI, простота навігації, адаптивний дизайн;
- можливості інтеграції з іншими системами – наявність API, сумісність форматів даних;
- задоволеність бізнес-користувачів – відповідність системи їх потребам та вимогам;
- відповідність сучасним стандартам і тенденціям розвитку галузі.

Ефективність full-stack рішення необхідно оцінювати на всіх рівнях: окремих компонентів і сервісів, підсистем та модулів, загальної архітектури та інфраструктури, бізнес-процесів та всієї системи в цілому. Комплексний підхід дозволяє виявити слабкі місця на кожному рівні та ефективно спрямувати зусилля на оптимізацію.

Масштабованість характеризує здатність системи стабільно функціонувати при значному зростанні навантаження – кількості користувачів, трафіку, обсягів даних [37]. Це надзвичайно важлива властивість сучасних високонавантажених застосунків.

Масштабованість досягається за рахунок:

- гнучкої архітектури, що дозволяє нарощувати потужності;
- використання хмарних технологій та сервісів автоматичного масштабування;
- механізмів кешування та балансування навантаження;
- асинхронної та паралельної обробки запитів.

Оцінити масштабованість можна за результатами навантажувального та стрес-тестування системи в умовах зростаючих навантажень. Аналізують такі показники:

- стабільність роботи сервісів;
- час відгуку на запити;
- відсоток помилок та відмов;
- використання ресурсів (процесор, пам’ять, мережа).

Належне тестування дозволяє виявити “прогалини” системи та розробити план дії для їх усунення.

1.3 Аналіз методів оптимізації розробки для підвищення ефективності та масштабованості

1.3.1 Мікросервісна архітектура

Мікросервісна архітектура – це архітектурний стиль, який використовується в розробці програмного забезпечення, де застосунок будується як набір невеликих, слабко пов’язаних між собою сервісів, які спілкуються між собою за допомогою легких механізмів, зазвичай HTTP (рисунок 1.2). Кожен сервіс призначений для виконання певної бізнес-функції і може бути розроблений, розгорнутий і масштабований незалежно [28].

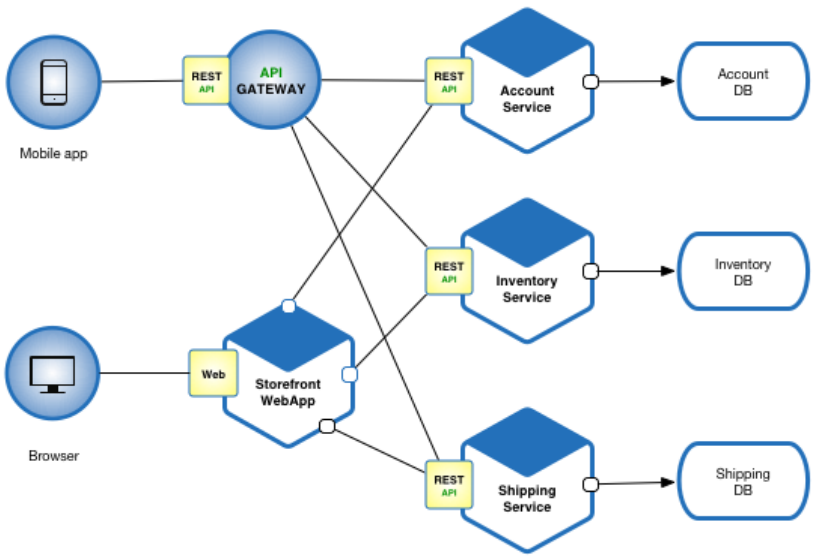

Рисунок 1.2 – Приклад мікросервісної архітектури e-commerce застосунку [28]

До ключових характеристик мікросервісної архітектури входять:

- Декомпозиція: застосунок розбивається на менші, керовані сервіси, кожен з яких відповідає за окремий аспект функціональності.
- Слабка зв’язність: сервіси не залежать один від одного і взаємодіють через чітко визначені API. Це дозволяє вносити зміни в один сервіс, не впливаючи на інші, що сприяє гнучкості та оперативності.
- Автономність: кожен сервіс можна розробляти, розгортати та масштабувати незалежно. Це дозволяє командам працювати над різними сервісами одночасно, використовуючи різні технології та мови програмування, за потреби.
- Масштабованість: оскільки сервіси є незалежними, їх можна масштабувати горизонтально, щоб впоратися зі зростаючим навантаженням, додаючи більше екземплярів певного сервісу.
- Стійкість: мікросервіси спроєктовані таким чином, щоб бути стійкими до збоїв. Якщо один сервіс виходить з ладу, це не призводить до падіння всієї системи.
- CD: мікросервіси сприяють безперервним практикам розробки та розгортання, оскільки кожен сервіс можна тестувати та розгортати незалежно.

Перейдемо до кешування.

1.3.2 Кешування

Кешування – це техніка, що використовується під час розробки програмного забезпечення для підвищення продуктивності та ефективності застосунків завдяки зберіганню даних, які часто використовуються, у пам’яті або на більш швидкому рівні зберігання.

Нижче наведено деякі найпоширеніші стратегії кешування:

- Мережі доставки контенту (CDN) – це мережа розподілених серверів, стратегічно розташованих у різних центрах обробки даних у всьому світі для доставки веб-контенту, включно з текстом, зображеннями, відео та іншими ресурсами, користувачам залежно від їхнього географічного розташування. Вони

підвищують продуктивність, масштабованість і доступність застосунків за рахунок зменшення затримки, розвантаження трафіку з вихідних серверів і розподілу контенту в різні місця. CDN особливо корисні для застосунків, керованих контентом. CDN зменшують затримку в мережі, вони забезпечують кращий користувацький досвід для застосунків, керованих контентом, особливо з великими медіа файлами. CDN надають масштабовану інфраструктуру і можуть розподіляти запити між різними прикордонними серверами. [15]

- Кешування в пам'яті (in-memory caching) передбачає зберігання часто використовуваних даних у пам'яті для швидшого їх отримання. Ці дані зазвичай кешуються в RAM (Random Access Memory), яка забезпечує швидший час відгуку порівняно з дисковим сховищем. Кешування в пам'яті часто використовується для кешування результатів запитів до баз даних, обчислених значень, даних сесій та інших тимчасових даних, які потребують швидкого доступу. Кешування даних у пам'яті дає змогу застосункам зменшити необхідність багаторазового звернення до повільних джерел даних, таких як бази даних, що забезпечує кращу продуктивність та масштабованість застосунку в цілому.

Перейдемо до асинхронної обробки.

1.3.3 Асинхронна обробка

Асинхронна обробка – це парадигма в розробці програмного забезпечення, де завдання виконуються незалежно від основного потоку програми (в фоновому режимі), що дозволяє застосунку продовжувати виконання, не чекаючи завершення цих завдань. [12] Ця концепція особливо корисна для обробки довготривалих або блокуючих операцій, таких як операції вводу/виводу, або завдань, що вимагають взаємодії із зовнішніми системами.

З погляду full-stack розробки дана концепція зазвичай реалізується за допомогою черг та брокерів повідомлень.

- Черги – це структури даних, які зазвичай працюють за принципом “Першим зайшов - першим вийшов” (FIFO – First-In-First-Out), де елементи

додаються в кінець і видаляються з початку. В контексті асинхронної обробки черги використовуються для відокремлення задач від основного потоку програми. Коли потрібно виконати задачу асинхронно, вона додається до черги, а не обробляється негайно. Це дозволяє застосунку продовжувати виконання, не чекаючи завершення задачі. Працівники або споживачі (workers/consumers) постійно відслідковують наявність задач в черзі та виконують їх, у разі появи.

- Брокери повідомлень – це системи проміжного програмного забезпечення, які полегшують зв'язок між різними частинами розподіленої системи шляхом маршрутизації та доставки повідомлень. У контексті асинхронної обробки, брокери повідомлень діють як посередники між виробниками (producers) (компонентами, які генерують повідомлення) і споживачами (consumers) (компонентами, які обробляють повідомлення). Коли виробник генерує повідомлення, він надсилає його брокеру повідомлень, який потім доставляє повідомлення одному або декільком споживачам на основі заздалегідь визначених правил або логіки маршрутизації.

Перейдемо до горизонтального масштабування.

1.3.4 Горизонтальне масштабування

Вертикальне масштабування, також відоме як масштабування вгору або масштабування по вертикалі, передбачає збільшення ресурсів (таких як процесор, оперативна пам'ять або сховище) одного сервера або екземпляра для обробки більших навантажень. Зазвичай це досягається шляхом модернізації апаратних компонентів сервера, наприклад, додаванням потужніших процесорів, збільшенням обсягу пам'яті або підключенням швидших пристроїв зберігання даних. Вертикальне масштабування є простим у реалізації, але має обмеження з точки зору масштабованості та економічної ефективності. Існує межа для вертикального масштабування одного сервера, і після певного моменту подальша модернізація апаратного забезпечення стає недоцільною або надто дорогою.

Горизонтальне масштабування, також відоме як масштабування по горизонталі, в свою чергу передбачає додавання нових екземплярів або вузлів до системи для розподілу робочого навантаження між декількома серверами. Замість того, щоб збільшувати потужності однієї й тієї самої машини, горизонтальне масштабування передбачає додавання більшої кількості серверів зі схожими або ідентичними характеристиками, щоб впоратися зі зростаючим попитом. Цей підхід за своєю суттю є більш масштабованим, ніж вертикальне масштабування, оскільки дозволяє додавати ресурси лінійно, що дає змогу системі обробляти більшу кількість користувачів або транзакцій, розподіляючи робоче навантаження між кількома серверами. Горизонтальне масштабування також забезпечує кращу відмовостійкість і доступність, оскільки кілька серверів можуть продовжувати працювати, навіть якщо один або кілька серверів вийдуть з ладу.

На рисунку 1.3 зображено відмінність між концепціями вертикального та горизонтального масштабувань.

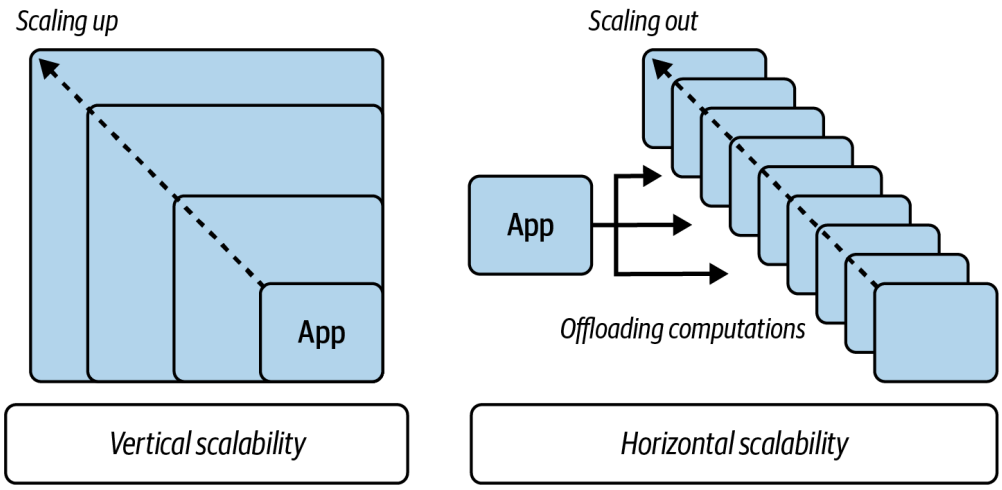

Рисунок 1.3 – Відмінності вертикального та горизонтального масштабування

Вертикальне масштабування має обмеження з точки зору масштабованості, оскільки існує максимальний поріг, до якого можна модернізувати один сервер. Горизонтальне масштабування, з іншого боку, пропонує практично безмежну масштабованість шляхом додавання нових серверів за потреби.

Вертикальне масштабування може бути дорогим, особливо при переході на висококласні апаратні компоненти. Горизонтальне масштабування, як правило, є більш економічно ефективним, оскільки воно передбачає додавання звичайного обладнання, а витрати лінійно залежать від кількості доданих серверів.

Вертикальне масштабування відносно простіше реалізувати, оскільки воно передбачає модернізацію одного сервера. Горизонтальне масштабування – складніше, оскільки вимагає управління кількома серверами, балансування навантаження та забезпечення узгодженості даних у розподілених системах.

1.3.5 Хмарні та безсерверні обчислення

Хмарні обчислення – це модель надання обчислювальних ресурсів (таких як сервери, сховища, бази даних, мережі та програмне забезпечення) через Інтернет за принципом “плати за користування” [47]. Замість того, щоб володіти та утримувати фізичну інфраструктуру, користувачі можуть отримати доступ до обчислювальних ресурсів, що надаються постачальниками хмарних послуг на вимогу.

За версією NIST (Національний інститут стандартів і технологій) хмарні обчислення поділяються на три основні моделі обслуговування [39]:

- Інфраструктура як послуга (IaaS): надає віртуалізовані обчислювальні ресурси, такі як віртуальні машини, сховища та мережева інфраструктура. Користувачі можуть розгортати свої програми та операційні системи на цих віртуалізованих ресурсах і керувати ними.
- Платформа як послуга (PaaS): пропонує високорівневі послуги та інструменти для розробки, розгортання та управління застосунками, не турбуючись про базову інфраструктуру. Постачальники PaaS керують інфраструктурою, середовищем виконання, проміжним програмним забезпеченням та інструментами розробки, дозволяючи розробникам зосередитися на створенні та розгортанні додатків.

- Програмне забезпечення як послуга (SaaS): надає програмні застосунки через Інтернет на основі підписки. Користувачі можуть отримати доступ до програмних додатків, розміщених у хмарних провайдерів, і використовувати їх без необхідності встановлювати або керувати ними локально.

Хмарні обчислення пропонують такі переваги, як масштабованість, гнучкість, економічність і надійність, що робить їх популярним вибором для розгортання широкого спектру застосунків і сервісів.

Безсерверні обчислення, що зазвичай розуміють як “Функція як послуга” (Function as a Service, FaaS) – це потенційно нова модель хмарних обчислень, в якій розробники можуть розгортати окремі функції або фрагменти коду без управління базовою інфраструктурою. У безсерверній архітектурі хмарний провайдер динамічно керує розподілом і наданням ресурсів, необхідних для виконання функцій у відповідь на події або тригери. У безсерверних обчисленнях розробники пишуть функції, які виконують певні завдання або реагують на події (наприклад, HTTP-запити, зміни в базі даних або завантаження файлів). Ці функції розгортаються на безсерверній платформі хмарного провайдера, де вони автоматично нарощують або ж зменшують потужності залежно від попиту. Користувачі платять лише за фактичний час виконання та ресурси, спожиті функціями, а не за надану інфраструктуру. Безсерверні обчислення пропонують такі переваги, як зменшення операційних накладних витрат, покращена масштабованість, економічна ефективність. Це дозволяє розробникам зосередитися на написанні коду та створенні застосунків, не турбуючись про управління серверами, масштабування інфраструктури або обробку низькорівневих деталей.

На рисунку 1.4 представлено порівняльну схему архітектурних відмінностей між згаданими моделями хмарних сервісів. Червоним кольором виділено компоненти, які потребують ручного конфігурування з боку користувача, тоді як зелений колір позначає ті складові, налаштування яких здійснюється провайдером хмарних послуг автоматично.

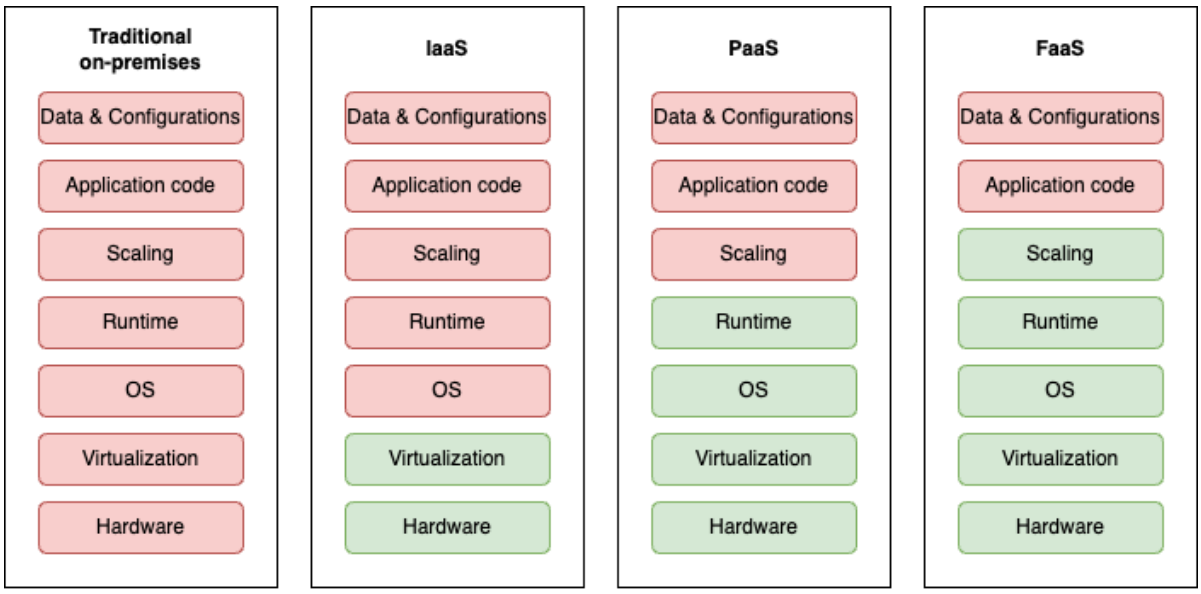

Рисунок 1.4 – Структурні відмінності моделей хмарних сервісів

В цілому, хмарні обчислення надають широкий спектр послуг і ресурсів для розгортання застосунків, тоді як безсерверні обчислення пропонують більший фокус на бізнес логіці, спрощуючи управління інфраструктурою. Обидві парадигми пропонують свої унікальні переваги та варіанти використання. При розробці зазвичай використовується комбінований підхід як традиційних так і безсерверних обчислень.

1.3.6 DevOps

DevOps – це культура та практика інженерії програмного забезпечення, яка має на меті об’єднати розробку програмного забезпечення (Dev) та його експлуатацію (Ops). DevOps націлений на скорочення циклів розробки, збільшення частоти розгортання, підвищення надійності випусків у тісному зв’язку з бізнес-цілями [18].

Ключовими принципами DevOps є:

- Співпраця: DevOps заохочує співпраці та командній роботі між відділами розробки програмного забезпечення, операцій та іншими зацікавленими сторонами, що беруть участь у процесі доставки програмного забезпечення.

- Автоматизація: є основним принципом DevOps, що дозволяє автоматизувати повторювані завдання, такі як збірка коду, тестування, розгортання та забезпечення інфраструктури. Автоматизуючи ручні процеси, DevOps зменшує кількість людських помилок, прискорює цикли доставки та підвищує загальну ефективність і узгодженість.

- Безперервна інтеграція – це практика розробки програмного забезпечення, коли розробники часто вносять зміни коду до спільного репозиторію, зазвичай по кілька разів на день. Автоматизовані процеси збірки та тестування запускаються після кожної фіксації коду, гарантуючи, що зміни будуть перевірені та інтегровані в основну кодову базу безперервно. CI допомагає виявити та виправити проблеми інтеграції на ранніх стадіях, що призводить до швидшого зворотного зв'язку та покращення якості коду.

- Безперервна доставка: безперервна доставка розширює принципи CI шляхом автоматизації процесу розгортання, дозволяючи командам швидко, надійно та з мінімальним ручним втручанням доставляти оновлення програмного забезпечення в робочі середовища. За допомогою CD кожна зміна коду, яка проходить автоматизовані тести, може бути розгорнута в продуктове середовище, дозволяючи організаціям швидко та безпечно випускати нові функції, виправлення та покращення для кінцевих користувачів.

- Інфраструктура як код (IaaC): це практика, в якій забезпечення та управління інфраструктурою ввідбувається за допомогою програмного коду та конфігураційних файлів. Дана концепція, забезпечує контроль версій, тестування та розгортання змін в інфраструктурі у повторюваний і послідовний спосіб, зменшуючи кількість помилок, що допускаються вручну, і забезпечуючи узгодженість в різних середовищах.

- Моніторинг та зворотній зв'язок: DevOps підкреслює важливість моніторингу продуктивності застосунків, стану інфраструктури та зворотного зв'язку з користувачами протягом усього життєвого циклу доставки програмного забезпечення. Збираючи та аналізуючи метрики, журнали та дані користувачів, команди можуть виявляти проблеми, вимірювати продуктивність та приймати

рішення на основі даних для постійного вдосконалення застосунків та інфраструктури.

DevOps є ключовою методологією, що використовується при розробці сучасних full-stack застосунків.

1.4 Висновок до першого розділу

У першому розділі кваліфікаційної роботи досліджено концепцію full-stack розробки програмних застосунків. Проаналізовано складові компоненти, переваги цього підходу, а також ключові аспекти ефективності та масштабованості full-stack застосунків.

Зокрема, було розглянуто поняття ефективності та її кількісні і якісні показники, такі як швидкість роботи, час відгуку, пропускна здатність, вартість розробки та підтримки, зручність використання. Визначено, що комплексний підхід до оцінки ефективності на всіх рівнях архітектури дозволяє виявити слабкі місця та ефективно спрямувати зусилля на оптимізацію.

Проаналізовано також поняття масштабованості, її важливість для сучасних високонавантажених застосунків та шляхи досягнення за рахунок гнучкої архітектури, використання хмарних технологій, механізмів кешування та балансування навантаження, асинхронної та паралельної обробки запитів.

Досліджено низку сучасних методів та підходів до оптимізації розробки full-stack застосунків для підвищення їх ефективності та масштабованості. Серед них: мікросервісна архітектура, стратегії кешування, асинхронна обробка, горизонтальне масштабування, хмарні та безсерверні обчислення, практики DevOps.

2 ДОСЛІДЖЕННЯ МОДЕЛІ БЕЗСЕРВЕРНИХ ОБЧИСЛЕНЬ, КОНЦЕПЦІЇ CI/CD КОНВЕЄРІВ ТА МЕТОДОЛОГІЇ ТЕСТУВАННЯ ПРОДУКТИВНОСТІ

У другому розділі кваліфікаційної розглядаються основні аспекти безсерверної моделі обчислень, зокрема концепції функції як послуги (Function-as-a-Service, FaaS) та бекенд як послуги (Backend-as-a-Service, BaaS). Окремо розглядається методологія тестування продуктивності як основна метрика для оцінки ефективності та масштабованості full-stack застосунків, а також популярні інструменти для виконання навантажувального тестування. Додатково описуються концепції безперервної інтеграції та безперервної доставки/розгортання (CI/CD), ключові показники ефективних CI/CD конвеєрів та здійснюється огляд відповідних платформ.

2.1 Безсерверні обчислення

2.1.1 Характеристика безсерверних обчислень

Безсерверні обчислення – це модель виконання хмарних обчислень, яка характеризується тим, що хмарний провайдер динамічно розподіляє машинні ресурси на основі попиту, таким чином керуючи серверами замість користувачів [38]. Незважаючи на термін "безсерверні", постачальники хмарних послуг все ще використовують сервери для розгортання застосунків. Зазвичай під терміном "безсерверні обчислення" розуміють те, що також відомо як "функція як послуга" (Function-as-a-Service, FaaS), коли окремі функції слугують одиницею розгортання замість цілих застосунків. Наразі різні хмарні провайдери пропонують FaaS на своїх платформах, зокрема Amazon з AWS Lambda, Google з Cloud Functions та Microsoft з Azure Functions.

У класифікації хмарних сервісів функція як послуга (FaaS) знаходиться між платформою як послугою (PaaS) та бекенд як послугою (BaaS) (рисунок 2.1). BaaS фактично є різновидом SaaS (програмне забезпечення як послуга).

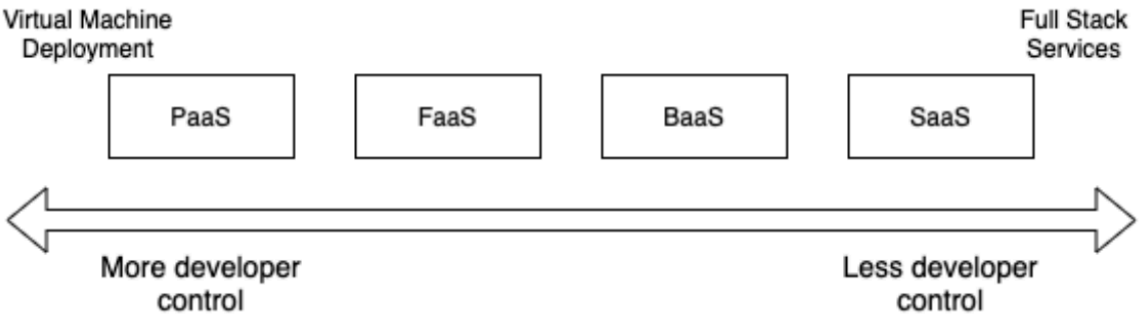

Рисунок 2.1 – Класифікація хмарних сервісів з точки зору контролю розробки

PaaS передбачає надання серверів і розгортання застосунків на віртуальних машинах у хмарі, пропонуючи розробникам більше контролю над інфраструктурою та розгорнутим кодом. З іншого боку, SaaS надає користувачам комплексні програмні послуги, при цьому постачальник послуг зберігає повний контроль над інфраструктурою та вихідним кодом, як, наприклад, у випадку з Gmail. FaaS займає проміжне місце між цими двома моделями.

Одна з істотних відмінностей між FaaS і PaaS полягає в масштабуванні та вартості. У той час як PaaS часто стягує плату в періоди простою, FaaS дозволяє функціям зменшувати потужності до нуля і динамічно масштабуватися при виклику, наприклад, у відповідь на зміни в базі даних або HTTP-запити. Функції FaaS розроблені як короткотривалі, їхні екземпляри автоматично створюються при надходженні події та знищуються після виконання та за відсутності надходження нової певний час, тим самим оптимізуючи ресурси сервера (рисунок 2.2).

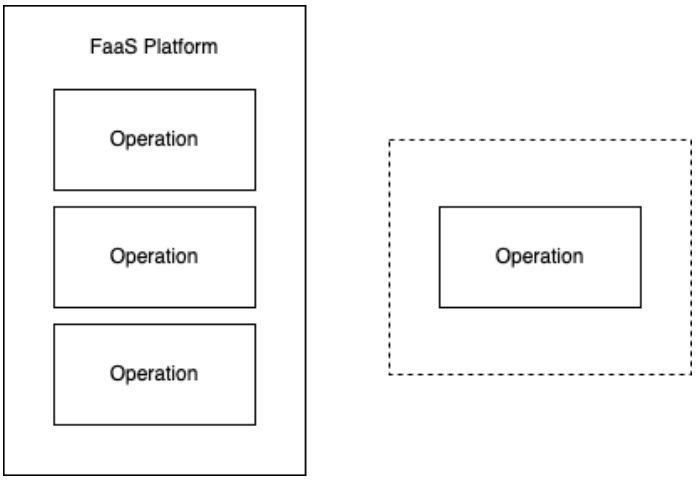

Рисунок 2.2 – Життєвий цикл FaaS [36]

Функції FaaS не зберігають стану за задумом, для полегшення масштабування, оскільки змінні, що зберігаються в пам'яті, можуть не зберігатися під час декількох викликів функції. Іншим хмарним сервісом, тісно пов'язаним з безсерверними обчисленнями, є Backend-as-a-Service (BaaS), який пропонує такі послуги, як зберігання даних та автентифікація від сторонніх постачальників, таких як Google Firebase. Як FaaS, так і BaaS усувають необхідність прямого управління сервером, підпадаючи під поняття безсерверних обчислень.

Ключовими характеристиками безсерверних функцій, на думку М. Робертса і Д. Чапіна, що визначаються в книзі “Що таке serverless?”, включають в себе:

- Позбавлення управління інфраструктурою або серверами; розгортання передбачає завантаження вихідного коду застосунку на функцію, що виконуватиметься.
- Автоматичне горизонтальне масштабування, кероване провайдером.
- Вартість на основі фактичного використання.
- Абстрагування користувача від конфігурації розміру хосту та кількості екземплярів.
- Висока доступність, що забезпечується провайдером, з перенаправленням запитів у разі збоїв у роботі базових компонентів.

По суті, безсерверна функція – це розміщена в хмарі, незалежно масштабована функція без стану, яка запускається і виконується у відповідь на зовнішні події.

2.1.2 Безсерверна архітектура

Подібно до мікросервісів, система з безсерверною архітектурою поділяється на невеликі компоненти. Однак, на відміну від “сервісів”, система з безсерверною архітектурою складається з численних невеликих, незалежних і самодостатніх функцій.

Як вже було згадано в розділі 2.1, мікросервісна архітектура передбачає розбиття системи на окремі сервіси. З іншого боку, в безсерверній архітектурі модулі розбивають ще дрібніше, на окремі функції. На відміну від мікросервісу, який може охоплювати різні типи застосунків, безсерверна функція зазвичай містить код лише для цієї конкретної задачі. Це означає, що будь-який шаблонний код, наприклад, налаштування REST API, абстрагується і управляється провайдером FaaS.

На рисунку 2.2, зображено приклад по якому зазвичай будується безсерверна архітектура. Дана структура включає в себе декілька функцій FaaS, що відповідають за різний функціонал застосунку. Ці функції розміщені за API-шлюзом, який спрямовує запити з клієнтської частини до відповідної функції. Налаштований на запуск за HTTP-запитом, при активації хмарний провайдер ініціює екземпляр, виконує код і згодом вимикається.

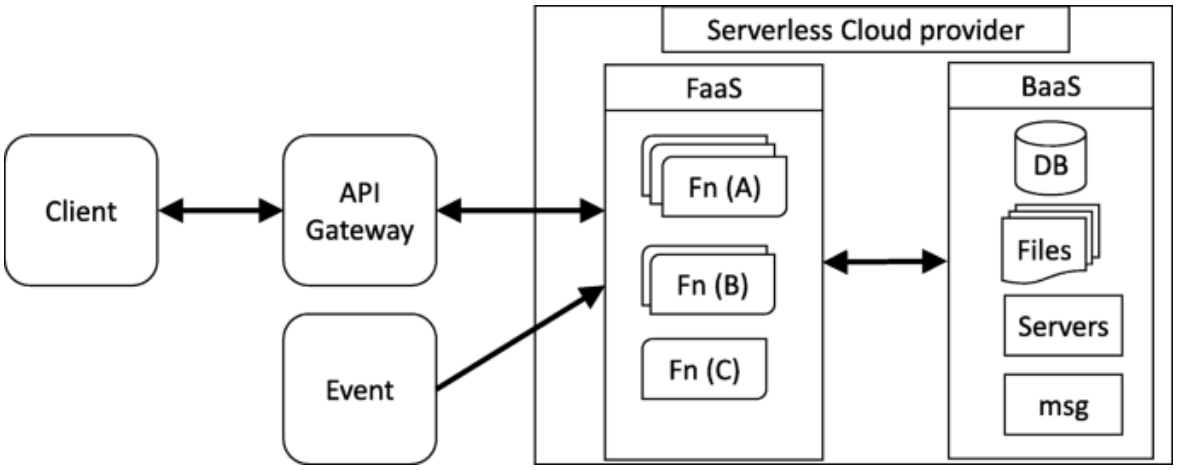

Рисунок 2.2 – Приклад функціонування безсерверної архітектури [24]

Дана безсерверна архітектура також інтегрує рішення Backend-as-a-Service (BaaS) для автентифікації та зберігання файлів, та даних в БД. На відміну від монолітних та мікросервісних підходів, ця архітектура абстрагується від усього, окрім основної бізнес-логіки, делегуючи управління серверами та відповідальність за масштабування стороннім сервісам. Більш складні додатки, розроблені в цій структурі, можуть використовувати кілька взаємопов'язаних безсерверних функцій для побудови складної логіки та систем.

2.1.3 Переваги та недоліки безсерверної архітектури

Використання безсерверної архітектури дозволяє створювати складні застосунки з простих функцій, пропонуючи ряд переваги. FaaS дозволяє запускати внутрішній код без необхідності керувати серверними системами, дозволяючи розробникам зосередитися на логіці застосунків, в той час як хмарний провайдер займається інфраструктурою та розгортанням.

Безсерверні архітектури забезпечують автоматичне масштабування, знижуючи витрати на інфраструктуру за рахунок розподілу ресурсів на основі фактичного використання. Така економічна ефективність особливо вигідна для застосунків з динамічним трафіком, де екземпляри необхідно масштабувати за потреби. Дослідження показують, що безсерверні архітектури можуть значно скоротити витрати на інфраструктуру.

З точки зору екології, безсерверні хмарні обчислення сприяють зменшенню споживання енергії за рахунок ефективного розподілу обчислювальних ресурсів за потребою. Такий підхід до спільної інфраструктури мінімізує потребу в центрах обробки даних, що призводить до зниження загального споживання енергії та позитивного впливу на навколишнє середовище.

Незважаючи на переваги, безсерверні архітектури мають і недоліки. Відмінності в реалізації FaaS у різних хмарних провайдерів можуть призвести до прив'язки до одного постачальника, що робить міграцію платформи складною і дорогою. Розробники втрачають контроль над певними аспектами своїх застосунків, стикаючись з обмеженнями в конфігурації та оптимізації обладнання через абстракцію базових компонентів.

Втрата контролю поширюється на вирішення проблем та безпеку, оскільки постачальники послуг керують базовою інфраструктурою та заходами безпеки.

Stateless природа безсерверних функцій створює проблеми в управлінні станом, вимагаючи зовнішнього зберігання стану програми, наприклад, токенів сеансів.

Крім того, “холодний старт” у FaaS призводить до затримок при виконанні функцій, що вимагає виділення “теплих екземплярів” з метою пришвидшення. На противагу цьому, “теплі старти” відбуваються, коли функції викликаються з уже існуючими контейнерами, що призводить до швидшого виконання. Хмарні провайдери намагаються зменшити затримки холодного старту, підтримуючи попередньо налаштовані пули серверів, але завантаження файлів і налаштувань у пам'ять все одно може спричинити затримку порівняно з теплим запуском.

Ще однією вагомою проблемою що несе за собою FaaS є кількість підключень до бази даних. Оскільки функції є короткотривалими, кожен новий екземпляр функції буде створювати нове з'єднання з базою даних. При значному навантаженні це може призвести до вичерпання доступних з'єднань у пулі з'єднань бази даних.

2.1.4 FaaS платформи

Першою комерційною платформою FaaS стала AWS Lambda, запущена компанією Amazon у 2014 році. Відповідником AWS Lambda від Microsoft є Azure Functions, випущений у 2016 році. Нещодавно, у 2018 році, Google запустив релізну версію своєї платформи безсерверних обчислень під назвою Google Cloud Functions. Платформи пропонують схожий функціонал, але існують певні відмінності, наприклад, у підтримуваних мовах програмування, вартості, моніторингу та налагодженні. Платформи також тісно інтегровані із загальною хмарною платформою компаній, що означає легке підключення BaaS-сервісів, таких як API-шлюзи та бази даних, що пропонуються відповідними платформами.

Окрім комерційних платформ, існують також платформи з відкритим вихідним кодом. Ці платформи дозволяють запускати безсерверні функції на власній інфраструктурі. Деякі з них – Apache OpenWhisk, OpenFaaS та OpenLambda.

В таблиці 2.1 нижче наведено порівняльний аналіз між даними платформами.

Таблиця 2.1 – відмінності в функціоналі між різними платформами постачальниками FaaS

Функції	AWS Lambda	Microsoft Azure Functions	Google Cloud Functions	Open Whisk	Open FaaS	Open Lambda
Інтерфейс	CLI/API/ GUI	CLI/API/ GUI	CLI/API/G UI	CLI/API	CLI/API/ GUI	CLI/API
Модель оплати	Тривалість виконання та спожита пам’ять	Тривалість виконання та виділена пам’ять	Тривалість виконання та виділена пам’ять	Невідомо	Невідомо	Невідомо
Тайм аут функції	900 секунд	600 секунд	540 секунд	600 секунд	30 секунд	Невідомо
Обмеження розміру коду	250 МБ не стиснутий та 50 МБ стиснутий	Немає	250 МБ не стиснутий та 100 МБ стиснутий	48 МБ стиснутий	Невідомо	Невідомо
Діапазон виділеної пам’яті	128 – 1024 МБ	128 – 4096 МБ	128 – 2048 МБ	Невідомо	Невідомо	Невідомо
Вбудовані засоби моніторингу	AWS Cloud Watch	Azure Application Insights	Google Cloud Operations	Немає	Немає	Немає

Як вже було згадано у розділі вище, прив'язка до конкретного постачальника є суттєвим недоліком, оскільки програмна реалізація відрізняється між платформами та потребує додаткових навичок. Пропонованим рішенням цієї проблеми є Serverless Framework. Serverless Framework – це популярний фреймворк з відкритим вихідним кодом для розробки та розгортання безсерверних застосунків, що підтримує більшість платформ серед яких вище згадані комерційні платформи. Фреймворк пропонує командний інтерфейс для створення та налаштування безсерверних проєктів, включаючи функції FaaS та інші ресурси хмарної інфраструктури.

2.2 Тестування продуктивності

2.2.1 Характеристика тестування продуктивності

Для визначення ефективності та масштабованості в розробці програмних застосунків зазвичай використовується поєднання різних видів функціонального

та не функціонального тестування. У даній роботі для оцінки ефективності та масштабованості буде використано саме нефункціональний вид тестування – тестування продуктивності.

Тестування продуктивності – це процес тестування застосунку шляхом імітації реальних користувачів за допомогою інструменту, що генерує навантаження, з метою виявлення вузьких місць системи. Часто його також називають тестуванням навантаження. Основна мета – перевірка масштабованості, доступності та продуктивності з точки зору як апаратного, так і програмного забезпечення [17].

Існує декілька типів тестування продуктивності, призначених для різних сценаріїв оцінки. Наприклад, навантажувальне тестування досліджує, як система поводиться за очікуваних рівнів навантаження, щоб оцінити такі характеристики продуктивності, як час відгуку та пропускна здатність. Стресове тестування, з іншого боку, навантажує систему більше її звичайної потужності, щоб виявити точки відмови та виміряти стійкість. Тестування на витривалість оцінює стабільність системи протягом тривалих періодів за постійного навантаження, тоді як тестування на пікові навантаження оцінює реакцію на раптові стрибки трафіку, а тестування обсягом даних перевіряє масштабованість за великих обсягів даних.

Тестування продуктивності спирається на конкретні показники для точного кількісного визначення продуктивності та поведінки системи. Час відгуку вимірює, скільки часу потрібно системі, для відповіді на запити користувачів, враховуючи затримку обробки та мережі. Пропускна здатність визначає швидкість, з якою система обробляє транзакції за певний проміжок часу. Паралельність оцінює здатність системи обробляти декількох одночасних користувачів або транзакцій. Частота помилок відстежує виникнення помилок або невдалих транзакцій під час тестування, що відображає стабільність системи. Показники використання ресурсів, такі як завантаження центрального процесора, пам'ять, дисковий ввід-вивід та використання мережі, надають інформацію про вузькі місця ресурсів та можливості оптимізації.

Результати навантажувального тестування зазвичай відображаються у вигляді гістограми яка в свою чергу відображає статистичний розподіл набору значень протягом вимірюваного інтервалу [11]. Зазвичай в переліку метрик гістограми відображаються наступні значення:

- Мінімальне значення (Min) відповідає найнижчому спостережуваному значенню певної метрики під час навантажувального тестування.
- Максимальне значення (Max) відповідає найвищому спостережуваному значенню певної метрики під час навантажувального тестування.
- Середнє значення (Mean) – це середнє арифметичне всіх спостережуваних значень певної метрики під час навантажувального тестування.
- Медіана – це значення, яке ділить упорядкований набір даних на дві рівні половини, причому 50% спостережень менше за медіану, а інші 50% – більші. У навантажувальному тестуванні медіанний час відгуку означає точку, в якій половина запитів отримала відповідь швидше за це значення, а інша половина - повільніше.
- Значення p90 – це пороговий час відгуку, нижче якого під час навантажувального тестування потрапляють 90% спостережень.
- Значення p99 – це пороговий час відгуку, нижче якого під час навантажувального тестування потрапляють 99% спостережень. Ця метрика є критичною для виявлення аномальних значень.

Перейдемо до інструментів тестування продуктивності.

2.2.2 Інструменти тестування продуктивності

Інструмент навантажувального тестування імітує поведінку реальних користувачів за допомогою "віртуальних" користувачів. Потім інструмент навантажувального тестування записує поведінку під навантаженням і надає інформацію про досвід віртуальних користувачів. Програмне забезпечення для навантажувального тестування часто є розподіленим за своєю природою. Воно розгортається на декількох серверах, що працюють одночасно, причому кожен сервер імітує декілька віртуальних користувачів.

Інструмент Artillery.io [11] – це інструмент, який використовується і пропонується для бенчмаркінгу FaaS-платформ і мікросервісів. Artillery – це набір інструментів для навантажувального та функціонального тестування з відкритим вихідним кодом. Він може імітувати роботу користувачів з застосунком, надсилаючи велику кількість мережевих запитів до вказаного веб-сайту або застосунку. Інструмент CLI (Command-Line-Interface) дозволяє визначати складні тестові сценарії, де користувачі можуть вказувати HTTP-запити і корисне навантаження даних, які будуть доставлені в додаток. Це робить його ідеальним інструментом для тестування продуктивності та поведінки застосунків, які взаємодіють через REST API.

Іншим інструментом з відкритим вихідним кодом, що використовується для бенчмаркінгу, є JMeter [9]. JMeter також використовується для бенчмаркінгу REST застосунків і являє собою Java-застосунок для тестування продуктивності статичних і динамічних веб-застосунків. Як і Artillery, він може імітувати високий користувацький трафік до застосунку і підтримує широкий спектр мережевих протоколів, наприклад, HTTP, HTTPS, REST тощо.

Інструмент k6 [23] – ще один потужний інструмент, який зазвичай використовується для тестування продуктивності. Він призначений для навантажувального тестування, стрес-тестування та моніторингу API, і веб-застосунків. k6 використовує мову JavaScript для написання тестових сценаріїв, що робить його популярним серед більшості розробників.

Перейдемо до інструментів тестування продуктивності.

2.3 CI/CD конвеєри

2.3.1 Характеристика CI/CD конвеєрів

Безперервна інтеграція (CI) та безперервна доставка/розгортання (CD) є взаємодоповнюючими практиками, що працюють разом для підвищення загальної якості та ефективності процесу розробки програмного забезпечення.

CI фокусується на ранніх стадіях конвеєру, де код збирається та проходить початкове тестування. Кілька розробників працюють над однією базою коду одночасно, часто (за можливості) роблячи коміти до спільного репозиторію. CI використовує різноманітні інструменти та методи автоматизації для створення збірок та проведення їх через початкове тестування, таке як модульне та інтеграційне тестування. CI відрізняється швидким і детальним циклом зворотного зв'язку, що означає, що помилки виявляються, локалізуються та виправляються відносно легко.

Після того, як збірка успішно пройде початкове тестування CI, вона переходить до більш комплексного тестування та перевірки на етапі CD. CD фокусується на пізніших стадіях конвеєру, де завершена збірка ретельно тестується, перевіряється та готується до розгортання. Це включає функціональне, приймальне, конфігураційне та за можливості навантажувальне тестування, щоб гарантувати, що збірка відповідає вимогам і готова до використання у продуктовому середовищі.

Ключова відмінність між безперервною доставкою та безперервним розгортанням полягає в тому, що безперервне розгортання автоматично розгортає кожну перевірену збірку в робоче середовище, тоді як безперервна доставка зазвичай просто підготовляє перевірену збірку для ручного розгортання або іншої людської авторизації.

Конвеєр – це абстракція або план виконання, що поєднує в собі обидві практики CI та CD (рисунок 2.3).

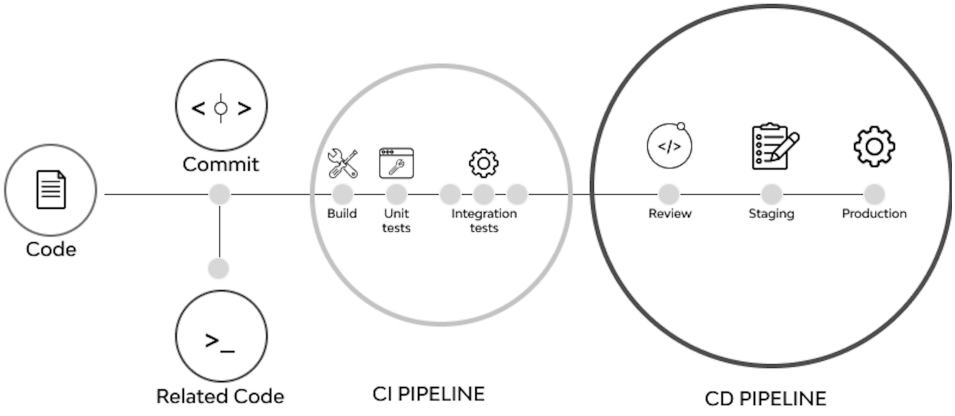

Рисунок 2.3 – Життєвий цикл конвеєру безперервної доставки та інтеграції

Конвеєри використовуються для підвищення загальної якості випусків програмного застосунку, шляхом автоматизації, надаючи статистику та моніторинг стану етапів релізу, інтеграції та тестування проєкту.

2.3.2 Ключові показники ефективного CI/CD конвеєру

Серед показників ефективного CI/CD конвеєру виділяють наступні [40]:

- Швидкість. Конвеєр повинен сприяти швидкому проходженню збірки через усі етапи, включаючи інтеграцію, тестування, доставку та розгортання. В ідеалі, інтеграція повинна бути завершена протягом декількох хвилин, а цикли тестування – протягом декількох годин. Тривале виконання конвеєру (дні) свідчить про потенційну неефективність, яка потребує оптимізації.
- Консистентність. Процеси та компоненти, що використовуються на певному етапі, повинні бути абсолютно однаковими на всіх інших етапах. Це забезпечує передбачуваність та сприяє створенню надійних сценаріїв автоматизації. Навпаки, процеси, що вводять мінливість або ручні кроки, перешкоджають ефективності конвеєру та підвищують сприйнятливість до помилок.
- Детальний контроль версій. Добре документовані репозитарії та ретельний контроль версій компонентів та збірок дозволяють швидко відновлювати попередні версії у випадку збоїв розгортання або збірки, що сприяє швидкому відкату до попередніх робочих версій, за потреби.
- Автоматизація. Широка автоматизація мінімізує ручне втручання, в ідеалі вимагаючи лише розробки коду, його коміту та потенційного схвалення перед розгортанням. Конвеєри, що залежать від ручних кроків, є за своєю природою повільними та схильними до помилок.
- Інтегровані механізми зворотного зв'язку: CI/CD конвеєр функціонує як ітераційний цикл, де кожен етап надає зворотний зв'язок, який може призвести до повернення до попередніх етапів. Наприклад, проблеми з вихідним кодом перешкоджають генерації збірки, помилки збірки зупиняють процес тестування, а проблеми з тестуванням або розгортанням вимагають виправлення коду.

- Інтеграція безпеки. Помилки в розробці та тестуванні можуть призвести до появи вразливостей та піддати застосунок зловмисній активності. Тому, найкращі практики безпеки необхідно суворо інтегрувати протягом усього CI/CD конвеєра.

Перейдемо до платформ розгортання CI/CD конвєрів.

2.3.3 Платформи для розгортання CI/CD конвеєрів

Існує безліч платформ що пропонують розміщення власних CI/CD конвеєрів: Jenkins, TravisCI, GitLab Actions, GitHub Actions. Порівняльний аналіз функціональні відмінностей цих платформ наведено в таблиці 2.2.

Таблиця 2.2 – Функціональні відмінності платформ для розгортання конвеєрів безперервної інтеграції та доставки

Функція	Jenkins	Travis CI	GitLab Actions	GitHub Actions
Модель розгортання	Ручна	Хмарна	Хмарна	Хмарна
Інтеграція з репозиторіями	Будь-який Git-репозиторій	GitHub, Bitbucket	GitLab	GitHub
Налаштування	На основі UI або Jenkinsfile	YAML-файл	YAML-файл	YAML-файл
Зручність використання	Потребує налаштування та обслуговування	Простіше налаштування	Легке налаштування	Легке налаштування
Масштабованість	Висока масштабованість з додатковими серверами	Автоматично	Масштабується з планом	Масштабується з планом
Плагіни та інтеграції	Велика екосистема плагінів	Обмежені інтеграційні можливості	Широкий спектр інтеграцій в межах GitLab та зовнішніми платформами	Широкий спектр інтеграцій в межах GitHub та зовнішніми платформами
Цінова політика	Freemium	Paid / Freemium (Обмеження по к-сті збірок)	Paid / Freemium (Обмеження по к-сті хвилин)	Paid / Freemium (Обмеження по к-сті хвилин)

В даній роботі в контексті написання CI/CD конвеєру буде використовуватися GitHub Actions.

GitHub Actions – це SaaS платформа для безперервної інтеграції та доставки (CI/CD), яка дозволяє автоматизувати рутинні процеси розробки програмного забезпечення. Вона дає змогу користувачам визначати та виконувати робочі процеси, що інтегруються з їхніми репозиторіями GitHub, автоматизуючи такі завдання, як збирання, тестування та розгортання коду. GitHub Actions пропонує гнучку та масштабовану інфраструктуру для автоматизації цих завдань, які можуть запускатися різними подіями, такими як коміти, pull-реквести або ж вручну.

Основною концепцією GitHub Actions є робочі процеси (рисунок 2.4). Робочий процес (workflow), визначений у YAML-файлі, що зберігається в каталозі .github/workflows, і являє собою налаштовуваний автоматизований процес, що складається з однієї або декількох робіт (jobs). Кожна робота складається з серії кроків (steps), а кожен крок використовує одну або кілька дій (actions).

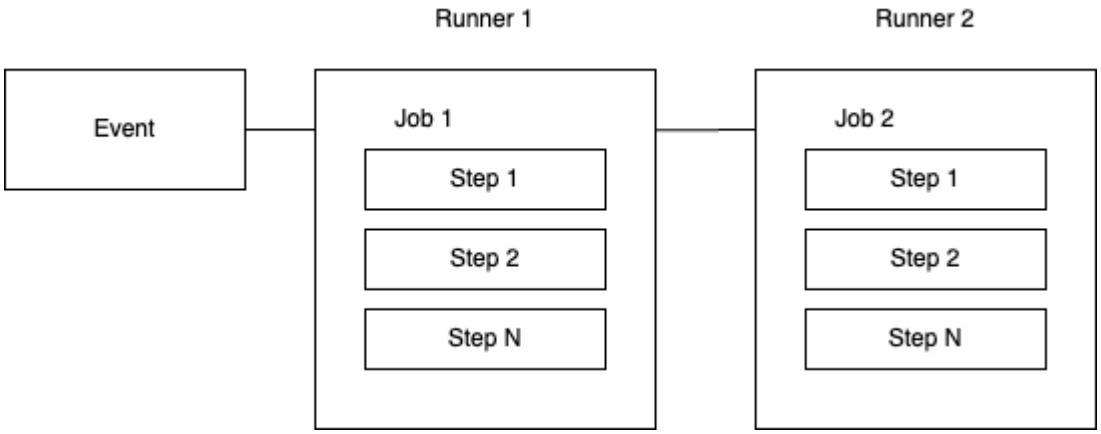

Рисунок 2.4 – Робочий процес GitHub Actions [43]

Робочі процеси виконуються на виконавцях (runners) – програмному забезпеченні, розгорнутому на на фізичних або віртуальних серверах. Виконавці можуть бути самостійно розміщені на користувацькиї машинах або ж розміщені та керовані GitHub.

Дії – це попередньо визначені функціональні можливості, які можна комбінувати для побудови складних робочих процесів. Ці дії можуть бути розроблені спільнотою GitHub або створені окремими користувачами.

Облікові дані (credentials), що використовуються в робочих процесах, безпечно зберігаються як секрети (secrets) в організації, сховищі або середовищі сховища. Секрети шифруються та безпечно зберігаються на серверах GitHub, і розшифровуються лише під час виконання робочого процесу. Доступ до секретів можна додатково обмежити за допомогою механізмів контролю доступу, що надаються GitHub. Робочі процеси можуть посилатися на ці секрети, при цьому вміст приховується з логів для підвищення безпеки.

GitHub Actions надає певну квоту сховища та безкоштовних хвилин для виконання робочих процесів, яка залежить від рівня підписки користувача. Безкоштовна підписка надає 500 МБ сховища та 2 000 хвилин для виконання завдань кожного місяця. Хвилини виконання множаться на визначений коефіцієнт в залежності від операційної системи, на якій виконується робота.

2.4 Висновок до другого розділу

У другому розділі кваліфікаційної роботи досліджено методологію безсерверних обчислень, методологію тестування продуктивності та концепцію безперервної інтеграції та доставки (CI/CD).

Спочатку було проаналізовано переваги та особливості безсерверної парадигми обчислень (Function-as-a-Service, FaaS), а також провідні комерційні та відкриті платформи для розгортання та виконання безсерверних функцій, такі як AWS Lambda, Microsoft Azure Functions, Google Cloud Functions та інші.

Далі розглянуто методологію тестування продуктивності як основний підхід для кількісної оцінки ефективності та масштабованості застосунків. Коротко описано основні типи тестування продуктивності, ключові показники, що вимірюються, а також популярні інструменти навантажувального тестування, такі як Artillery.io, JMeter та k6.

Окремо висвітлено концепцію CI/CD конвеєрів, їх ключові показники ефективності та переваги впровадження. Також проаналізовано провідні платформи для розгортання та управління CI/CD конвеєрами, зокрема GitHub Actions, яка буде використовуватися в практичній частині роботи.

3 ВПРОВАДЖЕННЯ ПРОЦЕСІВ ОПТИМІЗАЦІЇ НА ПРИКЛАДІ МУЗИЧНОЇ ПЛАТФОРМИ

Третій розділ присвячений впровадженню процесів оптимізації на прикладі музичної платформи. Спочатку проаналізовано функціональні та нефункціональні вимоги до експериментальної системи. Далі розглядається підхід до декомпозиції монолітної архітектури на безсерверні функції з використанням моделі безсерверних обчислень. Потім надається детальний опис запропонованої безсерверної архітектури музичної платформи, процесу її розгортання з використанням Serverless Framework та AWS сервісів. Наводяться результати навантажувального тестування системи з аналізом її ефективності та масштабованості. Також описується процес реалізації CI/CD конвеєра мобільного застосунку музичної платформи.

3.1 Аналіз вимог до експериментальної системи

3.1.1 Зміни та покрашення вихідної системи

З метою дослідження та впровадження процесів оптимізації розробки, спрямованих на підвищення ефективності та масштабованості, в якості експериментальної системи було обрано функціонал веб-застосунку музичної платформи, розробленого в рамках попередньої кваліфікаційної роботи освітнього рівня "бакалавр" [31].

Порівняно з вихідною системою були здійснені наступні зміни та покращення:

- розширено перелік функціональних та нефункціональних вимог;
- частково змінено комплекс використовуваних технологій;
- реалізовано мобільну версію застосунку.

Перейдемо до функціональних вимог до системи.

3.1.2 Функціональні вимоги

Нижче наведено перелік розширеного списку функціональних вимог до експериментальної системи музичної платформи.

Система авторизації та реєстрації користувачів:

- можливість реєстрації нових користувачів з унікальними обліковими даними з використанням сторонніх API;
- аутентифікація існуючих користувачів за допомогою логіна та пароля та можливості використання сторонніх провайдерів у майбутньому;

Управління профілем користувача:

- персоналізація облікового запису, включаючи завантаження фотографії профілю та редагування персональних даних;
- можливість створення та редагування власної бібліотеки музики;
- збереження історії прослуховування та пропозицій на основі вподобань користувача.

Потік музики:

- підтримка різних форматів аудіо;
- можливість завантаження власних треків;
- можливість створення та редагування плейлистів.

Пошук та відкриття музики:

- розширений пошук за назвою треку, виконавцем, чи жанром;
- можливість переглядати нові завантаження та топові композиції;
- підтримка обох платформ iOS та Android;
- зрозумілий зручний інтерфейс.

Відповідно до функціональних вимог діаграма варіантів використання набуває наступного вигляду (рисунок 3.1).

У даній системі у ролі акторів розглядатиметься лише користувач музичної платформи.

3.1.3 Нефункціональні вимоги

- Масштабованість: система повинна бути спроєктована з урахуванням можливості горизонтального та вертикального масштабування для належного реагування на зміни навантаження та зростання кількості користувачів.
- Ефективність: система має раціонально використовувати обчислювальні ресурси, уникаючи невиправданих витрат та простоїв.

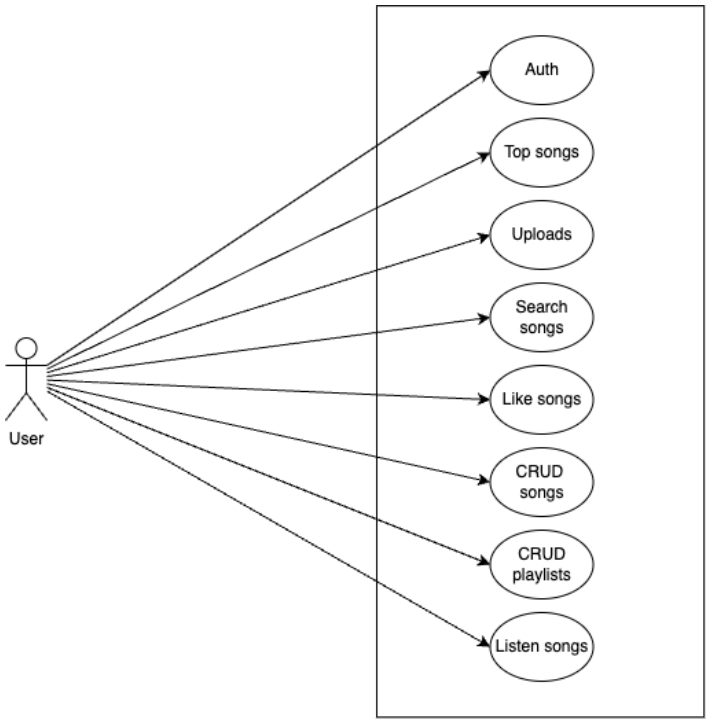

Рисунок 3.1 – Варіанти використання музичної платформи

- Висока доступність: система повинна забезпечувати безперебійну роботу та стійкість до відмов при передбачуваному навантаженні.
- Використання сучасних технологій: система має бути реалізована з використанням актуальних мов програмування, фреймворків та бібліотек, що відповідають найкращим практикам розробки програмних застосунків.
- Оптимізація: в основу системи повинні бути покладені методи та підходи до оптимізації, наведені в першому розділі роботи.

Перейдемо до принципів декомпозиції монолітної архітектури на безсерверні функції.

3.2 Принцип декомпозиції монолітної архітектури на безсерверні функції

Монолітна архітектура, що відповідає парадигмі багаторівневої архітектури [30], передбачає розподіл функціональності між кількома логічними рівнями, при цьому кожен рівень відповідає за певний аспект обробки даних та взаємодії з іншими компонентами системи.. У даному випадку архітектура складається з трьох основних рівнів: представлення, бізнес-логіки та доступу до даних, а також зовнішнього рівня сторонніх сервісів.

Рівень представлення реалізований у вигляді RESTful API з набором кінцевих точок, які приймають HTTP-запити від клієнтських застосунків. Ці запити перенаправляються відповідно до визначених маршрутів до відповідних контролерів на рівні бізнес-логіки.

Рівень бізнес-логіки містить контролери, які інкапсулюють логіку обробки даних відповідно до функціональних вимог системи. Наприклад, контролер користувачів відповідає за операції зі створення, читання, оновлення та видалення об'єктів користувачів, реалізуючи функції авторизації, реєстрації та управління обліковими записами. Аналогічно, контролер треків забезпечує виконання CRUD операцій над об'єктами музичних треків, реалізуючи функції завантаження, пошуку та відтворення музичного контенту.

Рівень доступу до даних відповідає за взаємодію зі сховищем даних, яким є база даних. Для цього використовуються моделі даних, які абстрагують деталі зберігання та доступу до даних, надаючи уніфікований інтерфейс для роботи з об'єктами.

Крім внутрішніх рівнів, система взаємодіє з зовнішнім рівнем сторонніх сервісів, таких як сервіси авторизації, сповіщень або інтеграції з іншими платформами.

На рисунку 3.2 наведено монолітну архітектуру музичної платформи, де відображено взаємозв'язок між різними рівнями та компонентами.

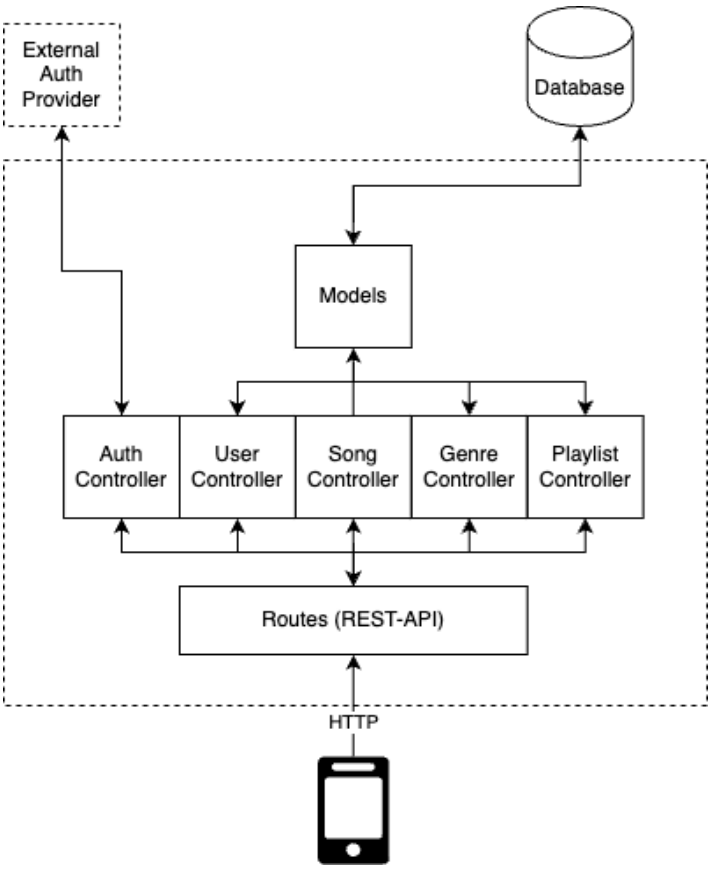

Рисунок 3.2 – Монолітна архітектура музичної платформи

Монолітна архітектура – це найпростіший підхід до проєктування та реалізації програмних систем. Така архітектура є простою на початкових етапах розробки, оскільки не потрібно розбивати систему на окремі частини. Однак, з розростанням і ускладненням вимог до системи, монолітна архітектура стає менш гнучкою та важче піддається змінам і масштабуванню. Це є одним з основних недоліків монолітного підходу [29].

Безсерверна архітектура – це підхід до переходу від цілком монолітної архітектури до частково розподіленої, шляхом декомпозиції існуючого монолітного REST API, на декілька безсерверних функції [29] (рисунок 3.3) з використанням моделі безсерверних обчислень.

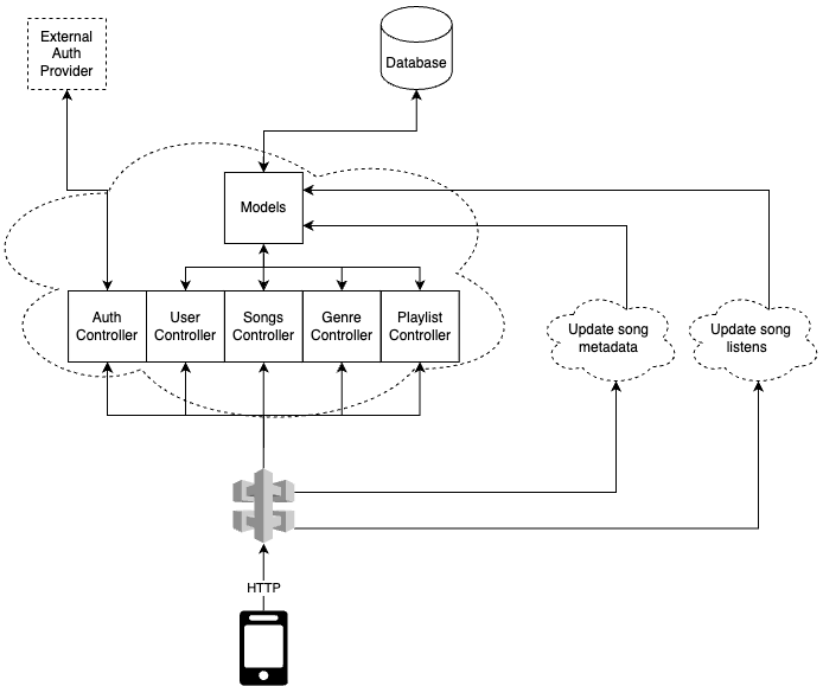

Рисунок 3.3 – Приклад декомпозиції функціональності монолітного застосунку

Таким чином, замість єдиного монолітного застосунку, система розбивається на набір розподілених, незалежних функціональних компонентів, які можуть масштабуватися та оновлюватися окремо один від одного. Такий підхід подібний до мікросервісної архітектури, проте з використанням безсерверних функцій замість традиційних мікросервісів.

3.3 Проєктування архітектури музичної платформи з впровадженням процесів оптимізації

Як хмарну платформу для розгортання музичної платформи було обрано Amazon Web Services (AWS). Згідно зі звітом компанії Canalys, AWS займає найбільшу частку ринку серед провайдерів комерційних хмарних рішень. Крім того, дослідження показують, що час холодного запуску безсерверних функцій AWS Lambda є більш стабільним між численними викликами, а також вони забезпечують кращу продуктивність процесора, пропускну здатність мережі та пропускну здатність вводу/виводу під час масштабування рішення [34]. AWS також надає широкий спектр сервісів, що легко інтегруються з Lambda, наприклад, DynamoDB, Kinesis, SNS, SQS, що полегшує розробку та впровадження безсерверної архітектури.

На рисунку 3.4 представлено безсерверну архітектуру музичної платформи з розгортанням на AWS.

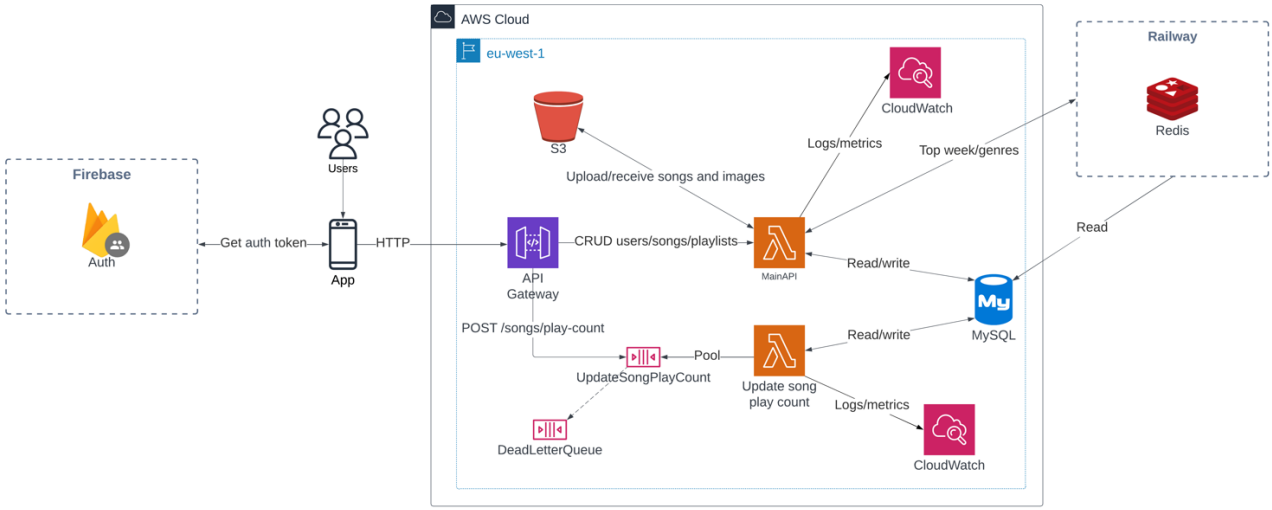

Рисунок 3.4 – Безсерверна архітектура музичної платформи

У представленій безсерверній архітектурі використовується низка хмарних сервісів для задоволення функціональних та нефункціональних вимог системи.

Для забезпечення процесів автентифікації використовується BaaS Firebase Auth. Даний сервіс надає готову та масштабовану реалізацію процесів автентифікації з можливістю інтеграції з численними провайдерами автентифікації, такими як Google, Apple, Facebook тощо.

Для кешування даних та зменшення навантаження на базу даних використовується хмарний сервіс Railway Redis – повністю керований екземпляр NoSQL бази даних Redis. Redis є розподіленим сховищем типу "ключ-значення", що характеризується високою продуктивністю.

Основні обчислювальні потужності забезпечуються сервісом AWS Lambda, який дозволяє розгортати та виконувати безсерверні функції без необхідності управління інфраструктурою серверів. Ці функції реалізують ключову бізнес-логіку застосунку та автоматично масштабуються відповідно до навантаження.

AWS API Gateway використовується для розгортання RESTful API, через який відбувається взаємодія з клієнтською частиною системи. API Gateway проксіює вхідні запити до відповідних Lambda функцій чи сервісів.

Для асинхронної обробки завдань та повідомлень застосовується черга повідомлень AWS SQS (Simple Queue Service). Це дозволяє розподілити навантаження та паралельно оброблювати великі обсяги даних.

Зберігання файлів, музичних треків та зображень здійснюється в об'єктному сховищі AWS S3 (Simple Storage Service), що характеризується високою масштабованістю та доступністю (99,999999999%) [7].

Для зберігання постійних даних використовується реляційна база даних MySQL, розгорнута з використанням сервісу AWS RDS (Relational Database Service).

Моніторинг та логування виконання Lambda функцій, API Gateway та інших компонентів забезпечується сервісом AWS CloudWatch.

3.4 Розгортання архітектури музичної платформи

Як зазначено у розділі 3.1, було прийнято рішення частково змінити комплекс технологій розробки, зокрема мігрувати серверну частину на фреймворк NestJS замість Express.js. Nest (NestJS) – це фреймворк для створення ефективних, масштабованих Node.js серверних застосунків [31]. NestJS є надбудовою над Express.js, що розширює його функціональність та впроваджує різні методології та патерни, запозичені з Angular. Однією з ключових переваг NestJS є можливість структурування коду за допомогою модульної архітектури та використання ін'єкції залежностей. Ця парадигма сприяє збереженню чистоти коду, підвищує його читабельність та розширюваність, що є критично важливими аспектами для проєктів будь-якої складності. Крім того, NestJS використовує TypeScript як основну мову програмування. TypeScript є строго типізованою мовою, що дозволяє виявляти помилки на етапі компіляції, зменшуючи кількість потенційних помилок та полегшуючи процес

рефакторингу. Також варто відзначити вбудовані інструменти інтеграційного тестування та Swagger.

Для розгортання ресурсів на AWS було вирішено використовувати Serverless Framework. Цей відкритий інструмент забезпечує абстракцію над різними хмарними платформами для розробки та розгортання застосунків на базі хмарних функцій (FaaS). Serverless Framework використовує парадигму "інфраструктура як код", де конфігурація застосунку та його ресурси визначаються в декларативному файлі конфігурації. Фреймворк автоматизує процеси упакування, розгортання та керування життєвим циклом хмарних функцій, забезпечуючи можливість налаштування подій, що ініціюють їх виконання. Крім того, він надає можливість локального тестування функцій та має екосистему плагінів для розширення функціональності. Зазвичай для написання конфігураційного файлу serverless використовується мова yml, але підтримка Typescript також вбудована.

Під час проєктування архітектури музичної платформи було визначено декілька ресурсів, які будуть описані в конфігураційному файлі Serverless Framework. Зокрема, буде оголошено основну функцію, яка міститиме REST API та буде викликатися при переадресуванні запитів від API Gateway. Також буде створено чергу повідомлень SQS (Simple Queue Service), в яку надсилатимуться повідомлення від кінцевої точки, оголошеної в API Gateway. Для забезпечення надійності та повторної обробки невдалих повідомлень буде створено чергу мертвого списку (Dead Letter Queue), яка міститиме повідомлення, що не були успішно оброблені обробником. Крім того, буде реалізовано асинхронну функцію для оновлення кількості прослуховувань. Конфігураційний файл також міститиме оголошення необхідних дозволів в IAM (Identity and Access Management) та їх призначення відповідним ресурсам.

У лістингу 3.1 представлено фрагмент коду у форматі AWS CloudFormation, який визначає ресурс черги повідомлень (UpdateSongPlayCountQueue), призначений для асинхронної обробки оновлення кількості прослуховувань музичних треків в системі.

Лістинг 3.1 – Фрагмент коду з оголошення черги оновлення кількості прослуховувань музичного треку

```
import type { CloudFormationResource } from "serverless/aws";

const UpdateSongPlayCountQueue: CloudFormationResource = {
  Type: "AWS::SQS::Queue",
  Properties: {
    QueueName:
"${self:custom.UpdateSongPlayCountQueue.queueName}",
    VisibilityTimeout: 300,
    ReceiveMessageWaitTimeSeconds: 20,
    RedrivePolicy: {
      deadLetterTargetArn:
"${self:custom.DeadLetterQueue.arn}",
      maxReceiveCount: 1,
    },
  },
};
```

Весь перелік ресурсів визначений в конфігураційному файлі serverless.ts представлено у лістингу Б.1 додатку Б.

Для розгортання описаних ресурсів використовується npm залежність serverless, куди можуть передаватися такі аргументи як stage та aws-profile, у разі якщо його не визначено по замовчуванню.

```
sls deploy -s <stage >--aws-profile <назва профілю>
```

Додатково для зменшення розміру архіву коду, з метою пришвидшення виконання функцій було використано ncc компілятор, що забезпечив компіляцію усіх модулів в один JavaScript файл разом з усіма залежностями. Це дозволило зменшити розмір архіву на 86%.

Для розгортання екземпляру Redis використовується хмарний сервіс Railway.app. Ця платформа пропонує альтернативу управлінню складною

інфраструктурою, фокусуючись на підвищенні продуктивності розробників за рахунок автоматизації завдань розгортання та масштабування застосунків.

На рисунку 3.5 наведено розгортання екземпляру Redis з використанням хмарного сервісу Railway.app.

Для підключення до екземпляру Redis та реалізації кешування використовуються такі npm залежності як cache-manager-redis-store та @nestjs/cache-manager.

У лістингу 3.2 наведено реалізацію кешування списку музичних жанрів з використанням вищезгаданих залежностей.

Лістинг 3.2 – Реалізація кешування списку музичних жанрів

```
@UseInterceptors(CacheInterceptor)
@CacheKey('song-genres')
@CacheTTL(60 * 60 * 24) // cache for 24 hours
@Get('cache')
async getAllCached() {
  return this.genresService.getAll()
}
```

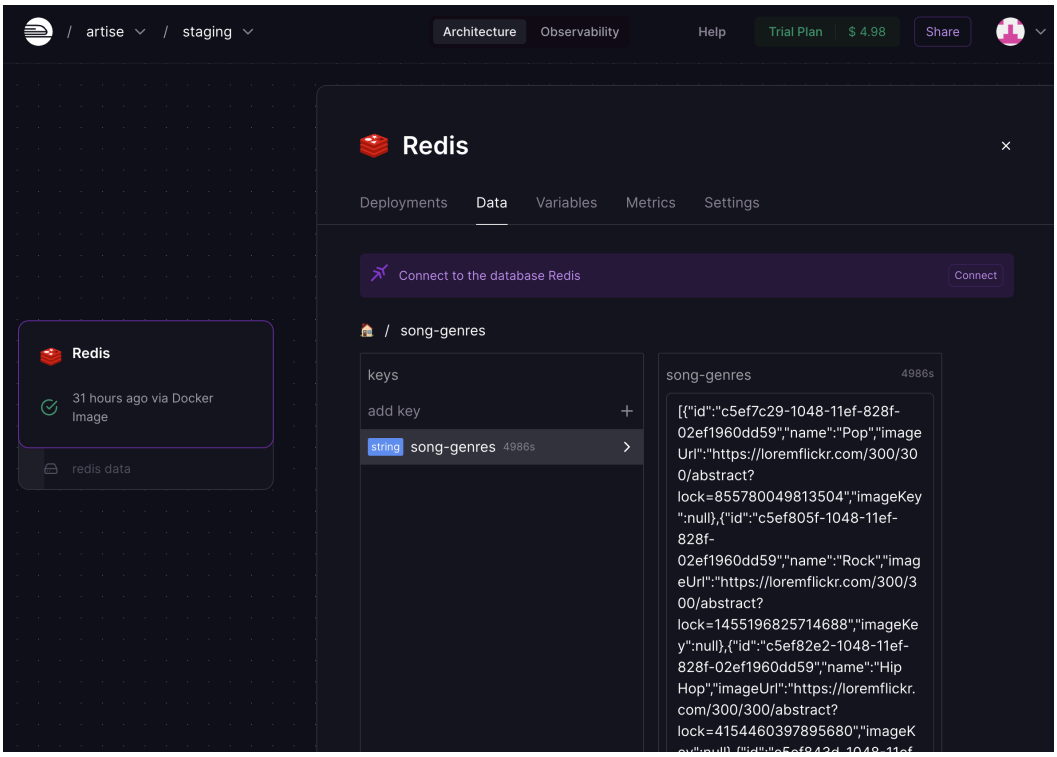

Рисунок 3.5 – Екземпляр Redis розміщений на хмарному сервісі Railway.app

У даному випадку, при первинному запиті на отримання списку жанрів, запит буде спрямовано до реляційної бази даних. Після успішного отримання результатів, вони будуть збережені в кеші Redis з життєвим циклом 24 години. Протягом цього періоду, при надходженні HTTP-запитів на відповідну кінцеву точку, результати будуть повертатися з кешу Redis замість звернення до бази даних

3.5 Навантажувальне тестування системи

3.5.1 Проведення навантажувального тестування

Як було визначено в попередньому розділі кваліфікаційної, оцінити ефективність та масштабованість системи можна за результатами тестування продуктивності, яке включає в себе декілька підвидів. Зокрема в даному випадку буде проводитися навантажувальне тестування. Робочі навантаження під час тестування буде розділено на три фази P0, P1, P2 [27]. P0 – це фаза прогріву, коли постійна кількість віртуальних користувачів в секунду (UPS) надсилає запити до тестованої системи протягом 60 секунд. У P1 навантаження лінійно збільшується з n віртуальних користувачів на секунду до m віртуальних користувачів на секунду протягом 60 секунд. Після фази масштабування починається фаза сталого навантаження (P2), і навантаження залишається постійним на визначеному рівні віртуальних користувачів на секунду протягом 180 секунд. Це робиться для того, щоб побачити, чи змінює система свою поведінку під навантаженням протягом більш тривалого періоду часу. Робочі навантаження, використані в цьому дослідженні, вказані в Таблиці 3.5.

Для генерації навантаження було використано інструмент Artillery описаний у попередньому розділі. Він використовується у більшості досліджень при тестуванні хмарних сервісів. Ще однією суттєвою перевагою при виборі інструментарію, стало те, що Artillery дозволяє записувати результати тестування JSON та на його основі генерувати репорт у вигляді HTML з візуалізацією результатів [11].

Таблиця 3.5 – Робочі навантаження використані при тестуванні

Навантаження	P0	P1	P2
1	1 UPS	1-5 UPS	5 UPS
2	2 UPS	2-10 UPS	10 UPS
3	10 UPS	10-50 UPS	50 UPS

Варто відзначити, що на момент проведення тестування Artillery відкрив доступ до закритої платформи Artillery Cloud, що дозволяє записувати результати тестування безпосередньо на саму хмарну платформу.

Сценарій для тест кейсів був згенерований таким чином, щоб максимально відтворити реальну поведінку користувача в мобільному застосунку.

Перший запит імітує верифікацію користувача в системі, шляхом валідації переданого токену, отриманого в наслідок Firebase Auth автентикації. Другий запит повертає список топ 20 найпопулярніших музичних треків. Третій запит список жанрів. Другий та третій запити виконуються одночасно як це відбувається в застосунку. Додатково запит на отримання списку жанрів у другому кейсі відмінний від інших двох, оскільки жанри тут є закешовані та отримуються безпосередньо з БД Redis замість основної бази даних. Четвертий запит імітує перехід в категорію жанрів та перегляд музичних треків. П’ятий запит імітує прослуховування користувачем певної музичної композиції та відправки асинхронного запиту з використанням черг для збільшення кількості прослуховувань музичної композиції. Шостий запит передбачає, що користувач додає прослухану композицію у список улюблених. Сьомий – перехід до улюблених треків. Восьмий – додавання треку до плейлиста. Конфігураційний файл сценарію тестування буде представлено у лістингу Б.2 додатку Б.

При аналізі результатів тестування увага зверталася на наступні метрики: медіанний час відгуку на запити, 95-квантиль, 99-квантиль, статус коди які були повернуті під час тестування, кількість паралельних екземплярів лямбда функції створених для обробки навантаження та кількість помилок. При цьому умову для таких метрик як 95-квантиль та 99-квантиль було встановлено у відповідності <800 мс та <1000 мс. Що означає, що в середня відповідь на запити 95%

користувачів повинна бути меншою за 800 мс (в продуктовому середовищі значення зазвичай беруться <300-400 мс) та для 99% меншою за 1000мс.

Визначившись з навантаженням, сценарієм та метриками для аналізу, та написавши код, можна переходити безпосередньо до самого тестування.

Запуск тестового сценарію відбувається за допомогою npm залежності artillery [11], яка може бути інстальована за допомогою таких пакетних менеджерів як npm чи yarn. При цьому в аргументах може передаватися лише назва сценарію, тоді результати будуть виводитися безпосередньо в термінал, з якого було виконано дану команду. Для додаткової генерації JSON файлу з результатами та запису на хмарну платформу передаються такі аргументи як record, key та output:

```
artillery run <назва сценарію для запуску>.yml
              --record
              --key <ключ отриманий на хмарній платформі>
              --output <назва файлу куди буде записано
              результати>.json
```

В результаті виконання першого сценарію з застосуванням параметрів першого експериментального навантаження, отримано наступні результати (рисунок 3.6):

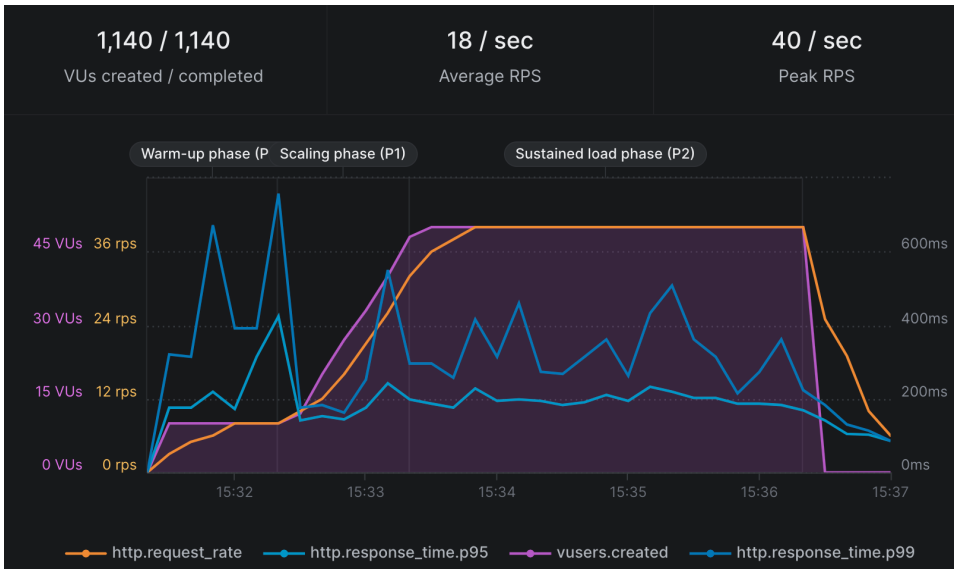

Рисунок 3.6 – Результати виконання першого сценарію

В даному випадку система показала доволі хороші результати, протягом виконання сценарію було виконано 9120 запитів, створено 1140 незалежних користувачів, які генерували запити до системи, при цьому кожен користувач отримав успішну відповідь на свої запити. Середня кількість запитів за секунду становила 18, а при піковому навантаженні, фазі P2 – 40. При цьому для обробки даного навантаження кількість екземплярів лямбда функцій лінійно зростала і в піку навантаження досягала 10 (рисунок 3.7), при цьому ніяких обмежень з боку платформи не спостерігається.

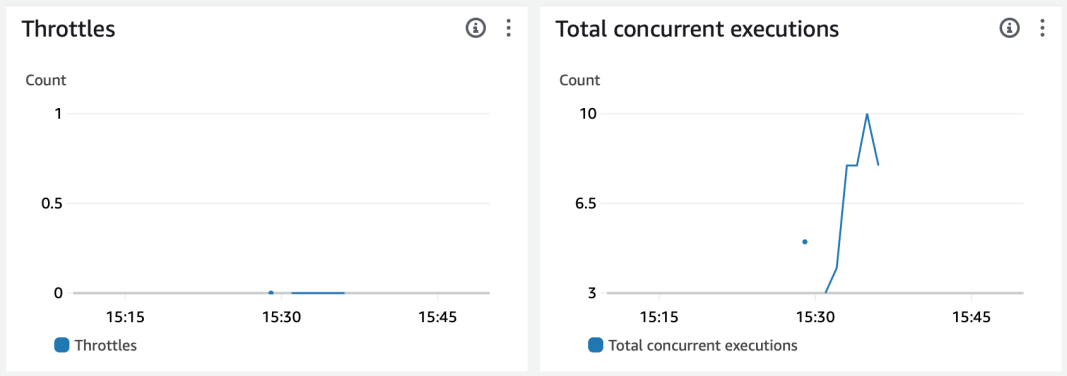

Рисунок 3.7 – Кількість обмежень та паралельних лямбда екземплярів застосованих при виконанні першого сценарію

Медіанна відповідь на запит становить 101,5 мілісекунди, p95 – 194,4, а p99 –383,3. При цьому максимальна відповідь на запити у менше ніж 1% випадків становить 4397 мс, що є результатом холодних стартів лямбда функцій.

В результаті виконання другого сценарію з застосуванням параметрів другого експериментального навантаження, отримано наступні результати (рисунок 3.8).

В даному випадку система показала також хороші результати, протягом виконання сценарію було виконано 18240 запитів, створено 2280 незалежних користувачів, які генерували запити до системи, при цьому 99.04% користувачів отримали успішні відповіді. Середня кількість запитів за секунду становила 64, а при піковому навантаженні, фазі P2 – 80. При цьому для обробки даного навантаження кількість екземплярів лямбда функцій збільшилася.

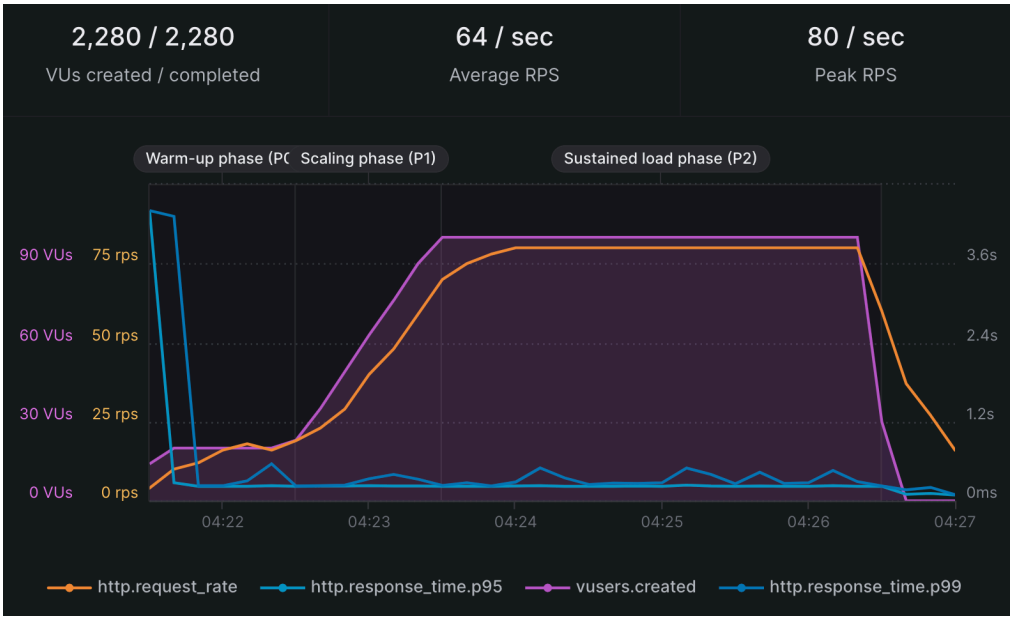

Рисунок 3.8 – Результати виконання другого сценарію

На початку можна бачити, що їх кількість рівна максимально можливому значенні виділеному платформою, що характеризується збільшенням кількості холодних стартів, далі при прогріванні функцій, протягом фази P1 кількість екземплярів зменшилася до 8, а при настанні фази P2 кількість знову зросла до пікової (10) (рисунок 3.9), і почала далі зростати, це призвело до обмеження з боку платформи, що характеризується появою 503 (Service Unavailable) помилок.

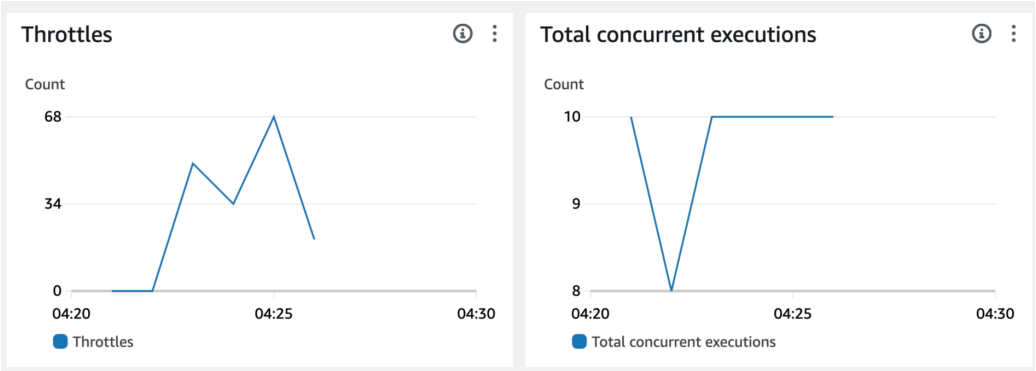

Рисунок 3.9 – Кількість обмежень та паралельних лямбда екземплярів застосованих при виконанні другого сценарію

Медіанна відповідь становить 111,4 мілісекунди, p95 – 219,2, при цьому можна спостерігати що p99 зменшився та становить 308. При цьому максимальна

відповідь на запити у менше ніж 1% випадків становить 4587 мс, що є результатом холодних стартів лямбда функцій.

Я вже було попередньо згадано, в другому сценарії отримання списку жанрів відбувалося з використанням кешування. Як можна бачити з рисунку 3,10, медіана та p95 зросли, проте p99 зменшився. Даний феномен пов'язаний з розміщенням екземпляру Redis в Орегоні, США, що підвищило латентність та в додаток обмеженням пропускної здатності мережі з боку постачальника хостингу.


```
{
  "plugins.metrics-by-endpoint.response_time./genres":{
    "min":61,
    "max":740,
    "count":1140,
    "mean":83.8,
    "p50":76,
    "median":76,
    "p75":82.3,
    "p90":89.1,
    "p95":100.5,
    "p99":295.9,
    "p999":487.9
  }
}
```


а)


```
{
  "plugins.metrics-by-endpoint.response_time./genres/cache":{
    "min":53,
    "max":591,
    "count":2280,
    "mean":218.8,
    "p50":219.2,
    "median":219.2,
    "p75":219.2,
    "p90":223.7,
    "p95":228.2,
    "p99":247.2,
    "p999":468.8
  }
}
```


б)

Рисунок 3.10 – Результати отримання списку жанрів, де а – без використання кешування та б з його використанням

В результаті виконання третього сценарію з застосуванням параметрів третього експериментального навантаження, отримано наступні результати (рисунок 3.11):

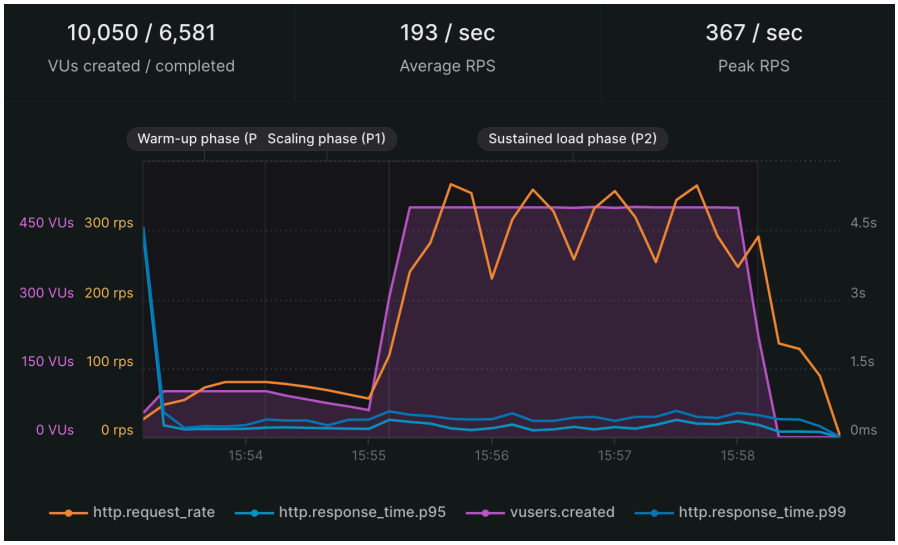

Рисунок 3.11 – Результати виконання третього сценарію

В даному випадку показники продуктивності продовжують задовільняти умови p95 < 800 мс та p99 < 1000 мс, але 48% користувачів отримують 503 помилки, через обмеження кількості паралельних екземплярів з боку платформи. Протягом виконання сценарію було виконано 66995 запитів, створено 10050 незалежних користувачів, які генерували запити до системи, при цьому тільки 51.69% користувачів отримали успішні відповіді. Середня кількість запитів за секунду становила 193, а при піковому навантаженні, фазі P2 – 367. Для обробки даного навантаження кількість екземплярів лямбда функцій становила 10 протягом всього часу (рисунок 3.12).

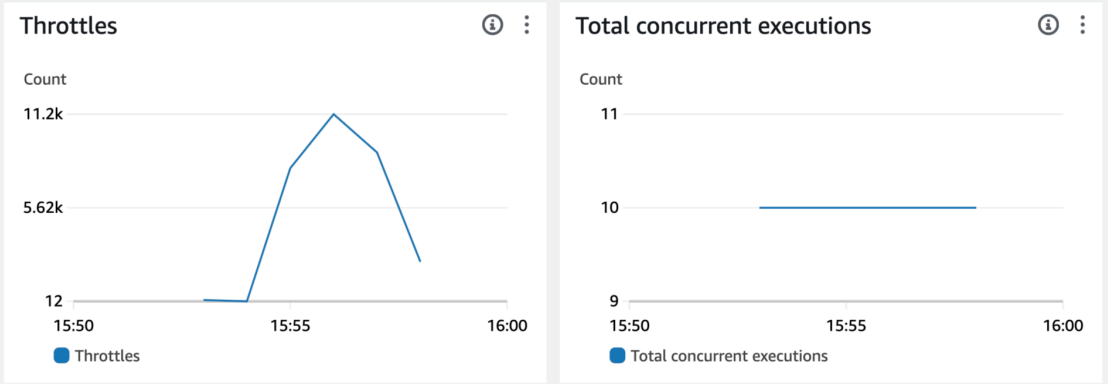

Рисунок 3.12 – Кількість обмежень та паралельних лямбда екземплярів застосованих при виконанні третього сценарію

Медіанну відповідь в даному випадку не було визначено, p95 – 223.7, при цьому можна спостерігати що p99 збільшився та становить 432,27. При цьому максимальна відповідь на запити у менше ніж 1% випадків становить 4724 мс.

3.5.2 Аналіз результатів тестування

На основі результатів тестування можна зробити висновок, що serverless функції, такі як лямбда-функції, є ефективним та масштабованим рішенням для обробки навантажень, проте накладають певні обмеження. Основним обмеженням у даному випадку виявилося обмеження кількості паралельних екземплярів функції, накладене провайдером хмарних обчислень. Зазвичай дане обмеження становить 1000 паралельних екземплярів, проте для нових

користувачів воно обмежене десятьма екземплярами. Це обмеження накладає обмеження на робочі навантаження, використані в тестових сценаріях. Збільшення навантаження, наприклад, до 200 користувачів за секунду, є недоцільним, оскільки більшість запитів повертатимуть код помилки 503 (Service Unavailable).

Щодо масштабованості запропонованого рішення, воно є дійсно масштабованим, адже навіть 10 екземплярів можуть справлятися зі звичайними навантаженнями та демонструвати хороші показники продуктивності системи (перший сценарій). Варто відзначити важливість використання черг у даній системі, оскільки черги дозволили уникнути обробки надлишкової кількості запитів, тим самим зменшивши необхідну кількість паралельних екземплярів функції. Ефективність даного рішення полягає у зниженні витрат на розгортання та обслуговування інфраструктури, адже при використанні традиційних хмарних обчислень потрібні додаткові витрати на підтримку та масштабування серверів, зокрема налаштування віртуальних приватних хмар (VPC), проксі-серверів, балансувальників навантаження тощо.

Хоча більшість джерел характеризують функції як сервіс (FaaS) як економічно ефективні рішення, варто зазначити, що економічна ефективність такого рішення безпосередньо залежить від інтенсивності надходження нових запитів за секунду [1]. Чим більше запитів за секунду, тим більшою є потреба у налаштуванні власних серверів, що може знизити економічну ефективність використання FaaS.

3.6 Реалізація CI/CD конвеєру для мобільного застосунку музичної платформи

Мобільний застосунок музичної платформи розроблено із застосуванням фреймворку React Native, який дозволяє використовувати спільну кодову базу написану на JavaScript та React, що в подальшому компілюється в нативний код відповідно до цільової платформи: Java для Android та Objective-C для iOS. Це

вимагає інтеграції в робочий процес GitHub Actions процедур випуску нових версій як для iOS, так і для Android.

Першим кроком у налаштуванні робочого процесу в GitHub Actions є створення нового робочого процесу. GitHub Actions пропонує низку попередньо підготовлених шаблонів для різних технологій, однак шаблон для React Native відсутній.

Далі необхідно визначити сценарій, за яким даний робочий процес буде запускатися. Зазвичай робочі процеси запускаються автоматично при надходженні змін до певної гілки (branch) для версії для тестувальників та вручну у випадку продуктової версії. У даному випадку робочий процес для обох середовищ буде запускатися вручну з можливістю запуску з будь-якої гілки. (рисунок 3.13)

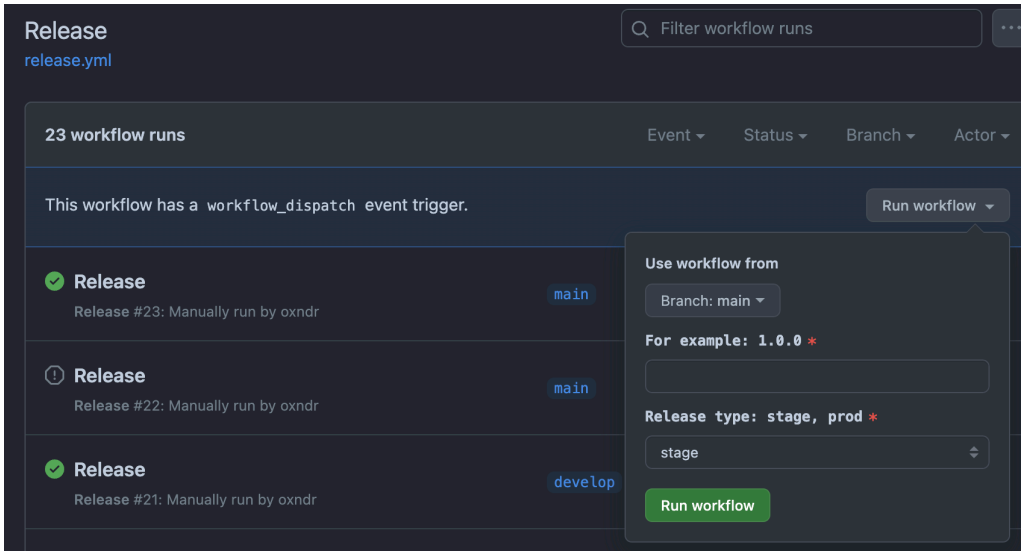

Рисунок 3.13 – Діалогове вікно запуску робочого процесу

Наступним кроком є визначення змінних середовища для уникнення повторюваності коду (рисунок 3.14). Чутливі змінні, такі як секретні ключі, рекомендовано безпечно зберігати у секретах репозиторію, які надійно шифруються та розшифровуються лише під час виконання робочого процесу, при цьому не відображаючись під час його виконання.

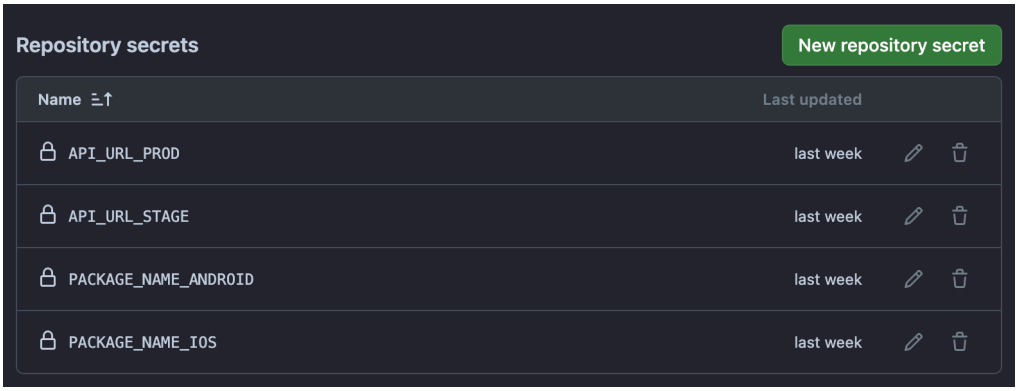

Рисунок 3.14 – Секрети репозиторію

Далі необхідно визначити перелік завдань, які будуть виконуватися під час робочого процесу. У даному робочому процесі визначено такі завдання:

- lint – перевірка коду на наявність стилістичних помилок.
- build-android – збірка нової версії застосунку для операційної системи Android (кешування Gradle файлів збірки для зменшення часу повторного виконання; встановлення залежностей проєкту; встановлення fastlane [21] плагінів; встановлення нової версії збірки, вказаної у діалоговому вікні, та власне збірка застосунку. Після успішного виконання етапу збірки, файл збірки завантажується до артефактів для подальшого прикріплення до ресурсів релізу. На завершальному етапі вихідний код змін відправляється до тимчасової гілки).
- build-ios – збірка нової версії застосунку для операційної системи iOS (встановлення залежностей проєкту; встановлення fastlane залежностей; встановлення нової версії збірки, вказаної у діалоговому вікні; додавання приміток до збірки та надсилання кодових змін в тимчасову гілку).
- release – створення нового GitHub релізу (злиття змін з тимчасових гілок у main гілку, якщо це продуктова збірка чи у develop якщо це версія для тестувальників; завантаження артефакту збірки Android; створення нового тегу та збірки).

Процес виконання завдань відбувається у визначеній послідовності: спершу виконується lint. Після успішного завершення цього завдання, паралельно розпочинаються build-android та build-ios. Після успішного виконання попередніх завдань виконується release.

Конфігураційний файл робочого процесу представлено у лістингу Б.3 додатку Б.

Результати успішного виконання робочого процесу можна побачити на рисунку 3.15 нижче.

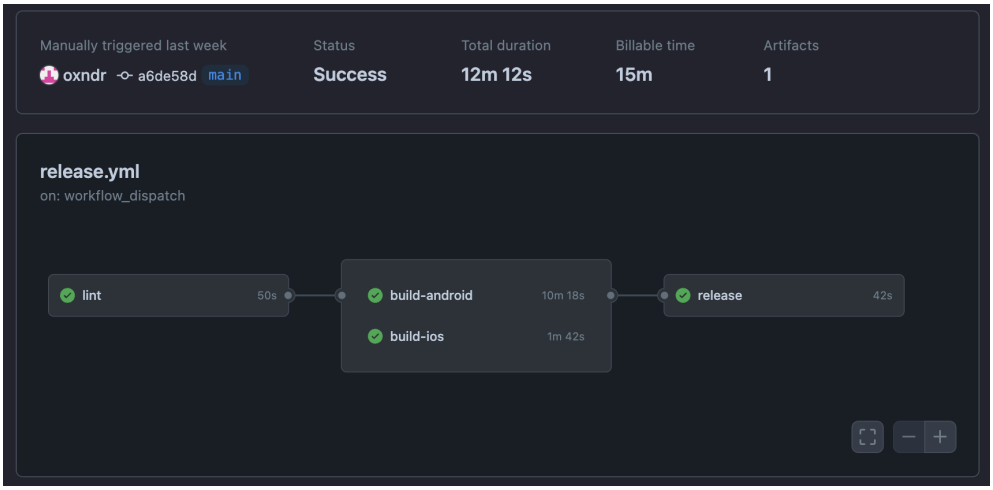

Рисунок 3.15 – Успішне виконання робочого процесу

При цьому серед релізів проєкту можна спостерігати нову версію (рисунок 3.16).

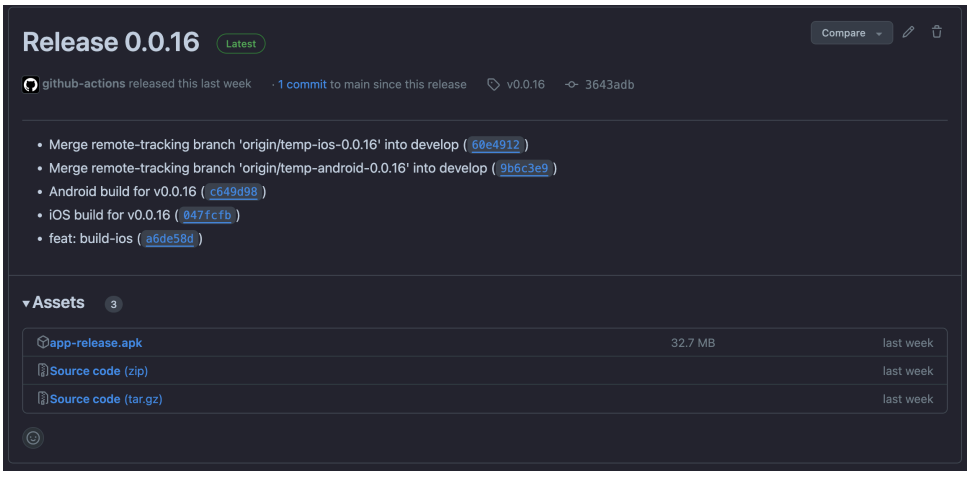

Рисунок 3.16 – Новий реліз проєкту

У розділі активів релізу прикріплюється файл збірки для Android-платформи, який доступний для завантаження. Цей файл генерується у форматі .apk під час випуску версії для тестувальників та у форматі .aab у випадку створення продуктової версії застосунку.

Для автоматизації процесу збірки для платформи iOS необхідно мати обліковий запис розробника в AppStore та використовувати середовище Xcode Cloud. Така вимога зумовлена політикою безпеки компанії Apple, яка забороняє встановлення сторонніх версій застосунків на своїх пристроях.

Як можна бачити з рисунку 3.15 час випуску нової версії застосунку зайняв 12 хв для обох платформ, що є значно швидше а ніж виконувати дані процеси вручну. До прикладу для виконання того ж самого сценарію в ручну для операційної системи Android необхідно в застосунку Android Studio виставити нову версію вручну, збільшити код версії та запустити процес збірки. Тривалість процесу збірки може варіюватися в залежності від наявності кешованих даних, доступності оперативної пам’яті та інших ресурсів локальної машини, після успішного виконання процесу збірки небхідно завантажити зміни до гілки репозиторію, створити новий реліз на GitHub, описати зміни релізу та прикріпити файл збірки до активів. При цьому у разі нехтування збільшення версії збірки чи її коду процес доведеться повторювати, через те, що версія коду застосунку повинна завжди бути більшою а ніж попередня для завантаження в GooglePlay.

3.7 Висновок до третього розділу

В третьому розділі кваліфікаційної роботи впроваджено процеси оптимізації на прикладі застосунку музичної платформи. Було здійснено перехід від монолітної до безсерверної архітектури розгортання з використанням хмарних сервісів AWS. Навантажувальне тестування продемонструвало ефективність та масштабованість запропонованого рішення, однак виявило певні обмеження, пов'язані з кількістю паралельних екземплярів безсерверних функцій з боку провайдера. Крім того, було реалізовано CI/CD конвеєр для мобільного застосунку музичної платформи з використанням GitHub Actions, що забезпечує автоматизацію процесів збірки та розгортання для платформ Android та iOS. Загалом, впровадження безсерверної архітектури та CI/CD конвеєру дозволило підвищити загальну ефективність та масштабованість системи.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

Четвертий розділ кваліфікаційної роботи присвячений питанням охорони праці та безпеки в надзвичайних ситуаціях. В даному розділі розглядаються фізіологічні та психологічні особливості розумової праці програмістів, вимоги до режимів праці та відпочинку при роботі з комп'ютерами. Також значна увага приділяється питанню підвищення стійкості роботи підприємств приладобудівної галузі у воєнний час, що є критично важливим для забезпечення обороноздатності держави.

4.1 Праця та її фізіологічні і психологічні особливості

Розробка full-stack застосунків вимагає як фізичних, так і розумових зусиль від розробників. Цей процес включає написання коду, вирішення проблем, аналіз вимог та оптимізацію рішень, що потребує значного інтелектуального навантаження.

На відміну від фізичної праці, де основне навантаження припадає на м'язи, під час розумової роботи над розробкою програмних застосунків активно задіяні такі психічні функції, як аналітичне та синтетичне мислення, обробка інформації, увага та пам'ять. Хоча енергетичні витрати можуть бути меншими, ніж при фізичній праці, потреба мозку в енергії підвищується і становить 15-20% від загального об'єму енергії, яка витрачається в організмі. При цьому вживання кисню 100 г кори головного мозку в 5 разів більше, ніж скелетними м'язами тієї ж ваги при максимальному фізичному навантаженні [3].

- При читанні вголос витрати енергії підвищуються на 48%;
- при публічному виступі – на 94%;
- при роботі операторів обчислювальних машин – на 60 – 100%.

Розробники часто зосереджені протягом тривалого часу, в середньому у 5 – 10 разів вище, ніж при фізичній праці, що вимагає високого ступеня концентрації уваги. Ця розумова напруга може призвести втоми та перевтоми, що може негативно вплинути на продуктивність та ефективність роботи.

При розумовій праці також погіршується робота органів зору, зокрема [3]:

- стійкість ясного бачення;
- гострота зору;
- адаптаційна можливість ока.

Під час розробки програмних застосунків також спостерігаються зміни у вегетативних функціях організму, таких як [3]:

- підвищення кров'яного тиску;
- зміни в електрокардіограмі;
- вентиляції легень і споживанні кисню;
- підвищення температури тіла.

Після завершення інтенсивної розумової роботи відчуття втоми може зберігатися довше, ніж після фізичної праці [3].

Крім того, розробники часто продовжують обробляти інформацію та вирішувати завдання навіть під час відпочинку, що підтверджує теорію про неперервну переробку інформації мозком. Це може призвести до зниження продуктивності та ефективності, якщо не дотримуватися належного балансу між роботою та відпочинком.

4.2 Вимоги до режимів праці і відпочинку при роботі з ПК

При організації праці за комп'ютером, для збереження здоров'я, запобігання професійним захворюванням та підтримки продуктивної працездатності працівників, необхідно передбачати регламентовані перерви для відпочинку протягом робочої зміни. Ці внутрішньо змінні режими праці та відпочинку включають додаткові нетривалі перерви в періоди, що передують появі ознак втоми та зниження працездатності.

Впродовж робочої зміни мають бути передбачені [3]:

- перерви для відпочинку та прийому їжі (обідні перерви);
- перерви для відпочинку та особистих потреб (згідно з трудовими нормами);

- додаткові перерви для окремих професійних груп з урахуванням особливостей їхньої діяльності.

Розробники програмного забезпечення виконують інтенсивну розумову творчу працю з високою концентрацією уваги, напруженням зору, нервово-емоційним навантаженням, вимушеною робочою позою та загальною гіподинамією. Їхня робота характеризується постійним активним діалогом з комп'ютером у вільному темпі з періодичним пошуком помилок в умовах дефіциту часу. Для розробників програмних застосунків при 8-годинній робочій зміні слід призначати регламентовану перерву для відпочинку тривалістю 15 хвилин через кожну годину роботи за екраном комп'ютера.

Оператори, які працюють з інформацією, отриманою з екрану монітора чи введеною в комп'ютер, виконують роботу з напруженням зору, невеликими фізичними зусиллями та нервовим навантаженням середнього ступеня у вільному темпі. Для операторів слід призначати регламентовані перерви для відпочинку тривалістю 15 хвилин через кожні дві години роботи з комп'ютером.

Оператори комп'ютерного набору виконують одноманітну роботу з документацією та клавіатурою, з нечастими короткими переключеннями погляду на екран. Їхня праця характеризується як фізична з підвищеним навантаженням на кисті рук, напруженням зору та нервово-емоційним навантаженням на тлі загальної гіподинамії. Для операторів комп'ютерного набору слід призначати регламентовані перерви для відпочинку тривалістю 10 хвилин після кожної години роботи за екраном.

У всіх випадках, коли виробничі обставини не дозволяють дотримуватись регламентованих перерв, безперервна робота з комп'ютером не повинна перевищувати 4 години [3].

При 12-годинній робочій зміні регламентовані перерви повинні встановлюватися протягом перших 8 годин аналогічно до 8-годинної зміни, а протягом останніх 4 годин, незалежно від виду діяльності, через кожну годину тривалістю 15 хвилин.

Для зниження нервово-емоційного навантаження, втоми зору, поліпшення мозкового кровообігу, подолання наслідків гіподинамії та запобігання втомі,

деякі перерви доцільно використовувати для виконання комплексу вправ відповідно до санітарних норм.

Психофізіологічне розвантаження. Під час сеансів психофізіологічного розвантаження рекомендується використовувати елементи аутогенного тренування, засновані на прийомах психічної саморегуляції та фізичних вправах із самонавіюванням. Головний акцент робиться на набутті навичок м'язового розслаблення. Рекомендований сеанс, який проводиться в спеціальній кімнаті, складається з трьох періодів: абстрагування працівників від робочого середовища, заспокоєння та активізації.

Також можуть проводитись сеанси психологічного розвантаження через індивідуальні навушники за єдиною програмою протягом двох 5-хвилинних періодів: 1) повне розслаблення; 2) активізація працездатності. при потребі, під музику можуть додаватись фрази навіювання відпочинку, гарного самопочуття та бадьорості [3].

Після сеансів психофізіологічного розвантаження у працівників зменшується відчуття втоми, з'являються бадьорість та гарний настрій, що загалом покращує їхній стан.

4.3 Підвищення стійкості роботи підприємств приладобудівної галузі у воєнний час

Здатність країни захищати свої кордони та забезпечувати життєдіяльність населення значною мірою залежить від стійкої роботи підприємств, особливо під час воєнного часу. Руйнування промислових об'єктів, значні втрати серед працівників та порушення виробничих зв'язків можуть призвести до різкого скорочення випуску необхідної продукції, що вплине на боєздатність збройних сил та життєдіяльність держави [5]. Тому підвищення стійкості роботи підприємств приладобудівної галузі у воєнний час є надзвичайно важливим завданням.

Приладобудівна галузь відіграє ключову роль у забезпеченні обороноздатності держави, оскільки виробляє різноманітне високотехнологічне

обладнання та прилади військового та цивільного призначення. Підприємства цієї галузі повинні бути максимально захищеними від руйнівного впливу надзвичайних ситуацій воєнного часу, щоб зберегти свій виробничий потенціал.

Під стійкістю роботи підприємства приладобудівної галузі розуміють його здатність за умов воєнного часу та можливих надзвичайних ситуацій виробляти необхідну продукцію в запланованих обсягах та асортименті, а при одержанні слабких чи середніх руйнувань – відновлювати своє виробництво в мінімальні терміни.

На стійкість роботи приладобудівних підприємств впливають такі фактори:

- Надійність захисту працівників від дії вражаючих факторів надзвичайних ситуацій.
- Здатність інженерно-технічного комплексу підприємства протистояти цим вражаючим факторам.
- Захищеність підприємства від дії вторинних вражаючих факторів.
- Надійність систем постачання підприємства всім необхідним для виробництва.
- Стійкість систем управління виробництвом та цивільною обороною.
- Готовність підприємства до ведення рятувальних та відновлювальних робіт.

Для підвищення стійкості роботи приладобудівних підприємств у воєнний час необхідно:

- Забезпечити надійний захист працівників від вражаючих факторів.
- Захистити виробничі приміщення, будівлі та споруди від цих факторів.
- Підвищити надійність та оперативність управління виробництвом і цивільною обороною.
- Забезпечити стійке постачання підприємства електроенергією, газом, водою, сировиною.

Ці вимоги реалізуються під час планування та розбудови нових виробничих об'єктів, реконструкції існуючих, а також у разі загрози виникнення надзвичайної ситуації.

Для об'єктивної оцінки стійкості роботи підприємства у воєнний час та розробки ефективних заходів з її підвищення проводяться ґрунтовні дослідження, які передбачають:

- Всебічне вивчення умов, які можуть скластися на підприємстві внаслідок надзвичайних ситуацій воєнного характеру.
- Оцінку впливу цих умов на подальшу виробничу діяльність підприємства з урахуванням специфіки технологічних процесів, обладнання, видів продукції.
- Виявлення вразливих місць у роботі підприємства, можливих "вузьких місць", порушення яких може повністю зупинити виробництво.
- Розробку обґрунтованих пропозицій та рекомендацій щодо підвищення стійкості роботи підприємства.

Дослідження проводяться в мирний час силами власних інженерно-технічних фахівців та спеціалістів цивільної оборони без відриву від виробництва. Створюються робочі групи за різними напрямками: енергетика, технологічні процеси, постачання, цивільна оборона тощо. Кожна група оцінює стійкість відповідних елементів виробничого комплексу підприємства та проводить необхідні розрахунки з урахуванням можливих уражаючих факторів надзвичайних ситуацій [5].

За результатами ґрунтовного дослідження готується загальний план конкретних заходів з підвищення стійкості роботи приладобудівного підприємства у воєнний час. Ці заходи плануються на мирний час (переважно трудомісткі та капіталовкладні) та на період загрози виникнення надзвичайних ситуацій (термінові роботи, які швидко підвищать захищеність підприємства).

Для належного підвищення стійкості роботи підприємств приладобудівної галузі у воєнний час можуть бути передбачені такі заходи:

- Будівництво захисних споруд цивільної оборони для працівників (сховищ, протирадіаційних укриттів).
- Посилення захисту виробничих приміщень, складів, допоміжних будівель від можливих руйнувань і пожеж. Облаштування захисних конструкцій.

- Підвищення живучості інженерних комунікацій, створення резервних джерел енергопостачання, водопостачання, каналізації.
- Будівництво захищених сховищ або споруд для зберігання запасів сировини, матеріалів, напівфабрикатів, готової продукції.
- Спорудження обвідних комунікацій для безперебійного постачання підприємства в умовах руйнувань.
- Створення необхідних резервних запасів сировини, палива, інструментів, засобів індивідуального захисту.
- Навчання та спеціальна підготовка персоналу до дій у надзвичайних ситуаціях воєнного часу.
- Удосконалення системи оперативного управління виробництвом, налагодження надійного зв'язку на випадок

Комплекс заходів визначається з урахуванням важливості підприємства, його місцезнаходження, виробничих особливостей та економічної доцільності [5]. Своєчасна реалізація цих заходів дозволить забезпечити стійку роботу приладобудівних підприємств у воєнний час.

4.4 Висновок до четвертого розділу

У четвертому розділі кваліфікаційної висвітлено наступні ключові питання: фізіологічні та психологічні аспекти праці, вимоги до режимів праці та відпочинку при роботі з комп'ютерами, заходи з підвищення стійкості роботи підприємств приладобудівної галузі у воєнний час. Були наведені рекомендації щодо організації робочого процесу та відпочинку для збереження здоров'я та продуктивності праці розробників програмного забезпечення. Також детально висвітлені заходи з підвищення захищеності приладобудівних підприємств від руйнівного впливу надзвичайних ситуацій воєнного часу для забезпечення їх стійкої роботи та виробництва необхідної продукції в умовах воєнного стану.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи освітнього рівня «Магістр» досліджено сучасні концепції та методології оптимізації розробки full-stack застосунків що сприяють підвищенню характеристик ефективності та масштабованості, зокрема, такі як мікросервісна архітектура, кешування, асинхронна обробка, горизонтальне масштабування, хмарні та безсерверні обчислення, а також DevOps практики. Проаналізовано моделі безсерверних обчислень, концепцію CI/CD конвеєрів та методологію тестування продуктивності. На прикладі музичної платформи продемонстровано впровадження процесів оптимізації та розгортання архітектури з використанням безсерверних функцій. Додатково проведено оцінку ефективності та масштабованості системи на основі результатів навантажувального тестування.

Виконано огляд концепції full-stack розробки, аналіз ефективності та масштабованості full-stack застосунків, а також розглянуто методи оптимізації розробки для підвищення ефективності та масштабованості, такі як мікросервісна архітектура, кешування, асинхронна обробка, горизонтальне масштабування, хмарні та безсерверні обчислення, DevOps.

Здійснено дослідження моделі безсерверних обчислень, концепції CI/CD конвеєрів та методології тестування продуктивності. Розглянуто безсерверну архітектуру, її переваги та недоліки, а також FaaS платформи. Проаналізовано інструменти тестування продуктивності, ключові показники ефективного CI/CD конвеєру та платформи для їх розгортання.

Проведено аналіз вимог до експериментальної системи на прикладі музичної платформи, продемонстровано принцип декомпозиції монолітної архітектури на безсерверні функції та спроектовано архітектуру музичної платформи з впровадженням процесів оптимізації. Також описано розгортання архітектури, проведено навантажувальне тестування системи з аналізом результатів та реалізовано CI/CD конвеєр для мобільного застосунку музичної платформи.

Розглянуто питання охорони праці та безпеки в надзвичайних ситуаціях, зокрема фізіологічні та психологічні аспекти праці, вимоги до режимів праці та відпочинку під час роботи з комп'ютерними системами та підвищення стійкості функціонування підприємств приладобудівної галузі в умовах воєнного часу.

ПЕРЕЛІК ДЖЕРЕЛ

1. Болож О. Розробка веб-сайту “Artise” засобами React, Node.js та MongoDB [Електронний ресурс] / Олександр Болож. – 2022. – Режим доступу до ресурсу: https://elartu.tntu.edu.ua/bitstream/lib/38366/1/2022_KRB_SN-41_Bolozh_O_V.pdf.

2. Волович В. Задача проєктування програмної архітектури в процесах забезпечення якості [Електронний ресурс] / В. Волович, Б. Береженко, І. Боднарчук // ТНТУ. – 2022. – Режим доступу до ресурсу: https://elartu.tntu.edu.ua/bitstream/lib/40339/2/XNTK_2022_Volovych_V-The_problem_of_software_104-106.pdf.

3. Гандзюк М. П. Основи охорони праці / М. П. Гандзюк, Є. П. Желібо, М. О. Халімовський. – Київ, 2005. – 407 с.

4. Проектування архітектури програмних систем в проектах з гнучкими методами управління [Електронний ресурс] / І.Боднарчук, О. Харченко, Б. Хоміцький, Г. Шимчук // ТНТУ. – 2019. – Режим доступу до ресурсу: https://elartu.tntu.edu.ua/bitstream/lib/28122/2/XXI_NK_2019_Bodnarchuk_I-Software_systems_architecture_46-48.pdf.

5. Рогозін А. С. Підвищення стійкості роботи об’єктів господарювання в надзвичайних ситуаціях мирного та воєнного часів [Електронний ресурс] / А. С. Рогозін. – 2016. – Режим доступу до ресурсу: https://studfile.net/preview/5197475.

6. Amazon API Gateway pricing [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/api-gateway/pricing/.

7. Amazon S3 [Електронний ресурс] // AWS – Режим доступу до ресурсу: https://aws.amazon.com/s3/.

8. Amazon SQS pricing [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/sqs/pricing/.

9. Apache JMeter [Електронний ресурс] – Режим доступу до ресурсу: https://jmeter.apache.org/.

10. Arokia Paul Rajan R. Serverless Architecture - A Revolution in Cloud Computing [Електронний ресурс] / Arokia Paul Rajan // IEEE. – 2018. – Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/8939081

11. Artillery Docs [Електронний ресурс] – Режим доступу до ресурсу: https://www.artillery.io/.

12. Asynchronous Processing [Електронний ресурс] // Reintech – Режим доступу до ресурсу: https://reintech.io/term/asynchronous-processing.

13. AWS Lambda Pricing [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/lambda/pricing/.

14. Choudary A. Performance Testing Life Cycle: A Comprehensive Guide To The Testing Phases [Електронний ресурс] / Archana Choudary. – 2019. – Режим доступу до ресурсу: https://medium.com/edureka/performance-testing-life-cycle-d4242d39a5aa.

15. Content delivery networks (CDNs) for Content-Driven Web Apps [Електронний ресурс] // Google – Режим доступу до ресурсу: https://developers.google.com/solutions/content-driven/hosting/cdn.

16. Dalmia A. The New Era of Full Stack Development Intoduction of Cloud and It\'s Impacts [Електронний ресурс] / A. Dalmia, A. Chowdary – Режим доступу до ресурсу: https://www.researchgate.net/publication/340460348_The_New_Era_of_Full_Stack_Development_Intoduction_of_Cloud_and_It\'s_Impacts.

17. Deep M. D. Performance Testing: Methodology for Determining Scalability of Web Systems [Електронний ресурс] / Manishkumar Dave Deep. – 2024. – Режим доступу до ресурсу: https://www.researchgate.net/publication/377656310_Performance_Testing_Methodology_for_Determining_Scalability_of_Web_Systems.

18. DevOps [Електронний ресурс] // Wikiversity – Режим доступу до ресурсу: https://en.wikiversity.org/wiki/DevOps/

19. Economics of "Serverless" [Електронний ресурс] / [Á. Rodríguez, G. López, M. Evgeniev та ін.] // BBAV. – 2018. – Режим доступу до ресурсу: https://www.bbva.com/en/innovation/economics-of-serverless/.

20. Efficiency - Maximizing Software Performance [Електронний ресурс] // Reintech – Режим доступу до ресурсу: https://reintech.io/terms/category/efficiency-software-development/.

21. fastlane docs [Електронний ресурс] – Режим доступу до ресурсу: https://docs.fastlane.tools/.

22. Full Stack Development Explained [Електронний ресурс] – Режим доступу до ресурсу: https://www.mongodb.com/languages/full-stack-development.

23. Grafana k6 documentation [Електронний ресурс] – Режим доступу до ресурсу: https://grafana.com/docs/k6/latest/.

24. Hassan B. H. Survey on serverless computing [Електронний ресурс] / B. H. Hassan, A. B. Saman, I. S. Qusay. – 2021. – Режим доступу до ресурсу: https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-021-00253-7.

25. Kharchenko A. The method of quality management software [Електронний ресурс] / A. Kharchenko, I. Galay, V. Yatcyshyn // IEEE. – 2011. – Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/5960278.

26. Kharchenko A. The Survey of Global Software Design Processes [Електронний ресурс] / A. Kharchenko, I. Raichev, I. Bodnarchuk // IEEE. – 2021. – Режим доступу до ресурсу: https://ieeexplore.ieee.org/document/9772205.

27. Kuhlenkamp J. https://sci-hub.se/https://dl.acm.org/doi/10.1145/3341105.3373948 [Електронний ресурс] / J. Kuhlenkamp, S. Werner, M. Borges – Режим доступу до ресурсу: https://sci-hub.se/https://dl.acm.org/doi/10.1145/3341105.3373948.

28. Martin F. Microservices [Електронний ресурс] / F. Martin, L. James – Режим доступу до ресурсу: https://martinfowler.com/articles/microservices.html.

29. Migrating from Monolithic to Serverless: A FinTech Case Study [Електронний ресурс] // ResearchGate. – 2020. – Режим доступу до ресурсу: https://www.researchgate.net/publication/340681076_Migrating_from_Monolithic_to_Serverless_A_FinTech_Case_Study.

30. Milić M. Development of a Quality-Based Model for Software Architecture Optimization: A Case Study of Monolith and Microservice Architectures

[Електронний ресурс] / M. Milić, D. Makajić-Nikolić – Режим доступу до ресурсу: https://www.mdpi.com/2073-8994/14/9/1824.

31. NestJS Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/.

32. PaaS vs. IaaS vs. SaaS vs. CaaS: How are they different? [Електронний ресурс] // Google – Режим доступу до ресурсу: https://cloud.google.com/learn/paas-vs-iaas-vs-saas.

33. Performance Testing - Software Testing [Електронний ресурс] // GeeksforGeeks. – 2023. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/performance-testing-software-testing/.

34. Pogiatzis A. An Event-Driven Serverless ETL Pipeline on AWS [Електронний ресурс] / Antreas Pogiatzis. – 2021. – Режим доступу до ресурсу: https://www.mdpi.com/2076-3417/11/1/191.

35. Richardson C. Microservice Architecture pattern [Електронний ресурс] / Chris Richardson – Режим доступу до ресурсу: https://microservices.io/patterns/microservices.

36. Roberts M. What is Serverless? [Електронний ресурс] / M. Roberts, J. Chapin // O'Reilly Media. – 2017. – Режим доступу до ресурсу: https://www.symphonia.io/what-is-serverless.pdf.

37. Scalability [Електронний ресурс] // Gartner – Режим доступу до ресурсу: https://www.gartner.com/en/information-technology/glossary/scalability

38. Serverless computing [Електронний ресурс] // Wikipedia – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Serverless_computing.

39. Software Efficiency Matters [Електронний ресурс] // O'Reilly. – 2022. – Режим доступу до ресурсу: https://www.oreilly.com/library/view/efficient-go/9781098105709/ch01.html.

40. Stephen B. CI/CD pipelines explained: Everything you need to know [Електронний ресурс] / Bigelow Stephen. – 2021. – Режим доступу до ресурсу: https://www.techtarget.com/searchsoftwarequality/CI-CD-pipelines-explained-Everything-you-need-to-know.

41. The NIST Definition of Cloud Computing [Електронний ресурс] // NIST – Режим доступу до ресурсу: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf.

42. Trade-off optimal decision of the problem of software system architecture choice [Електронний ресурс] / A.Kharchenko, I. Halay, N. Zagorodna, I. Bodnarchuk // IEEE. – 2015. – Режим доступу до ресурсу: https://ieeexplore.ieee.org/abstract/document/7325465.

43. Understanding GitHub Actions [Електронний ресурс] // GitHub – Режим доступу до ресурсу: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions.

44. Wen, J., Chen, Z., Jin, X., Liu, X. Rise of the Planet of Serverless Computing: A Systematic Review [Електронний ресурс] / Wen. – 2022. – Режим доступу до ресурсу: https://arxiv.org/abs/2206.12275.

45. What is a CI/CD pipeline? [Електронний ресурс] // RedHat. – 2022. – Режим доступу до ресурсу: https://www.redhat.com/en/topics/devops/what-cicd-pipeline.

46. What is a Tech Stack and How Do They Work? [Електронний ресурс] – Режим доступу до ресурсу: https://www.mongodb.com/resources/basics/technology-stack.

47. What is Cloud Computing? [Електронний ресурс] // TechTarget – Режим доступу до ресурсу: https://www.techtarget.com/searchcloudcomputing/definition/cloud-computing.

48. What Is Full Stack Development? [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/what-is/full-stack-development

ДОДАТКИ

Додаток А

Тези конференцій

Міністерство освіти і науки України,
Тернопільський національний технічний університет
імені Івана Пулюя
Маріборський університет (Словенія)
Технічний університет в Кошице (Словаччина)
Каунаський технологічний університет (Литва)
Львівський національний університет
імені Івана Франка,
Гірничо-металургійна академія ім. Станіслава Сташиця (Польща)
Луцький національний технічний університет,
Чернівецький національний університет
імені Юрія Федьковича,
Вроцлавський економічний університет (Польща)
Університет технологій та економіки
імені Хелени Ходковської (Польща)
Донбаська державна машинобудівна академія

Студентське наукове товариство

VI МІЖНАРОДНА

студентська науково - технічна конференція

"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ.

АКТУАЛЬНІ ПИТАННЯ"

27-28 квітня 2023 р.

(збірник тез конференції)

Тернопіль 2023

VI Міжнародна студентська науково - технічна конференція
"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ. АКТУАЛЬНІ ПИТАННЯ"

УДК 004.07

Болож О. – ст. гр. СНнм-51
Тернопільський національний технічний університет імені Івана Пулюя

ОГЛЯД SERVERLESS АРХІТЕКТУРИ ТА ЇЇ ПЕРЕВАГИ

Науковий керівник: старший викладач Шимчук Г.

Bolozh O.
Ternopil Ivan Puluj National Technical University

OVERVIEW OF SERVERLESS ARCHITECTURE AND ITS ADVANTAGES

Supervisor: Senior Lecturer Shymchuk G.

Ключові слова: serverless, serverless architecture, BaaS, FaaS, cloud platform, AWS Lambda, Google Cloud Functions, DB, Microsoft Azure, back-end
Key words: serverless, serverless architecture, BaaS, FaaS, cloud platform, AWS Lambda, Google Cloud Functions, DB, Microsoft Azure, back-end

Serverless архітектура – це спосіб створення та запуску додатків, який зменшує потребу в управлінні ресурсами. Такий підхід дозволяє виконавцям запускати додатки без управління фізичними серверами.[1]. Це дозволяє хмарним провайдерам виконувати код, розподіляючи ресурси та масштабуючи інфраструктуру в залежності від потреб, які виникають. Таким прикладом може бути розміщена на певній хмарній інфраструктурі база даних, яка з часом розростається і потребує більше ресурсів, таких як оперативна пам'ять чи кількість ядер процесора. Тут потрібно особисто перевіряти метрики і вручну підбирати тарифи, а у разі використання serverless – маштабування проходить автоматично, безболісно і менш затратно.

Прикладами платформ що надають можливості впровадження безсерверної архітектури є AWS Lambda, Azure Functions, Google Cloud Functions тощо.[1]

Serverless архітектура функціонує наступним чином: сервери дозволяють користувачам спілкуватися з додатком і отримувати доступ до його бізнес-логіки, але управління серверами займає значний час і ресурси. Командам доводиться обслуговувати серверне обладнання, дбати про оновлення програмного забезпечення та безпеки, а також створювати резервні копії на випадок збоїв. Впроваджуючи безсерверну архітектуру, розробники можуть перекласти ці обов'язки на стороннього постачальника, що дозволить їм зосередитися на написанні коду додатків.[2]

Однією з найпопулярніших безсерверних архітектур є "Функція як послуга" (Function as a Service, FaaS), де розробники пишуть код своїх додатків як набір окремих функцій. Кожна функція виконує певне завдання, коли її запускає певна подія, наприклад, вхідний електронний лист або HTTP-запит. Після звичайних етапів тестування розробники розгортають свої функції разом з тригерами в акаунті хмарного провайдера. Коли функція викликається, хмарний провайдер або виконує її на працюючому сервері, або, якщо сервер наразі не працює, запускає новий сервер для виконання функції. Цей процес виконання абстрагований від розробників, які зосереджені на написанні та розгортанні коду програми.[2]

Ще однією послугою що використовується в Serverless є "Backend як послуга" (Backend as a Service) – такий архітектурний підхід позбавляє розробників від деяких завдань пов'язаних з серверною розробкою, таких як хостинг хмарного сховища та управління базами даних.

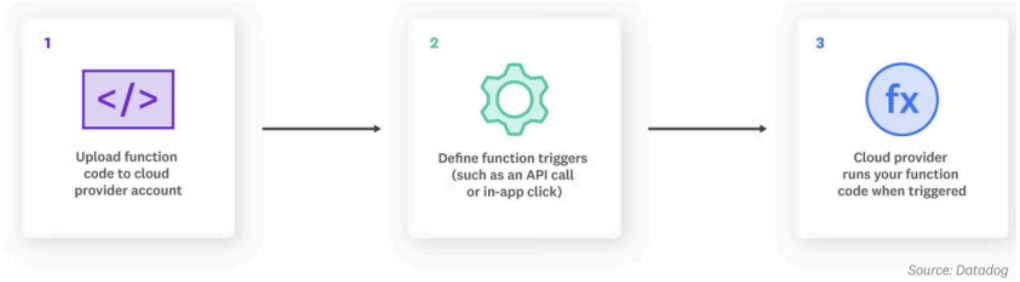

Рисунок 1 – Принцип функціонування FaaS

Основні переваги Serverless архітектури:

- нижча вартість, якщо врахувати зменшення потреби в персоналі DevOps для налаштування та підтримки інфраструктури з плином часу, а також оплату за використання, що означає, що періоди з низьким трафіком можуть коштувати $0 за годину;[3]
- швидкість розробки різко зростає, оскільки набагато простіше збирати рішення разом і розгортати їх у виробництво;[3]
- стабільність оскільки хмара керує всіма сервісами, ризик що щось піде не так – мінімальний або ж відсутній взагалі;
- готовність до різких навантажень, як вже було описано раніше Serverless маштабується згідно навантажень, які виникають.

Деякі сценарії в яких може бути використана безсерверна архітектура:

- реалізація асинхронної обробки повідомлень у додатках;
- обробка даних для забезпечення потужного машинного навчання;
- створювати додатки з високою затримкою в реальному часі, наприклад, мультимедійні додатки, для автоматичного розподілу пам'яті та складної обробки даних;
- обслуговування непередбачуваних робочих навантажень для швидкозмінних потреб розробки, додавання функцій на вимогу клієнтів та інших складних потреб у масштабуванні;
- динамічно змінювати розмір зображень або перекодовувати відео та спрощувати обробку мультимедійних даних на різних пристроях;
- для побудови спільної системи диспетчеризації доставки;
- для додатків на основі Інтернету речей (IoT) і розумних пристроїв;
- у прикладних модулях на основі сценаріїв трансляції відео в реальному часі;
- реалізація безперервної інтеграції (CI) та безперервної доставки (CD).

Література:

1. Solanki J. Serverless Architecture – What It Is? Benefits, Limitations & Use cases [Електронний ресурс] / Jignesh Solanki. – 2023. – Режим доступу до ресурсу: https://www.simform.com/blog/serverless-architecture-guide.
2. Serverless Architecture Overview [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://www.datadoghq.com/knowledge-center/serverless-architecture.
3. McCumskey G. A Guide to Serverless Architecture [Електронний ресурс] / Gareth McCumskey. – 2022. – Режим доступу до ресурсу: https://www.serverless.com/blog/serverless-architecture.

Міністерство освіти і науки України
Тернопільський національний технічний університет
імені Івана Пулюя
Маріборський університет (Словенія)
Технічний університет в Кошице (Словаччина)
Каунаський технологічний університет (Литва)
Львівський національний університет
імені Івана Франка,
Гірничо-металургійна академія ім. Станіслава Сташиця (Польща)
Луцький національний технічний університет,
Чернівецький національний університет
імені Юрія Федьковича,
Вроцлавський економічний університет (Польща)
Університет технологій та економіки
імені Хелени Ходковської (Польща)
Донбаська державна машинобудівна академія

Студентське наукове товариство

VII МІЖНАРОДНА

студентська науково - технічна конференція

"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ.

АКТУАЛЬНІ ПИТАННЯ"

25-26 квітня 2024 р.

(збірник тез конференції)

Тернопіль 2024

VII Міжнародна студентська науково - технічна конференція
"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ. АКТУАЛЬНІ ПИТАННЯ"

Секція: **Інформаційні технології**

УДК 621.326

Болож О. – ст. гр. СНнм-61
Тернопільський національний технічний університет імені Івана Пулюя

ХАРАКТЕРИСТИКИ ЕФЕКТИВНОСТІ ТА МАСШТАБОВАНОСТІ FULL-STACK ЗАСТОСУНКІВ

Науковий керівник: к.т.н., Боднарчук І.

Bolozh O.
Ternopil Ivan Pul'uj National Technical University

EFFICIENCY AND SCALABILITY CHARACTERISTICS OF FULL-STACK APPLICATIONS

Supervisor: Ph. D. Bodnarchuk I.

Ключові слова: full-stack, ефективність, масштабованість
Keywords: full-stack, efficiency, scalability

Ефективність є комплексною характеристикою, що визначає здатність програмного забезпечення виконувати необхідний функціонал з мінімальними витратами ресурсів і часу [1]. Вона залежить від багатьох чинників: вибору оптимальної архітектури, сучасних фреймворків і бібліотек, якості коду, можливостей масштабування та розширення функціоналу.

До ключових кількісних показників ефективності відносять: швидкість роботи системи та окремих сервісів, час відгуку на запити користувачів, пропускну здатність при високих навантаженнях, вартість розробки та підтримки, собівартість внесення змін та додавання нового функціоналу. Серед якісних показників – зручність використання з точки зору кінцевих користувачів, можливості інтеграції з іншими системами, відповідність сучасним стандартам і тенденціям розвитку галузі.

Ефективність full-stack застосунку необхідно оцінювати на всіх рівнях: окремих компонентів і сервісів, підсистем та модулів, загальної архітектури та інфраструктури, бізнес-процесів та всієї системи в цілому. Комплексний підхід дозволяє виявити слабкі місця на кожному рівні та ефективно спрямувати зусилля на оптимізацію.

Масштабованість характеризує здатність системи стабільно функціонувати при значному зростанні навантаження кількості користувачів, трафіку, обсягів даних [2]. Це надзвичайно важлива властивість сучасних високонавантажених застосунків. Масштабованість досягається гнучкою архітектурою, використанням хмарних обчислень та сервісів автоматичного масштабування, механізмами кешування та балансування навантаження, асинхронною та паралельною обробкою запитів.

Оцінити масштабованість можна за результатами навантажувального та стрес-тестування системи в умовах зростаючих навантажень. Аналізують такі показники: стабільність роботи сервісів, час відгуку на запити, відсоток помилок та відмов, використання ресурсів (процесор, пам'ять, мережа). Тестування дозволяє виявити "вузькі місця" архітектури та скоригувати її.

Отже, аналіз ефективності та масштабованості є важливим етапом оцінки якості та оптимізації full-stack застосунків. Він базується на кількісних та якісних метриках, вимірюваних на всіх рівнях системи під час функціонального та навантажувального тестування. Комплексний підхід дозволяє виявити недоліки архітектури та спрямувати зусилля на їх усунення.

Література

1. Efficiency - Maximizing Software Performance [Електронний ресурс] // Reintech – Режим доступу до ресурсу: https://reintech.io/terms/category/efficiency-software-development/.
2. Definition of Scalability [Електронний ресурс] // Gartner – Режим доступу до ресурсу: https://www.gartner.com/en/information-technology/glossary/scalability/.

УДК 621.326

Болож О. – ст. гр. СНнм-61
Тернопільський національний технічний університет імені Івана Пулюя

ЗАСТОСУВАННЯ ПРАКТИК DEVOPS В РОЗРОБЦІ FULL-STACK ЗАСТОСУНКІВ

Науковий керівник: к.т.н., Боднарчук І.

Bolozh O.
Ternopil Ivan Puluj National Technical University

APPLICATION OF DEVOPS PRACTICES IN FULL-STACK APPLICATION DEVELOPMENT

Supervisor: Ph. D. Bodnarchuk I.

Ключові слова: full-stack, DevOps, автоматизація
Keywords: full-stack, DevOps, automation

DevOps – це культура та практика інженерії програмного забезпечення, яка має на меті об’єднати розробку програмного забезпечення (Dev) та його експлуатацію (Ops). DevOps націлений на скорочення циклів розробки, збільшення частоти розгортання, підвищення надійності випусків у тісному зв’язку з бізнес-цілями. [1]

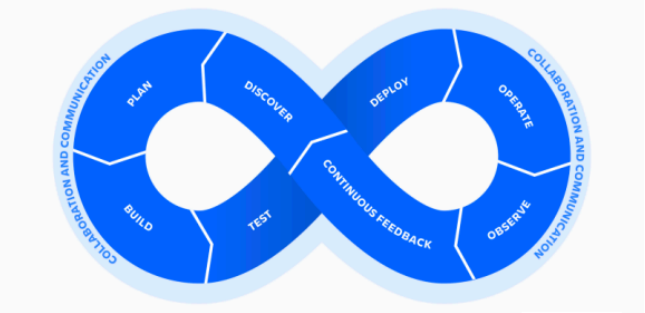

Рисунок 1 – Життєвий цикл DevOps (Atlassian)

DevOps включає в себе низку концепцій, серед яких: автоматизація, контейнеризація, моніторинг та CI/CD. Автоматизація сприяє мінімізації помилок, пов'язаних з людським фактором, та суттєвому скороченню часу розробки. Контейнеризація дозволяє ізолювати програмні рішення та їх залежності, забезпечуючи консистентність між середовищами розробки та виробництва. Моніторинг надає розуміння поведінки системи, забезпечуючи можливість ідентифікувати та локалізувати помилки на різних рівнях, від системних збоїв до помилок у бізнес-логіці. CI/CD дозволяє автоматизувати процеси збірки, тестування та розгортання.

У full-stack розробці DevOps застосовується на різних рівнях. На рівні фронтенду DevOps сприяє оптимізації та автоматизації процесів тестування інтерфейсу користувача, збирання та стиснення статичних ресурсів, забезпечуючи високу якість та продуктивність користувацького досвіду. На рівні бекенду DevOps охоплює автоматизацію збірки, контроль версій та моніторинг роботи серверів та баз даних. В контексті інфраструктури DevOps використовується для автоматизації розгортання та налаштування інфраструктурних компонентів, забезпечуючи високу доступність, стійкість та ефективність використання ресурсів.

Отже, впровадження практик DevOps у розробку full-stack застосунків сприяє скороченню часу до випуску на ринок, підвищенню якості розроблюваних застосунків та суттєвому покращенню процесів розробки загалом. Крім того, це зменшує ризики відмов у продуктовому середовищі та забезпечує масштабованість для подальшого розвитку бізнес процесів.

Література

1. DevOps [Електронний ресурс] // Wikiversity – Режим доступу до ресурсу: https://en.wikiversity.org/wiki/DevOps/
2. What is DevOps? [Електронний ресурс] // Atlassian – Режим доступу до ресурсу: https://www.atlassian.com/devops/

Додаток Б

Лістинги команд та конфігураційних файлів

Лістинг Б.1 – Вміст файлу serverless.ts для розгортання ресурсів на AWS

```
import type { Serverless } from "serverless/aws";
import ApiGatewaySqsRole from "./iam/ApiGatewaySqsRole";
import UpdateSongPlayCountApiSqsIntegration, {
  UpdateSongPlayCountApiSqsIntegrationRoute,
} from "./resources/integrations/UpdateSongPlayCountApiSqsIntegration";
import DeadLetterQueue, {
  DeadLetterQueueCustom,
} from "./resources/sqs/DeadLetterQueue";
import UpdateSongPlayCountQueue, {
  UpdateSongPlayCountQueueCustom,
} from "./resources/sqs/UpdateSongPlayCountQueue";

const serverlessConfiguration: Serverless = {
  app: "artise",
  service: "artise",
  useDotenv: true,
  package: { patterns: ["!**", "dist/*.*"] },
  plugins: ["serverless-offline", "serverless-dotenv-plugin"],
  provider: {
    name: "aws",
    runtime: "nodejs18.x",
    region: "eu-west-1",
    stage: '${opt:stage, "stage"}',
    environment: {
      STAGE: "${env:STAGE}",
    },
    apiName: "${self:app}-${self:provider.stage}",
    architecture: "arm64",
  },
```

```
        custom: {
          baseName: "${self:app}-${self:provider.stage}",
          ...UpdateSongPlayCountQueueCustom,
          ...DeadLetterQueueCustom,
        },
        functions: {
          main: {
            name: "${self:app}-${self:provider.stage}",
            handler: "dist/index.handler",
            memorySize: 256,
            events: [
              {
                httpApi: {
                  method: "*",
                  path: "*",
                },
              },
            ],
            timeout: 29,
          },
          updateSongPlayCount: {
            handler: "dist/index.updateSongPlayCountHandler",
            memorySize: 256,
            events: [
              {
                sqs: {
                  arn:
"${self:custom.UpdateSongPlayCountQueue.arn}",
                  batchSize: 200,
                  maximumBatchingWindow: 120,
                },
              },
            ],
            timeout: 180,
          },
        },
        resources: {
```

```
    Resources: {
      //* SQS
      UpdateSongPlayCountQueue,
      DeadLetterQueue,
      //* API Gateway -> SQS
      ApiGatewaySqsRole,
      UpdateSongPlayCountApiSqsIntegration,
      UpdateSongPlayCountApiSqsIntegrationRoute,
    },
  },
};

module.exports = serverlessConfiguration;
```

Лістинг Б.2 – Конфігураційний файл сценарію тестування

```
config:
  target: <API URL>
  phases:
    - duration: 60
      arrivalRate: 2
      name: Warm-up phase (P0)
    - duration: 60
      arrivalRate: 2
      rampTo: 10
      name: Scaling phase (P1)
    - duration: 180
      arrivalRate: 10
      name: Sustained load phase (P2)
  plugins:
    ensure: {}
    apdex: {}
    metrics-by-endpoint: {}
  apdex:
    threshold: 100
  ensure:
    maxErrorRate: 1
    thresholds:
```

```
        - http.response_time.p99: 1000
        - http.response_time.p95: 800
    http:
      defaults:
        headers:
          Authorization: "Bearer <Token>"
    scenarios:
      - flow:/
          - post:
              url: /auth/verify # login
          - think: 1
          - get:
              url: /songs/top-n?n=20 # get top songs
          - get:
              url: /genres/cache # get genres
          - think: 3
          - get:
              url:     /songs/search?genreId=c5ef805f-1048-11ef-
828f-02ef1960dd59&take=20&skip=0 # get 20 songs by selected genre
          - think: 8
          - post:
              url: /songs/play-count # listen to the song
              json:
                songId: 000c5d3f-40e9-49e9-9ebb-eb4569a51713
                userId: efe672e3-b8e1-4ac0-b193-4495fbd0ce90
          - think: 10
          - patch:
              url:               /songs/000c5d3f-40e9-49e9-9ebb-
eb4569a51713/like # like the song
          - think: 5
          - get:
              url: /songs/liked
          - think: 3
          - post:
              url:           /playlists/0278c4c0-b414-4325-a227-
5d70f812c7bd/songs # add the song to the playlist
              json:
```

```
                songId: 000c5d3f-40e9-49e9-9ebb-eb4569a51713
          - think: 5
```

Лістинг Б.3 – Конфігураційний файл робочого процесу GitHub Actions

```
name: Release
'on':
  workflow_dispatch:
    inputs:
      version:
        description: 'For example: 1.0.0'
        required: true
      release_type:
        description: 'Release type: stage, prod'
        required: true
        type: choice
        options:
          - stage
          - prod
        default: stage
concurrency:
  group: >-
    ${{ github.workflow }} @ ${{
github.event.pull_request.head.label ||
    github.head_ref || github.ref }}
  cancel-in-progress: true
env:
  VERSION_CODE: '${{ github.run_number }}'
  VERSION_NAME: '${{ github.event.inputs.version }}'
  PACKAGE_NAME_ANDROID: '${{ secrets.PACKAGE_NAME_ANDROID }}'
  PACKAGE_NAME_IOS: '${{ secrets.PACKAGE_NAME_IOS }}'
  STAGE: '${{ github.event.inputs.release_type }}'
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4
```

```
            with:
              fetch-depth: 0
          - name: Setup Node
            uses: actions/setup-node@v4
            with:
              node-version: 18
              cache: yarn
              registry-url: 'https://registry.npmjs.org'
          - name: Install npm deps
            run: yarn install --frozen-lockfile --ignore-scripts
          - name: Lint
            run: yarn validate
      build-android:
        needs: lint
        runs-on: ubuntu-latest
        steps:
          - name: Checkout
            uses: actions/checkout@v4
            with:
              fetch-depth: 0
          - name: Setup Node
            uses: actions/setup-node@v4
            with:
              node-version: 18
              cache: yarn
              registry-url: 'https://registry.npmjs.org'
          - name: Cache Gradle Wrapper
            uses: actions/cache@v4
            with:
              path: ~/.gradle/wrapper
              key: >-
                ${{ runner.os }}-gradle-wrapper-${{
                hashFiles('gradle/wrapper/gradle-
wrapper.properties') }}
          - name: Cache Gradle Dependencies
            uses: actions/cache@v4
            with:
```

```
            path: ~/.gradle/caches
            key: >-
              ${{ runner.os }}-gradle-caches-${{
              hashFiles('gradle/wrapper/gradle-
wrapper.properties') }}
            restore-keys: |
              ${{ runner.os }}-gradle-caches-
        - name: Install npm deps
          run: yarn install --frozen-lockfile
        - name: Setup .env file
          run: |
            ENV_FILE=".env"
            if [ "${{ env.STAGE }}" == "stage" ]; then
              API_URL="${{ secrets.API_URL_STAGE }}"
            else
              API_URL="${{ secrets.API_URL_PROD }}"
            fi
        - name: Setup Ruby
          uses: ruby/setup-ruby@v1
          with:
            ruby-version: 3.1.2
        - name: Setup Java
          uses: actions/setup-java@v4
          with:
            java-version: 11
            distribution: temurin
        - name: Setup Android SDK
          uses: android-actions/setup-android@v3
        - name: Install ruby deps
          run: bundle lock --add-platform x86_64-linux && bundle
install
          working-directory: android
        - name: Install fastlane plugins
          run: bundle exec fastlane install_plugins --verbose
          working-directory: android
        - name: Build Android
```

```
            run: chmod +x gradlew && bundle exec fastlane android
build
            working-directory: android
          - name: Setup git
            run: >-
              git config --global user.email
              "41898282+github-
actions[bot]@users.noreply.github.com" && git config
              --global user.name "github-actions[bot]"
          - name: Push Android build changes to temporary branch
            run: >
              git       checkout       -b       temp-android-${{
github.event.inputs.version }}

              git   commit   -a   -m   "Android   build   for   v${{
github.event.inputs.version
              }}" || true

              git       push       origin       temp-android-${{
github.event.inputs.version }}
          - name: Upload APK artifact
            if: '${{ env.STAGE == ''stage'' }}'
            uses: actions/upload-artifact@v4
            with:
              name: android-build-artifact
              path:     android/app/build/outputs/apk/release/app-
release.apk
          - name: Upload AAB artifact
            if: '${{ env.STAGE == ''prod'' }}'
            uses: actions/upload-artifact@v4
            with:
              name: android-build-artifact
              path:  android/app/build/outputs/bundle/release/app-
release.aab
          - name: Clear branch
            if: failure()
            run: >-
```

```
              git    push    origin    --delete    temp-android-${{
github.event.inputs.version
              }}
       build-ios:
         needs: lint
         runs-on: macos-latest
         steps:
           - name: Checkout
             uses: actions/checkout@v4
             with:
               fetch-depth: 0
           - name: Setup Node
             uses: actions/setup-node@v4
             with:
               node-version: 18
               cache: yarn
               registry-url: 'https://registry.npmjs.org'
           - name: Install npm deps
             run: yarn install --frozen-lockfile --ignore-scripts
           - name: Setup Ruby
             uses: ruby/setup-ruby@v1
             with:
               ruby-version: 3.1.2
           - name: Install ruby deps
             run: bundle lock --add-platform x86_64-linux && bundle
install
             working-directory: ios
           - name: Set version code
             run: fastlane ios version
             working-directory: ios
           - name: Add test notes
             run: >
               echo      "${{      env.VERSION_NAME      }}      ${{
github.event.inputs.release_type }}"
               > ios/TestFlight/WhatToTest.en-US.txt
           - name: Setup git
             run: >-
```

```
              git config --global user.email
              "41898282+github-
actions[bot]@users.noreply.github.com" && git config
              --global user.name "github-actions[bot]"
          - name: Push iOS build changes to temporary branch
            run: >
              git         checkout         -b         temp-ios-${{
github.event.inputs.version }}

              git   commit   -a   -m   "iOS   build   for   v${{
github.event.inputs.version }}"
              || true

              git         push         origin         temp-ios-${{
github.event.inputs.version }}
          - name: Clear branch
            if: failure()
            run:   'git   push   origin   --delete   temp-ios-${{
github.event.inputs.version }}'
      release:
        needs:
          - build-android
          - build-ios
        runs-on: ubuntu-latest
        steps:
          - name: Checkout
            uses: actions/checkout@v4
            with:
              fetch-depth: 0
          - name: Setup Node
            uses: actions/setup-node@v4
            with:
              node-version: 18
              cache: yarn
              registry-url: 'https://registry.npmjs.org'
          - name: Install npm deps
            run: yarn install --frozen-lockfile --ignore-scripts
```

```
      - name: Setup git
        run: >-
          git config --global user.email
          "41898282+github-
actions[bot]@users.noreply.github.com" && git config
          --global user.name "github-actions[bot]"
      - name: Merge Android and iOS build changes
        run: |
          if [[ "${{ env.STAGE }}" == "prod" ]]; then
            git checkout main
            git pull origin main
            git merge -X theirs origin/temp-android-${{
github.event.inputs.version }}
            git merge -X theirs origin/temp-ios-${{
github.event.inputs.version }}
          else
            git checkout develop
            git pull origin develop
            git merge -X theirs origin/temp-android-${{
github.event.inputs.version }}
            git merge -X theirs origin/temp-ios-${{
github.event.inputs.version }}
          fi
      - name: Remove temporary branches
        run: >
          git push origin --delete temp-android-${{
github.event.inputs.version
          }}

          git push origin --delete temp-ios-${{
github.event.inputs.version }}
      - name: Download android build artifact
        uses: actions/download-artifact@v4
        with:
          name: android-build-artifact
          path: artifacts
      - name: Release stage
```

```
            if: '${{ env.STAGE == ''stage'' }}'
            run: 'yarn release ${{ github.event.inputs.version }}
--ci --preRelease=rc'
            env:
              GITHUB_TOKEN: '${{ secrets.GITHUB_TOKEN }}'
          - name: Release prod
            if: '${{ env.STAGE == ''prod'' }}'
            run: 'yarn release ${{ github.event.inputs.version }}
--ci'
            env:
              GITHUB_TOKEN: '${{ secrets.GITHUB_TOKEN }}'
```