МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

кафедра комп'ютерної інженерії та інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему
Побудова конвеєра постачання програмного забезпечення

Виконав: студент групи 1П-20
Спеціальності
121 – «Інженерія програмного забезпечення»

Станіслав АНГОЛЮК

Керівник:
Станіслав МАРЧЕНКО

Черкаси 2024

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ БІЗНЕС-КОЛЕДЖ

Кафедра комп’ютерної інженерії та інформаційних технологій
(повна назва випускової кафедри)

Спеціальність 121 “Інженерія програмного забезпечення”
(шифр і назва спеціальності)

Освітня програма Інженерія програмного забезпечення
(назва освітньої програми)

ЗАТВЕРДЖУЮ

Завідувач кафедри
комп’ютерної інженерії та
інформаційних технологій
(назва кафедри)
______________ Хотунов В.І.
(підпис) (ПІБ)
«____»____________2023 р.

З А В Д А Н Н Я

НА ВИПУСКНУ РОБОТУ СТУДЕНТУ

Анголюку Станіславу Дмитровичу
(прізвище, ім’я, по батькові студента)

1. Тема випускної роботи Побудова конвеєра постачання програмного забезпечення

Керівник роботи Марченко Станіслав Віталійович, спеціаліст I категорії
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від “ 13 ” жовтня 2023 року № 65У.

2. Строк подання студентом випускної роботи 03.06.2024

3. Вихідні дані до випускної роботи мова програмування TS та технологій React.js/Node.js (мікросервісна архітектура) і Next.js (монолітний додаток), сервер неперервної інтеграції GitHub Actions, управління артефактами за допомогою Docker, розгортання вебдодатка в Amazon Web Services, моніторинг засобами AWS CloudWatch.

4. Зміст випускної роботи (перелік питань, які потрібно розробити) огляд предметної області (базові поняття в контексті постачання програмного забезпечення, інструменти побудови конвеєра постачання), архітектурний опис вебдодатків (концептуальна схема додатку, інструментальні засоби програмної реалізації, прототипування вебсайту), впровадження конвеєра постачання програмного забезпечення (здійснення неперервної інтеграції коду, упаковка програмного коду, розгортання програмного забезпечення, моніторинг додатка).

5. Дата видачі завдання 15.09.2023р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання з підписами наукового керівника і студента
1	Вступ	20.10.2023	
2	Розділ 1. Огляд предметної області	22.12.2023	
3	Розділ 2. Архітектурний опис вебдодатків	15.03.2024	
4	Розділ 3. Впровадження конвеєра постачання програмного забезпечення	15.05.2024	
5	Висновки	17.05.2024	
6	Оформлення випускної роботи (чистовий варіант)	27.05.2024	
7	Здача випускної роботи на кафедру для рецензування (за 14 днів до захисту)	31.05.2024	
8	Перевірка випускної роботи на наявність ознак плагіату (за 10 днів до захисту)	03.06.2024	
9	Подання випускної роботи на затвердження завідувачу кафедри (за 7 днів до захисту)	06.06.2024	

Студент ____________________ Анголюк С.Д.
(підпис) (прізвище та ініціали)

Керівник роботи ____________________ Марченко С.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота присвячена аналізу, розробці та впровадженню конвеєра постачання програмного забезпечення для вебдодатка. Метою роботи було створення ефективного процесу розробки та постачання програмного забезпечення, який забезпечує високу якість, швидкість та надійність.

У роботі було розглянуто базові поняття в контексті постачання програмного забезпечення, що включало основні терміни та концепції цієї сфери, проведено аналіз інструментів, які використовуються для побудови конвеєра постачання, таких як системи контролю версій, засоби автоматизації збірок, платформи для неперервної інтеграції та розгортання, сформульована задача на розробку, включаючи вимоги до системи та очікувані результати.

Було розроблено концептуальну схему програмного забезпечення, яка включала основні компоненти та їх взаємодію. Описано інструментальні засоби програмної реалізації, які використовувалися для розробки, тестування та розгортання вебдодатка і впроваджено конвеєр постачання програмного забезпечення, включаючи здійснення неперервної інтеграції коду з автоматизованим тестуванням та збіркою. Також описано методи упаковки програмного коду для різних середовищ розгортання. Розглянуто аспекти моніторингу додатка для забезпечення його стабільної роботи та швидкого виявлення проблем. Окрім того, були визначені перспективи для подальшого удосконалення конвеєра постачання, включаючи можливості інтеграції нових інструментів та методів автоматизації.

Робота зробила внесок у розвиток ефективних методик постачання програмного забезпечення, що може бути корисним для розробників та IT-команд у покращенні їхніх процесів розробки та впровадження програмних продуктів.

ABSTRACT

The thesis was devoted to the analysis, development and implementation of a software delivery pipeline for a web application. The work method was to create an efficient software development and delivery process that ensures high quality, speed and reliability.

The work considered the basic concepts in the context of software delivery, which included the main terms and concepts of this field, analyzed the tools used to build the delivery pipeline, such as version control systems, build automation tools, platforms for continuous integration and deployment, formulated development task, including system requirements and expected results.

A conceptual diagram of the software was developed, which included the main components and their interaction. The software implementation tools used to develop, test, and deploy the web application are described, and the software delivery pipeline is implemented, including continuous code integration with automated testing and assembly. It also describes methods for packaging software code for different deployment environments. Aspects of monitoring the application to ensure its stable operation and quick detection of problems are considered. In addition, prospects were identified for further improvement of the supply pipeline, including the integration of new tools and automation methods.

The work has contributed to the development of effective software delivery methodologies, which can be useful for developers and IT teams in improving their processes for developing and implementing software products.

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

IaaS – Infrastructure as a Service

PaaS – Platform as a Service

SaaS – Software as a Service

IaC – Infrastructure as Code

CI/CD – Continuous Integration/Continuous Deployment

RDS – Relational Database Service

API – Application Programming Interface

ORM – Object-Relational Mapping

AWS – Amazon Web Services

DevOps – Development and Operations

EC2 – Elastic Compute Cloud

ECS – Elastic Container Service

ЗМІСТ

ВСТУП 3
РОЗДІЛ 1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ 5
1.1. Базові поняття в контексті постачання програмного забезпечення 5
1.2. Інструменти побудови конвеєра постачання 9
1.3. Постановка задачі на розробку 16
РОЗДІЛ 2 АРХІТЕКТУРНИЙ ОПИС ВЕБДОДАТКА 19
2.1. Концептуальна схема програмного забезпечення 19
2.2. Інструментальні засоби програмної реалізації 23
2.3. Детальний опис архітектури додатків 25
РОЗДІЛ 3 ВПРОВАДЖЕННЯ КОНВЕЄРА ПОСТАЧАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 30
3.1. GitHub Actions як інструмент неперервної інтеграції коду 30
3.2. Упаковка програмного коду 36
3.3. Моніторинг проєкта 40
ВИСНОВКИ 44
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 45

ВСТУП

Актуальність обраної теми. Побудова конвеєра постачання програмного забезпечення – це підхід до розробки, який спрямований на автоматизацію та оптимізацію процесів від написання коду до випуску готового продукту. Замість традиційного ручного управління кожним етапом розробки окремо, конвеєр постачання дає змогу автоматизувати ці процеси та створити їхню неперервну інтеграцію. Це означає, що коли розробник завершує певну частину коду, він може автоматично викликати тестування цього коду, а потім автоматично розгорнути його в тестове або продуктове середовище. Цей процес автоматизації допомагає уникнути помилок, збільшує швидкість розробки та забезпечує вищу якість продукту. Крім того, конвеєр постачання дає змогу зменшити час між розробкою нового функціоналу та його випуском на ринок. Замість того, щоб чекати на великий реліз з багатьма новими функціями, команди можуть швидко реагувати на зміни та випускати оновлення в невеликих ітераціях. Це дозволяє компаніям бути більш гнучкими та реагувати на вимоги ринку набагато швидше.

При впровадженні конвеєра постачання DevOps-спеціаліст стикається з низкою інженерних проблем: складність інтеграції різних інструментів у єдиний механізм, вплив масштабування проекту на роботу конвеєра, безпекові питання автоматизації та контролю доступу, налагодження якісної системи моніторингу й аналізу процесів і помилок тощо. Звідси, побудова конвеєра постачання є комплексним завданням, яке охоплює переважну більшість діяльностей у галузі інженерії програмного забезпечення та потребує всебічного дослідження. Здатність виконання таких видів робіт свідчить про високий рівень кваліфікації відповідного спеціаліста та є ключовим елементом для забезпечення конкурентоспроможності компаній у сучасному світі технологій. Вона дозволяє компаніям розробляти та впроваджувати програмне забезпечення швидше, ефективніше та з вищою якістю, що є вирішальним для їхнього успіху на ринку.

Мета дослідження. Метою проєкту є вдосконалення навичок побудови та впровадження конвеєру постачання програмного забезпечення, спрямованого на

автоматизацію процесів розробки вебдодатків на основі мови програмування TypeScript та пов’язаних з нею технологій.

Об’єкт дослідження. Об’єктом дослідження є автоматизація процесів розробки в галузі вебдодатків, зокрема на засадах неперервних інтеграції та постачання програмного забезпечення.

Предмет дослідження. Предметом дослідження виступають сучасні технології та інструменти конвеєризації розробки вебдодатків, включаючи етапи проєктування, збирання, тестування, контейнеризації та розгортання програмного продукту.

Завдання дослідження. Практичні завдання в контексті вказаної тематики та мети дослідження такі:

1) аналіз сучасних технологій розробки та впровадження програмного забезпечення. Вибір та налаштування інструментів для автоматизації процесів конвеєра постачання;

2) розробка та імплементація конвеєра постачання з урахуванням потреб та характеристик конкретного проекту;

3) тестування та оптимізація конвеєра для забезпечення швидкості, надійності та якості випуску програмного забезпечення;

4) оцінка ефективності впровадження конвеєра постачання на практиці та визначення його впливу на процес розробки та якість продукту.

РОЗДІЛ 1
ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Базові поняття в контексті постачання програмного забезпечення

Нині сфера хмарних послуг демонструє значний ріст та активний розвиток. Ринковий обсяг цих послуг очікується досягнути вражаючу суму в $600 мільярдів до 2024 року, що становить значне збільшення порівняно з показниками попередніх років [5]. Спостерігається значний попит на послуги хмарного зберігання даних, що відображається у великому обсязі витрат на IT-інфраструктуру, спрямований на хмарні сервіси. Тенденція росту користувачів хмарних послуг також вражає: за останні роки кількість користувачів зростає експоненціально, що свідчить про загальне розповсюдження цих технологій [5].

Організації все швидше впроваджують хмарні технології, з високим відсотком тих, хто вже користується хмарними послугами, і значною кількістю тих, хто планує розширити свої хмарні ініціативи. Ринок обчислювальних послуг в хмарі також стрімко росте, прогнозується значне збільшення його обсягу до 2032 року [7].

Усі ці показники свідчать про те, що хмарні технології стають не лише популярними, але й необхідними для багатьох бізнесів та організацій. Їхнє впровадження відображає загальну тенденцію до цифрової трансформації і є ключовим елементом стратегічного розвитку в багатьох галузях.

Різноманітні моделі обчислень, такі як SaaS (Software as a Service), PaaS (Platform as a Service) та IaaS (Infrastructure as a Service), надають ефективні рішення для вирішення цих викликів, дозволяючи підприємствам пристосовуватися до зростаючих потреб ринку. SaaS [44] ставить своїм основним завданням полегшення доступу до програм для користувачів через хмару. Замість традиційного моделлю встановлення та обслуговування програм на локальних серверах, SaaS надає можливість використання програм через Інтернет. AWS (Amazon Web Services) [2] є провідним постачальником хмарних послуг, включаючи широкий спектр SaaS-продуктів, які охоплюють різні сфери бізнесу.

PaaS [4] вирізняється тим, що надає розробникам платформу та середовище для створення, тестування та впровадження програм без необхідності управління інфраструктурою. AWS Elastic Beanstalk є однією з послуг, яка спрощує розгортання та управління додатками на хмарній інфраструктурі.

IaaS [4] надає віртуальні обчислювальні ресурси, такі як обчислювальна потужність та зберігання даних, що дозволяє підприємствам будувати та управляти власною інфраструктурою в хмарі. AWS EC2 (Elastic Compute Cloud) [3] є прикладом IaaS, надаючи масштабовані обчислювальні ресурси в хмарному середовищі. Загальне співвідношення доступної для користувача функціональності показано на рисунку 1.1.

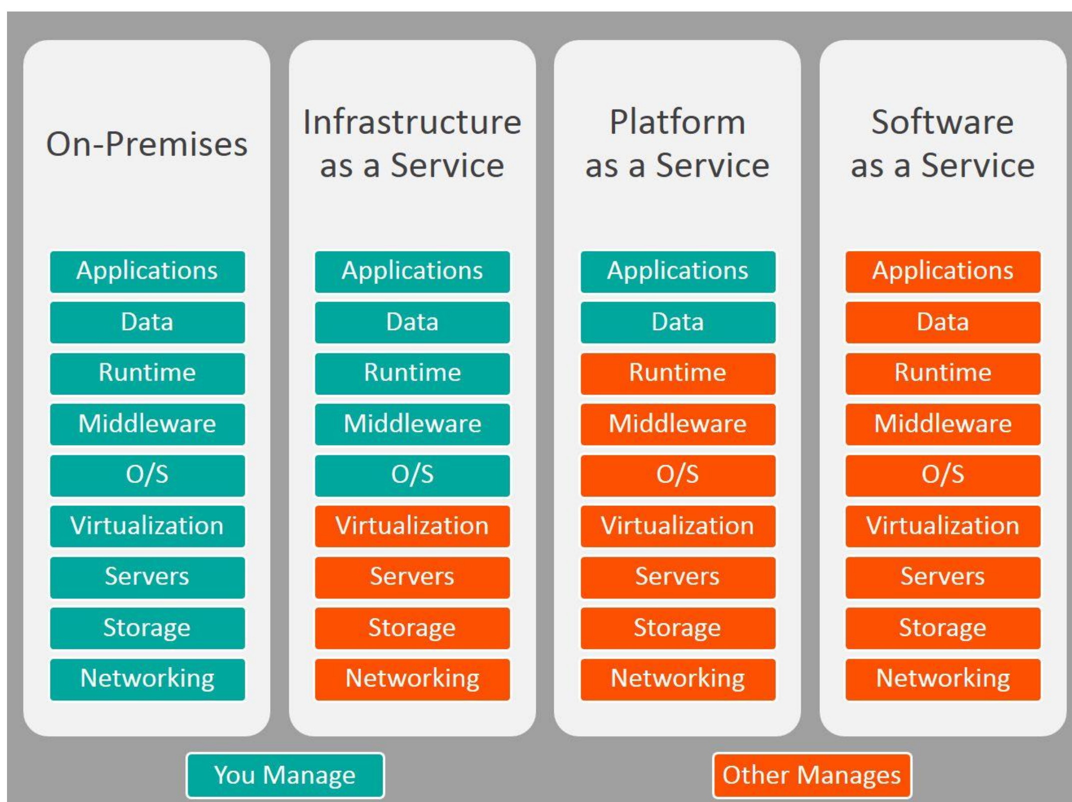


Рисунок 1.1 – Відмінності між моделями хмарних обчислень

Ці різновиди обчислювальних послуг впливають на ринок, роблячи розробку та впровадження програм більш швидкими, гнучкими та ефективними. За даними Flexera [8], в 2022 році понад 90% підприємств використовуватимуть хоча б один вид хмарних послуг. SaaS, PaaS та IaaS стають невід’ємною

частиною стратегії розвитку підприємств, допомагаючи їм адаптуватися до швидкозмінного технологічного середовища.

Одночасно з цим існує питання архітектурних підходів до розробки програмного забезпечення – мікросервіси та моноліти. Мікросервісна архітектура, яка передбачає розбиття програм на невеликі, незалежні компоненти, стає все популярнішою. За даними ThoughtWorks [12], майже 50% опитаних компаній вже використовують мікросервіси чи планують їх впровадження.

Зокрема, є широке застосування мікросервісної архітектури в розробці ReactJS-додатків [14]. Реактивні компоненти, які є ключовим елементом ReactJS, можуть бути індивідуальними мікросервісами, які легко масштабувати та підтримувати.

З іншого боку, традиційні монолітні архітектури також залишаються важливими, особливо для менших проєктів. Варто звертати увагу на індивідуальні потреби проєкту та вибрати підхід, який краще відповідає конкретним завданням.

Обидва ці підходи мають свої переваги та недоліки. Мікросервіси забезпечують гнучкість та масштабованість, але можуть вимагати більше управління та обслуговування. Моноліти можуть бути простішими у розробці та обслуговуванні, але їх важче масштабувати з часом.

Ідеологія конвеєризації постачання програмного забезпечення є складовою DevOps – практичного підходу до розробки програмного забезпечення, який поєднує в собі розробку (Development) і операції (Operations). Мета DevOps полягає в автоматизації процесів розробки, тестування та впровадження програмного забезпечення, забезпеченні високої швидкості розгортання, зменшенні часу між внесенням змін та їх випуском в продукцію, а також у підвищенні співпраці між розробниками та операторами систем. У цілому, DevOps спрямований на створення більш ефективного, стабільного та швидкого процесу розробки і розгортання програмного забезпечення.

Відповідна функціональна підмножина – CI/CD – це скорочення від Continuous Integration (постійна інтеграція) та Continuous Deployment (постійне розгортання). Ці практики є ключовими складовими DevOps методології. Постійна інтеграція передбачає автоматичне об'єднання коду розробників для якнайшвидшого виявлення та розв'язання інтеграційних проблем. Постійне постачання стосується автоматизованого процесу випуску зміненого коду в виробниче середовище після проходження всіх необхідних тестів та перевірок. Використання CI/CD допомагає забезпечити швидке, надійне та автоматизоване внесення змін у програмне забезпечення.

На рівні поставки програмного забезпечення засоби CI/CD використовуються для автоматизації процесів розробки та розгортання. Служба GitHub Actions [10] надає інструменти для реалізації CI/CD, що дає змогу розробникам автоматизувати тести, збірку та розгортання додатків.

Окремо варто відзначити інструмент Docker, який забезпечує контейнеризацію додатків. Використання Docker [1] надає можливість створювати, розгортати та запускати додатки в стандартизованих контейнерах, що полегшує переносимість та масштабованість. За даними Docker Usage [1], кількість завантажень Docker досягла позначки у 12 мільярдів, свідчать про високий рівень зацікавленості розробників у цьому інструменті.

Поєднуючи все вище сказане, ми отримаємо конвеєр для нашого застосунку, проходячи який він буде автоматично:

- відтестований і зібраний з допомогою Github Actions. Завдяки можливостям Github Actions ми маємо можливість проводити будь-які тести: модульні, e2e, інтеграційні, а також зробити готове складання додатка;
- ізольований в контейнері Docker;
- запущений і, в випадку успішного проходження всіх попередніх пунктів, доступний користувачам за допомогою AWS-сервісів ECS, EKS, EC2, Kubernetes.

Все це буде здійснено із мінімальним залученням програміста, затрати часу передбачаються тільки на створення цього конвеєру, що є дуже вигідним у випадку проектів, які планується підтримувати довго і оновлювати регулярно. Вигляд типового конвеєру показано на рис. 1.2.

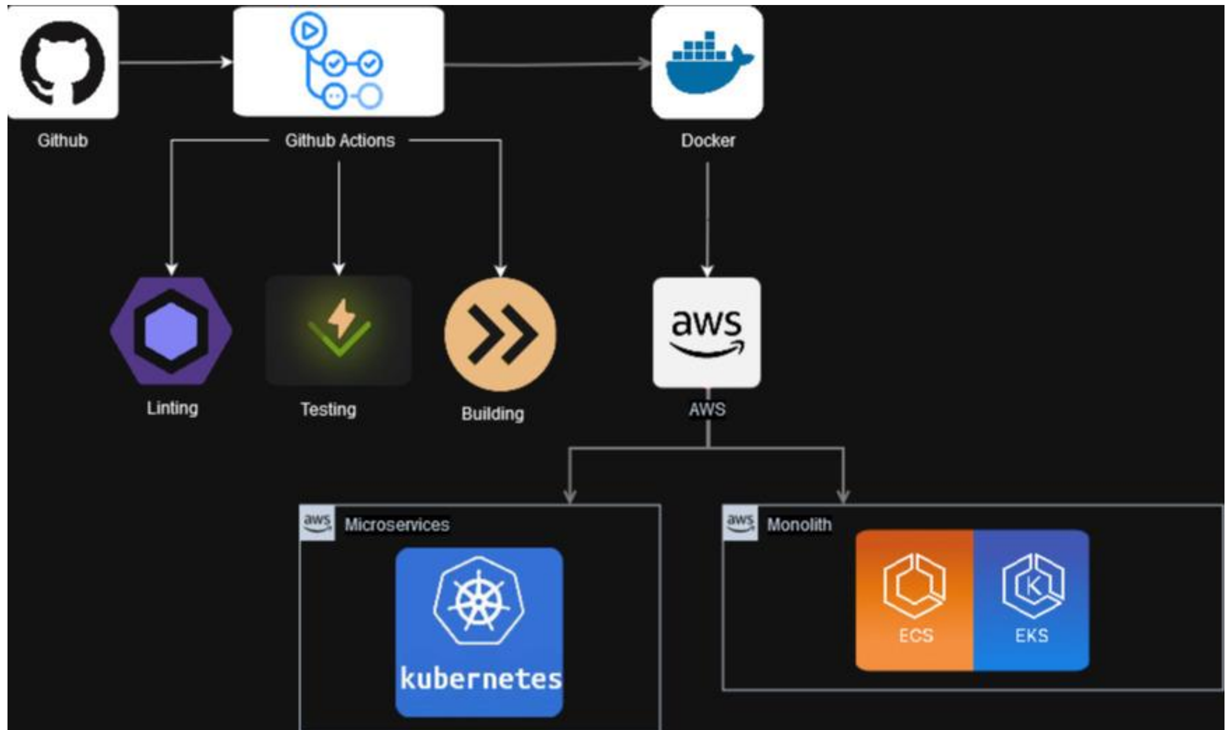


Рисунок 1.2 – Вигляд конвеєра

Загалом, ці технології формують фундаментальну частину сучасного процесу розробки програмного забезпечення. Вони надають підприємствам та розробникам необхідні інструменти для ефективної та конкурентоспроможної розробки та управління програмами.

1.2. Інструменти побудови конвеєра постачання

Побудова конвеєра постачання може відбуватися в кілька кроків. До переліку можливих етапів можна віднести такі:

– Code Commit. Перший етап конвеєра CI/CD – це здійснення змін у коді. На цьому кроці розробники фіксують свої зміни в системі контролю версій, такій як Git. Система контролю версій забезпечує історію всіх внесених

змін до кодової бази і дає змогу розробникам спільно працювати над змінами в коді. Вигляд історії комітів для проєкту відображено на рис. 1.3.

– Code Build. Другий етап конвеєра CI/CD – це збирання коду. На цьому етапі вихідний код перетворюється у виконуваний артефакт, такий як бінарний файл або контейнер Docker. Процес побудови може включати компіляцію коду, упаковку залежностей та створення артефакту.

– Code Test. Третій етап конвеєра CI/CD – це тестування коду. На цьому кроці зібраний артефакт тестується за допомогою автоматизованих тестів для перевірки відповідності якості, такої як функціональність, продуктивність і безпека. Сюди можна включити модульні тести, інтеграційні тести та тести зі сторони користувача.

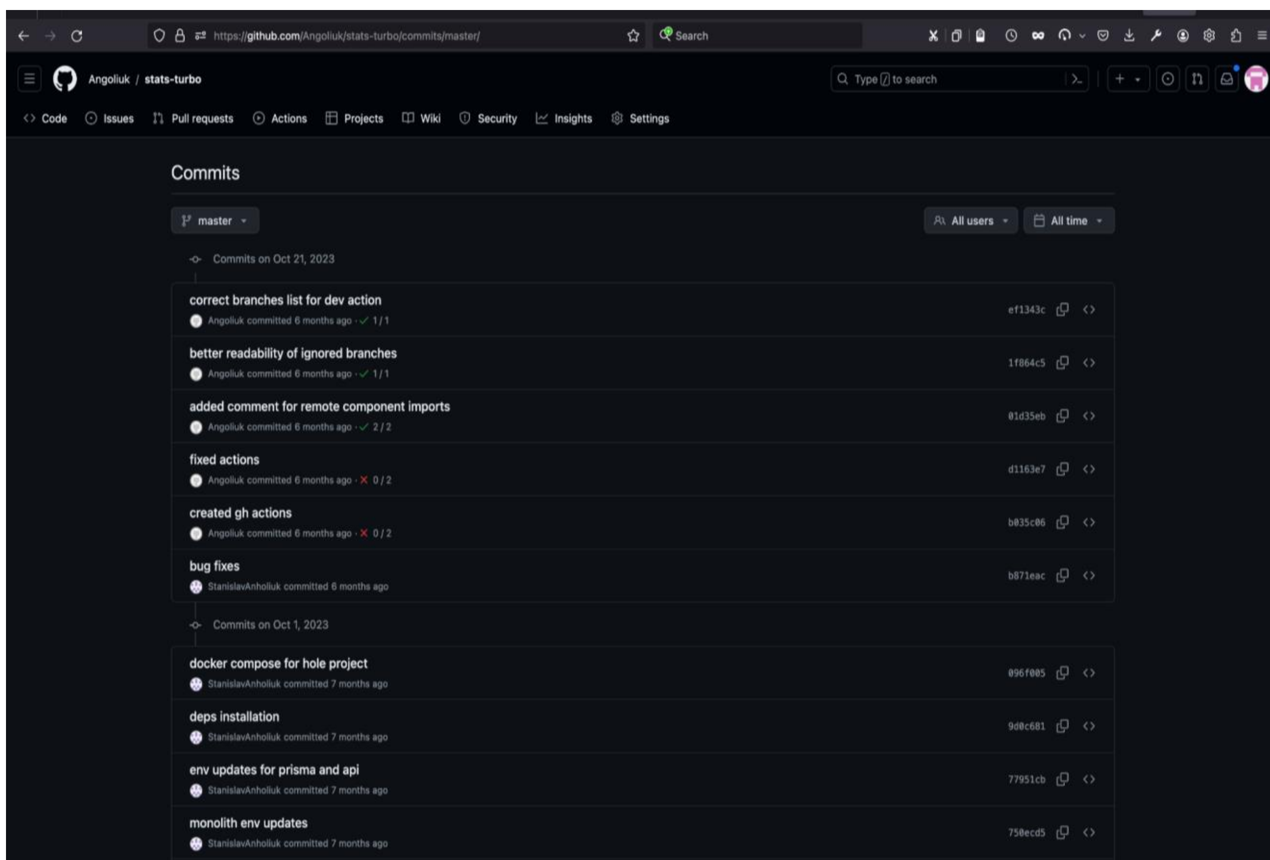


Рисунок 1.3 – Коміти проєкта

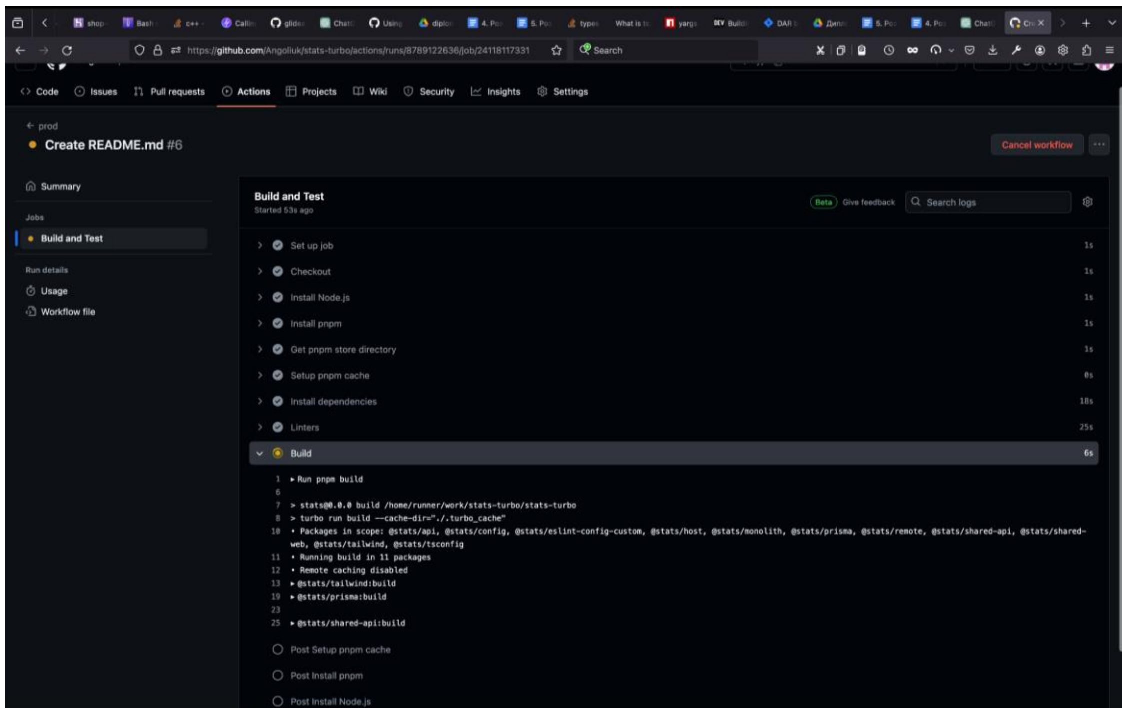


Рисунок 1.4 – CI-процес для проєкта

– Code Review. Четвертий етап конвеєра CI/CD – це перегляд коду. На цьому етапі зміни у коді переглядаються колегами або автоматизованими інструментами для забезпечення відповідності стандартам коду та кращим практикам. Перегляд коду допомагає виявити та виправити проблеми до їх злиття у головну кодову базу.

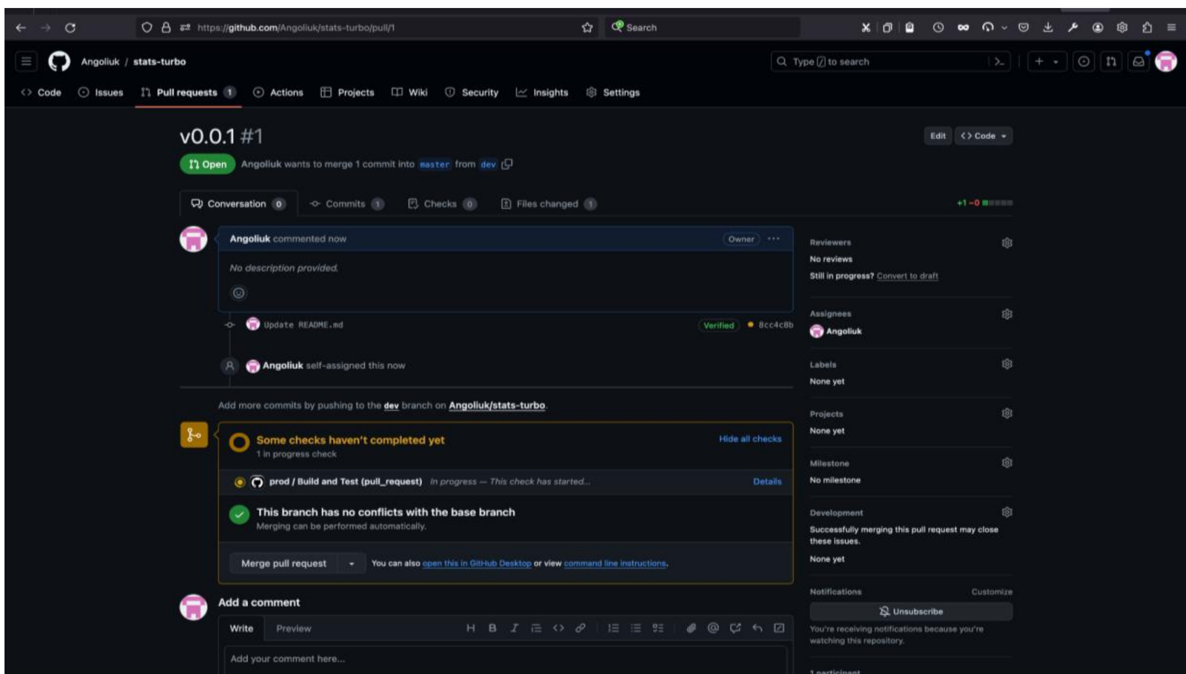


Рисунок 1.5 – Code Review проєкта

– Об’єднання кодової бази (Code Merge). П’ятий етап конвеєра CI/CD – це злиття коду. На цьому кроці зміни у коді зливаються у головну гілку системи контролю версій. Процес злиття може включати вирішення конфліктів та перевірку того, що зміни у коді не порушують існуючу кодову базу. Зразок виконаної дії зображено на рис. 1.6.

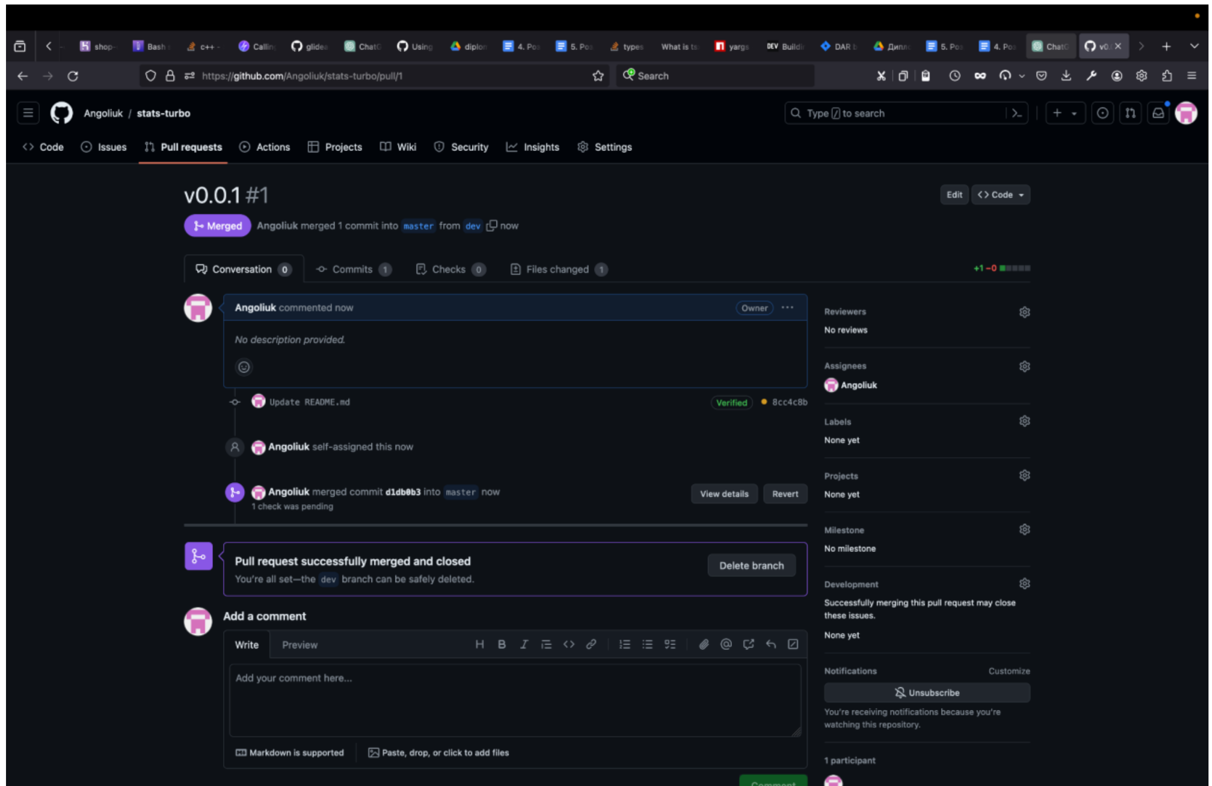


Рисунок 1.6 – Code Merge проекту

– Випуск артефакта (Artifact Release). Шостий етап конвеєра CI/CD – це реліз артефакту. На цьому кроці побудований і відтестований артефакт випускається у виробничне середовище, зокрема на тестовий або виробничий сервер. Процес випуску може включати маркування артефакту, створення приміток до релізу та повідомлення зацікавленим сторонам.

– Розгортання (Deployment). Сьомий етап конвеєра CI/CD. На цьому кроці випущений артефакт розгортається у виробниче середовище, що може відбуватись як вручну, так і автоматично. Процес розгортання може включати налаштування середовища, міграцію даних та тестування розгорнутої програми.

– Моніторинг. Восьмий етап конвеєра CI/CD. На цьому кроці виробничне середовище моніториться для виявлення проблем або помилок, а також для збору метрик та журналів для аналізу та вдосконалення. Моніторинг допомагає забезпечити, що програмне забезпечення працює так, як очікувалося, і що будь-які проблеми вирішуються вчасно.

Для наведених вище етапів часто використовуються популярні та навіть стандартні рішення. Наприклад, на першому етапі – комітів коду – найчастіше застосовується сервіс GitHub. Він має свої переваги завдяки великій спільноті користувачів та підтримці відкритого програмного забезпечення, характеризується зручним інтерфейсом користувача та багатьма функцій для співпраці та управління проєктами. Доступні широкі можливості інтеграції з іншими сервісами CI/CD, які спрощують автоматизацію розробки.

Альтернативними популярними сервісами виступають GitLab і Bitbucket. Сервіс GitLab пропонує вбудований сервіс CI/CD (GitLab CI/CD), який надає змогує автоматизувати процеси збірки, тестування та розгортання. Його потужна система управління задачами та розгортанням дає можливість зосередитися на всій розробці програмного забезпечення на одній платформі. Вбудований Docker-реєстр полегшує контейнеризацію та розгортання додатків.

У той же час, Bitbucket надає інтеграцію з іншими сервісами Atlassian, такими як Jira та Confluence, що полегшує співпрацю та управління проектами. Доступна можливість безкоштовного використання для невеликих команд або приватних проєктів, широкі можливості роботи з різними типами репозиторіїв, включаючи Git та Mercurial.

Для збирання коду застосовують цілу низку різних інструментів. Популярним є AWS CodeBuild, який забезпечує повну інтеграцію з іншими AWS-сервісами, що дає змогу створювати різноманітні складні робочі процеси. Пропонуються гнучка конфігурація оточення збірки, яка дає можливість використовувати різні типи образів, включаючи Docker, та масштабована архітектура, яка дозволяє швидко збирати великі проєкти та працювати з великими обсягами даних.

Однією з найпоширеніших альтернатив є Jenkins. Платформа пропонує широкі можливості налаштування та розширення завдяки великій кількості доступних плагінів, гнучкість в налаштуванні робочих процесів збірки, тестування та розгортання. Відкритий первинний код дає змогу спільноті користувачів активно розробляти та підтримувати проєкт.

Сервіс Travis CI пропонує можливість швидко налаштувати CI/CD для проєкту. Інтеграція з репозиторіями GitHub спрощує автоматизацію для відповідних проєктів. Доступна можливість безкоштовного використання для відкритих проєктів, що сприяє розвитку відкритого програмного забезпечення.

На етапах тестування, огляду та об'єднання кодової бази все залежатиме від обраного технологічного стеку та сервісів фіксації коду. Випуск артефактів може здійснюватись такими інструментами:

1) AWS CodeArtifact. Повністю керований сервіс для зберігання, керування та розповсюдження артефактів програмного забезпечення. Інтегрований з іншими сервісами AWS, такими як AWS CodePipeline та AWS CodeBuild, що спрощує процес автоматизації розробки. Присутня підтримка приватних репозиторіїв з можливістю контролю доступу, що забезпечує безпеку та конфіденційність артефактів.

2) JFrog Artifactory. Потужний репозиторій артефактів, який підтримує різноманітні типи пакетів, включаючи Java, Docker, npm, RubyGems та багато інших. Його гнучкі можливості керування версіями, авторизацією та доступом дають змогу точно контролювати процеси розповсюдження артефактів. Для продукту характерна масштабована архітектура, яка підтримує високий обсяг даних та масштабування для великих організацій.

3) GitHub Packages. Сервіс для зберігання та керування пакетами програмного забезпечення, який інтегрований з GitHub репозиторіями. Інтегрований з іншими сервісами GitHub, що полегшує розробку та автоматизацію процесів роботи з пакетами. Доступна підтримка різних типів пакетів, включаючи Docker, Maven, npm, RubyGems, PyPI та інші.

На етапі розгортання можемо мати справу зі спектром різних інструментів. Серед них популярні такі:

1) AWS CodeDeploy. Повністю керований сервіс для автоматизованого розгортання програмного забезпечення на різних інфраструктурних середовищах, включаючи Amazon EC2, Lambda, і сервіси Elastic Beanstalk. Надає можливість автоматичного розгортання коду з репозиторіїв GitHub, Bitbucket або Amazon S3. Присутня гнучка конфігурація розгортання, включаючи ролі, стратегії розгортання та різні типи середовищ.

2) Jenkins. Вже згаданий вище програмний продукт пропонує широкі можливості інтеграції з іншими інструментами CI/CD та хмарними платформами. Підтримує різні стратегії розгортання

3) Heroku. Хмарна платформа, яка надає послуги розгортання та керування додатками. Характеризується простотою використання та автоматизованого розгортання з допомогою Git. Пропонує широкі можливості масштабування та підтримку різних типів додатків, включаючи вебдодатки, мікросервіси та контейнеризовані додатки.

Впровадження моніторингу іноді залежить від стеку. Наприклад, PM2 – це менеджер процесів для Node.js додатків, який дає змогу керувати запуском, моніторингом та управлінням Node.js процесами на сервері. Основні переваги PM2:

– автоматичне відновлення: PM2 автоматично перезапускає додатки в разі їх падіння або збою;

– моніторинг: платформа надає інтерфейс командного рядка та вебпанель для моніторингу стану додатків, їх ресурсів та використання пам’яті;

– керування розгортанням: PM2 дає змогу керувати розгортанням додатків та їх версіями;

– журналювання: платформа може автоматично записувати журнали додатків для наступного аналізу та відлагодження.

У хмарній інфраструктурі для моніторингу та журналювання застосовується служба AWS CloudWatch. До основних переваг сервісу можна віднести:

– моніторинг ресурсів: AWS CloudWatch надає змогу моніторити різні ресурси AWS, такі як EC2 і RDS і вимірювати їх використання ресурсів, стан та метрики продуктивності;

– сповіщення та тривоги: сервіс дає можливість налаштовувати сповіщення та тривоги на основі певних метрик, щоб вчасно реагувати на проблеми;

– аналіз журналів: CloudWatch Logs забезпечує збирання, моніторинг та аналіз журналів з різних сервісів AWS та власних додатків;

– інтеграцію з іншими сервісами AWS: AWS CloudWatch інтегрований з іншими службами AWS, що дозволяє використовувати його в різних аспектах розробки та експлуатації.

1.3. Постановка задачі на розробку

Для розробки в межах дипломної роботи пропонується побудова конвеєра постачання для вебдодатка. За результатами огляду популярних інструментів було вирішено використати:

1) ReactJS. Це один з найпопулярніших JavaScript-фреймворків для створення інтерактивних користувацьких інтерфейсів. ReactJS надає ефективну та динамічну модель роботи з відображенням даних, що робить його ідеальним вибором для розробки вебдодатків та інтерфейсів.

2) NextJS. Це фреймворк React для створення високопродуктивних вебдодатків. NextJS дає змогу легко реалізувати функціонал серверного рендерингу, статичного генерації та інших передових можливостей, що покращують швидкодію та роблять розробку більш зручною.

3) TurboRepo. Це інструмент, який допомагає зберігати та керувати великими монорепозиторіями. Використання TurboRepo полегшує спільну

роботу над багатьма проектами та зменшує складність управління залежностями та версіями.

4) TypeScript. Це мова програмування, яка додає статичну типізацію до JavaScript, забезпечуючи переваги такі як покращене розуміння коду, зменшення помилок та полегшення рефакторингу. TypeScript дозволяє збільшити швидкодію розробки та зробити код більш надійним.

5) Express: Це популярний фреймворк для створення серверних додатків на платформі Node.js. Використання Express дозволяє швидко створювати маршрути, обробники запитів та інші складові серверної частини вебдодатків.

Для автоматизації інфраструктури передбачене застосування таких програмних інструментів:

1) Github як загальна платформа для здійснення керування кодом. Він надає широкий набір функцій для спільної роботи, рецензування коду, відстеження проблем, автоматизації CI/CD та багато іншого. GitHub має високий рівень безпеки та надійності, забезпечуючи захист вашого коду та даних. Крім того, GitHub надає можливості керування доступом, аудиту та захисту вашого репозиторію.

2) CI-сервер на базі GitHub Actions. Інструмент інтегрованіий безпосередньо з репозиторіями GitHub, що забезпечує легке створення, налаштування та запуск робочих процесів прямо з вашого репозиторію. GitHub Actions доступний безкоштовно для відкритих проектів, що дозволяє спільноті використовувати потужні інструменти автоматизації без додаткових витрат. Сервіс дає змогу налаштовувати різні типи робочих процесів, включаючи CI/CD, автоматизоване тестування, розгортання та багато іншого. Ви можете створювати власні дії або використовувати сторонні дії для виконання різноманітних завдань у вашому репозиторії.

3) Управління артефактами за допомогою Docker. Контейнери Docker забезпечують можливість упаковувати програмне забезпечення та всі його залежності в один стандартизований пакет. Це дозволяє вам запускати

контейнери на будь-якому середовищі, яке підтримує Docker, незалежно від операційної системи або хмарної платформи. Контейнери Docker надають високий рівень ізоляції для програм та сервісів, що дає змогу запускати їх у відокремленому середовищі. Це допомагає уникнути конфліктів між додатками та забезпечує безпеку на рівні системи. Docker забезпечує швидке створення та розгортання контейнерів, а також можливість їх масштабування на основі потреб вашого додатку.

4) Розгортання вебдодатка засобами AWS. Хмарна платформа AWS пропонує один з найширших наборів можливостей на ринку з великою кількістю сервісів для обчислення, зберігання даних, баз даних, машинного навчання, аналізу даних, Інтернету речей (IoT) та багатьох інших. AWS інтегрується з великою кількістю інших сервісів та інструментів, що робить його привабливим вибором для побудови комплексних рішень. Багато сторонніх продуктів і сервісів також інтегруються з AWS. Провайдер хмарних послуг має глобальну мережу датацентрів, що дає йому змогу забезпечити низькі затримки та високу доступність для користувачів з різних куточків світу. Також AWS забезпечує високий рівень безпеки та відповідність різним стандартам безпеки та конфіденційності даних, таким як GDPR, HIPAA, PCI DSS тощо.

5) Моніторинг засобами AWS CloudWatch зумовлений вибором провайдера хмарних послуг. Як частина екосистеми Amazon Web Services, CloudWatch легко інтегрується з іншими сервісами AWS, що дає можливість легко моніторити та управляти різними аспектами вашого хмарного середовища. CloudWatch надає широкий набір інструментів для моніторингу та журналювання різних аспектів вашого хмарного середовища, включаючи метрики, журнали, тривоги та інші аналітичні функції.

Для кращої обґрунтованості проєктних рішень стосовно побудови конвеєра беремо дві версії коду в різних архітектурних стилях: монолітному та мікросервісному. У результаті очікуємо отримати розгорнутий вебдодаток, що пройшов статичний аналіз та збірку з первинного коду, модульне, інтеграційне та приймальне тестування, а також розгортання на базі сервісів AWS.

РОЗДІЛ 2
АРХІТЕКТУРНИЙ ОПИС ВЕБДОДАТКА

2.1. Концептуальна схема програмного забезпечення

На базі рішення стосовно розроблення двох аналогічних вебдодатків з різними архітектурними стилями спочатку опишемо їх концептуальну структуру та основні технологічні аспекти реалізації. Перший додаток буде монолітним, тобто весь його код і функціональність будуть розміщені в одному додатку. Фронтенд (клієнтська частина) та бекенд (серверна частина) будуть взаємодіяти між собою в рамках цього додатку, а база даних буде централізованою для всієї системи. Другий додаток відповідатиме мікросервісному стилю, що передбачає розділення функціональності на невеликі, незалежні компоненти – мікросервіси. Кожен мікросервіс буде відповідальний за свою конкретну частину функціональності.

Монолітний додаток буде реалізований за допомогою фреймворку Next.js, який надає можливість розробки як клієнтської, так і серверної частини в одному проєкті. Цей додаток буде зосереджений на функціональності статистики гравця для онлайн-ігор. Ключовими можливостями цього додатку будуть:

– відображення статистики гравців за весь період користування, що включатиме в себе різні показники, такі як загальна кількість гравців, найкращий результат, середній результат тощо;

– перегляд детальної статистики за кожним окремим матчем, включаючи інформацію про результати гравця, час гри.

Такий підхід дає змогу об’єднати усі необхідні функції в одному додатку. Монолітний додаток буде спроєктовано відповідно до модульної архітектури, що передбачає розділення функціональності на окремі модулі або компоненти. Цей підхід дозволяє полегшити розробку, тестування та підтримку додатку. Кожен модуль відповідає за певну частину функціоналу і може бути незалежно розроблений, тестований та масштабований. Концептуальна схема монолітного додатка наведена на рис. 2.1.

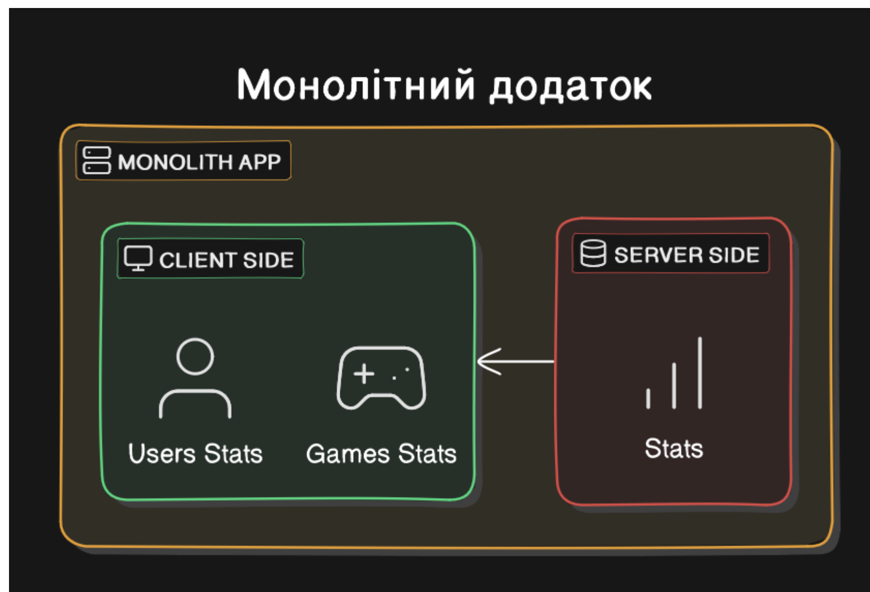


Рисунок 2.1 – Схема зв’язків монолітного додатку

Мікросервісний додаток буде розроблений за допомогою фреймворку Express для створення серверної частини та React для клієнтської частини. Цей підхід передбачає розділення функціоналу між незалежними мікросервісами, які взаємодіють між собою через API. Ключові можливості мікросервісного додатку будуть аналогічні монолітному:

– відображення статистики гравців за весь період користування;
– детальна статистика за кожним окремим матчем.

Проте в мікросервісному підході ця функціональність буде розділена між окремими сервісами, що надасть змогу краще масштабувати та підтримувати систему. Кожен сервіс буде відповідати за свою функціональну частину і може бути незалежно розроблений та розгорнутий.

Так як мікросервіси на бекенді – це вже досить популярна річ, було вирішено зробити так звані мікрофронтенди. Замість одного великого фронтенду функціональність розділено на два окремі фронтенди, кожен з яких відповідатиме за свою частину додатку. Перший фронтенд буде відповідати за відображення статистики гравців. Це головна частина додатку, де користувачі можуть переглядати інформацію про гравців та їхню статистику.

Другий фронтенд відповідатиме за відображення статистики ігор. Він буде вбудовуватися в головний фронтенд та відповідати за відображення детальної статистики про ігри. Обидва фронтенди будуть побудовані згідно з модульною архітектурою, що передбачає розділення функціональності на незалежні модулі або компоненти. Концептуальна взаємодія сервісів додатку показана на рис. 2.2.

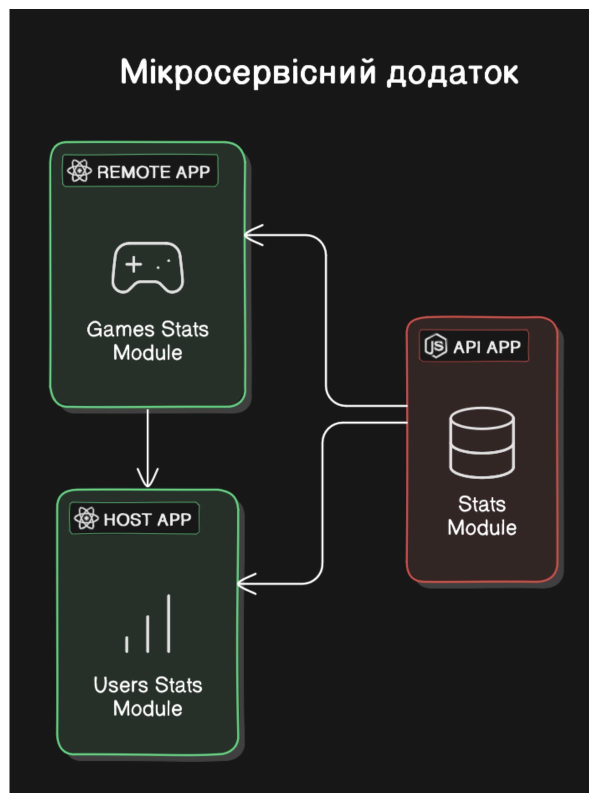


Рисунок 2.2 – Схема зв’язків мікросервісного додатку

Бекенд, у свою чергу, буде просто монолітним. У контексті проєкту маємо характерну особливість – роботу на базі монорепозиторію. Це підхід до організації розробки програмного забезпечення, коли всі проєкти знаходяться в одному централізованому репозиторії. Він дуже доречний для розвитку комплексних програмних продуктів, що складаються з багатьох взаємозалежних частин. Вибір монорепозиторію як основного організаційного підходу для розробки монолітного додатка здійснено з метою оптимізації та спрощення процесу розроблення. Монорепозиторій дозволяє зберігати всі додатки в одному місці, що полегшує спільну розробку та управління залежностями між ними. Усі

вище описані додатки знаходяться в одному Git-репозиторії, згруповані відповідно до їхньої функціональності для полегшення навігації.

Особливість проєкту полягає в тому, що ми вирішили застосувати монорепозиторій не лише для бекенду, а й для фронтенду. Тобто використовуємо монорепозиторій для керування всіма додатками та компонентами на клієнтській і серверній сторонах. Загальна структура монорепозиторію включає в себе різні проєкти та компоненти, розділені на папки. Крім того, ми використовуємо глобальний конфігураційний файл для керування всіма «пакетами» в монорепозиторії, що забезпечує консистентність та однорідність у всьому проєкті. Цей підхід дає змогу забезпечити ефективну та зручну розробку програмного забезпечення, а також сприяє спільному управлінню та координації роботи між різними частинами системи. Отже, маючи два додатки зі спільною функціоналністю та подібними технологіями, ми можемо розділити все на пакети, які потім перевикористаємо. Перелічимо ці пакети:

1) config: містить конфігураційні файли, які використовуються у всіх додатках у монорепозиторії. Він включає в себе налаштування середовища, змінні середовища та інші глобальні налаштування, які можуть знадобитися для конфігурації окремих частин проекту.

2) eslint: містить налаштування ESLint, яке використовується для статичного аналізу коду та виявлення потенційних проблем. Він забезпечує консистентність коду та допомагає уникнути типових помилок під час розробки.

3) web-shared: містить загальні компоненти, стилі та ресурси, які використовуються у всіх клієнтських додатках. Він дає змогу зберігати та перевикористовувати спільний код та компоненти між різними фронтендами.

4) tailwind: містить конфігураційні файли та ресурси, які використовуються для налаштування та кастомізації фреймворку Tailwind CSS. Він дає можливість забезпечити однорідний стиль та вигляд усіх клієнтських додатків.

5) api-shared: містить загальні сервіси, утиліти та ресурси, які використовуються у всіх серверних додатках. Забезпечує можливість зберігати

та перевикористовувати логіку та функціональність, яка використовується в різних сервісах.

6) prisma: містить моделі та конфігураційні файли, які використовуються для роботи з базою даних за допомогою ORM Prisma. Він дозволяє забезпечити єдиний інтерфейс для взаємодії з базою даних у всіх серверних додатках.

7) ts-config: містить конфігураційні файли для TypeScript, які використовуються у всіх проектах у монорепозиторії. Він встановлює налаштування для компіляції коду TypeScript та допомагає забезпечити типобезпечність та консистентність у всьому проєкті.

2.2. Інструментальні засоби програмної реалізації

Розглянемо інструментальні засоби, які використовуються для програмної реалізації проєктів. Для монолітного вебдодатка застосовуються такі бібліотеки:

1) Prisma. Це ORM-пакет, який забезпечує зручний спосіб взаємодії з базою даних. Prisma дає можливість використовувати мову програмування для роботи з базою даних замість SQL запитів.

2) trpc. Ця бібліотека використовується для створення типобезпечних RPC- служб. Вона дає змогу визначати RPC (Remote Procedure Call) методи та їх типи даних за допомогою TypeScript.

3) React Hook Form. Дозволяє легко управляти формами в React за допомогою хуків. Вона надає зручний API для валідації та збереження даних форми.

4) zod. Бібліотека для валідації даних у JavaScript та TypeScript. Вона надає можливість визначати схеми даних та перевіряти, чи відповідають дані цим схемам.

5) Tailwind CSS. Це CSS-фреймворк, який дає змогу швидко створювати стильні й адаптивні вебінтерфейси. Він відрізняється від інших фреймворків тим, що не має готових компонентів, а замість цього пропонує набір універсальних класів для створення інтерфейсу.

Використання цих бібліотек допоможе спростити розробку та забезпечити швидке виконання функціональних вимог монолітного додатку.

У межах мікросервісного підходу аналогічно застосовується бібліотека trpc як на рівні фронтенда, так і на рівні бекенда. Також на фронтенді повторно використовуємо фреймворк Tailwind CSS, а на бекенді – бібліотеку zod. Основною технологією для імплементації фронтенду є бібліотека React, яка дає можливість розробникам будувати великі та складні вебдодатки з використанням компонентного підходу. Вона відома своєю ефективністю, зручністю використання та широкою спільнотою. Для прискорення розробки також застосовуємо програмний інструмент Vite. Він працює на основі сучасних стандартів, таких як ES Modules, та дає змогу швидко збирати та запускати проєкти.

На рівні бекенда використовуємо фреймворк Express. Він є популярним засобом для створення серверних додатків на Node.js та за допомогою простого й зручного API надає можливості швидко створювати мережеві додатки.

Для управління monorepo було обрано інструмент turborepo. Він використовується для керування монорепозиторіями, що дозволяє розробникам працювати з різними частинами великого проекту у єдиному середовищі. В контексті нашого проекту, використання TurboRepo має наступні переваги:

1) Ефективність збірки та тестування. TurboRepo дає змогу розпаралелити процеси збірки та тестування для кожної частини вашого проекту, що, в свою чергу, допомагає зменшити час збірки та тестування. Це особливо важливо для великих монорепозиторіїв.

2) Зручне середовище розробки. Використання TurboRepo надає розробникам можливості працювати з різними частинами проекту у єдиному середовищі розробки, що спрощує спільну розробку та спілкування між членами команди.

3) Керування залежностями. За допомогою TurboRepo ви можете керувати залежностями між різними частинами проекту та забезпечити їх

консистентність. Це допомагає уникнути конфліктів версій та забезпечити стабільність проєкту.

4) Швидке впровадження змін. TurboRepo дозволяє швидко впроваджувати зміни у всіх частинах проєкту одночасно, що зменшує час від розробки до випуску нової функціональності та полегшує процес релізів.

Розглядаючи аспекти бази даних у проєкті, було обрано PostgreSQL в поєднанні з ORM-інструментом Prisma. Це рішення є стратегічним, оскільки дає змогу ефективно керувати даними та забезпечує гнучкість у розвитку додатку. Бібліотека Prisma, розміщена в окремому пакеті, забезпечує можливість перевикористовувати цей інструмент у всіх частинах проєкту, впроваджуючи консистентність та однорідність у взаємодії з базою даних. PostgreSQL, як надійна реляційна база даних, гарантує стабільність та високу продуктивність додатку. Її розширена функціональність дає змогу ефективно керувати даними та забезпечує їхню безпеку.

Загальна схема залежностей в проєкті продемонстрована на рис. 2.3. Варто зазначити, що для вебдодатків характерне обширне використання бібліотечних засобів та фреймворків, тому з кута зору архітектури це окреме важливе питання, яке суттєво впливає на атрибути якості програмного забезпечення.

2.3. Детальний опис архітектури додатків

Додаток має дві сторінки – статистика користувачів і статистика ігор. Для користувача ці сторінки виглядають так, як показано на рис. 2.4. Взаємодія користувача і додатку візуалізована на рис. 2.5. Загальна логіка полягає в клієнт-серверній взаємодії, що й відображено на рисунку. Враховуючи суто демонстраційну природу розроблених вебдодатків, було обрано для імплементування мінімально життєздатну функціональність сторінок.

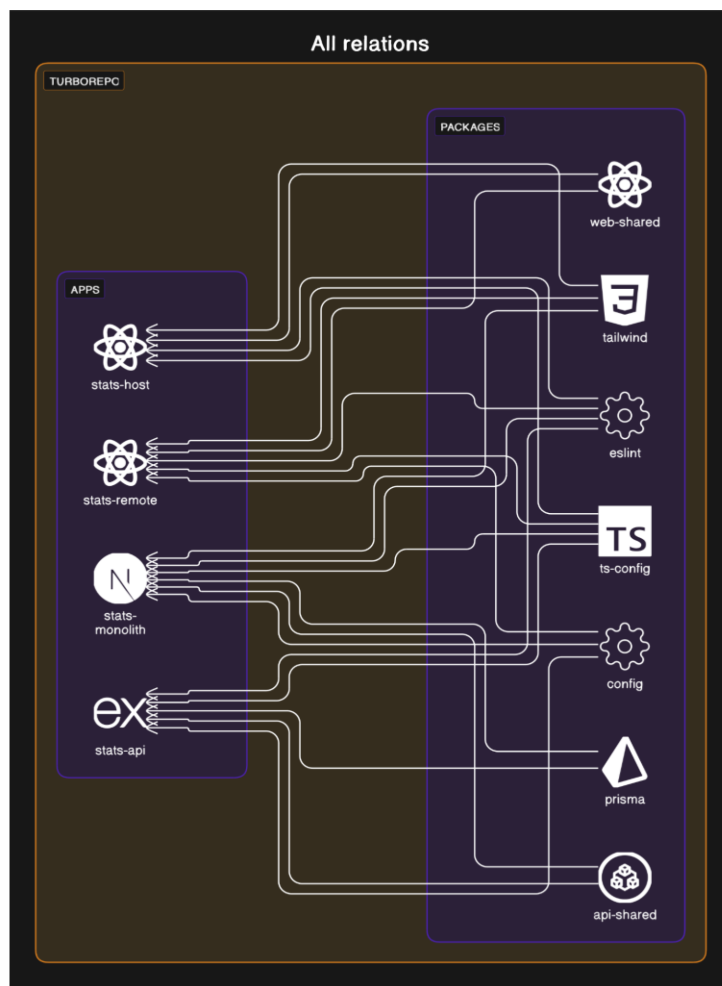


Рисунок 2.3 – Схема зв'язків проєкту

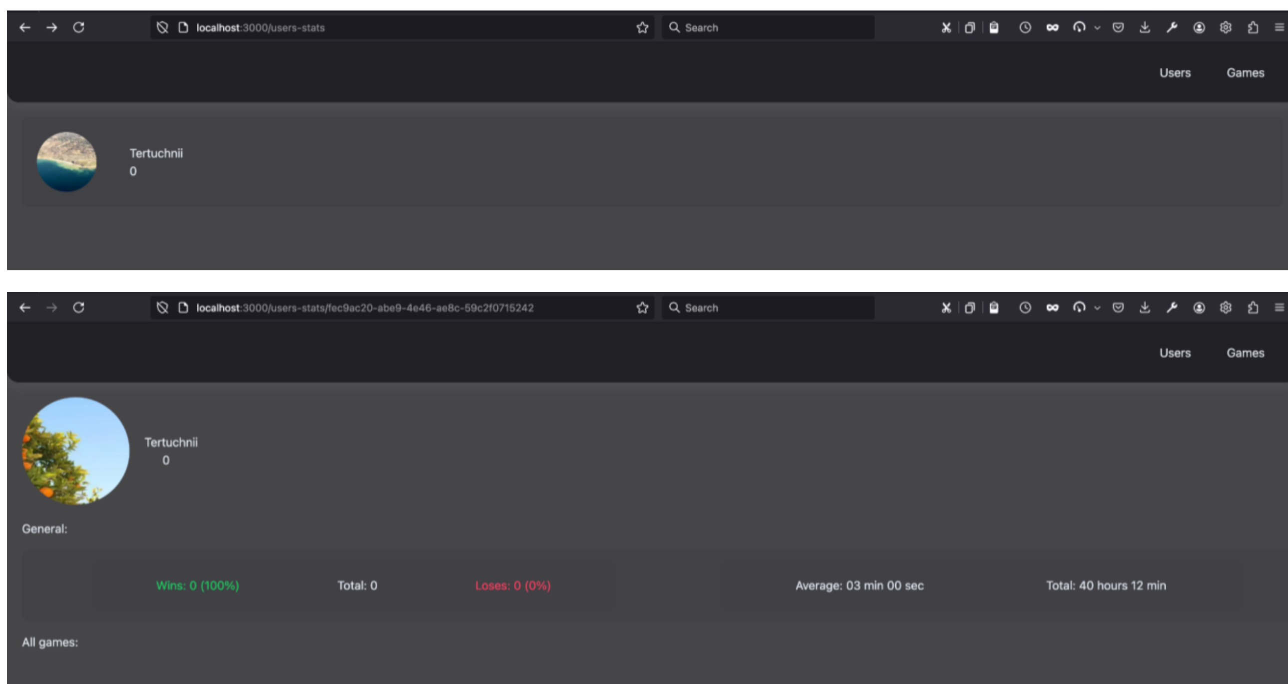
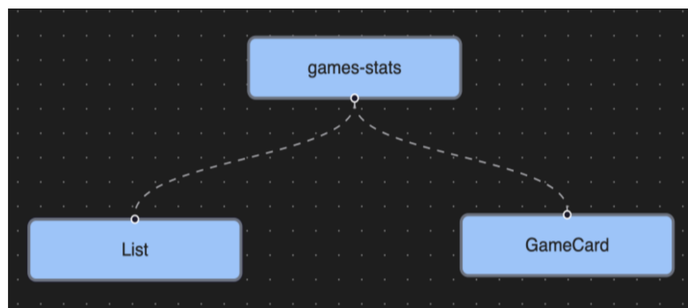


Рисунок 2.4 – Вигляд сторінок додатку

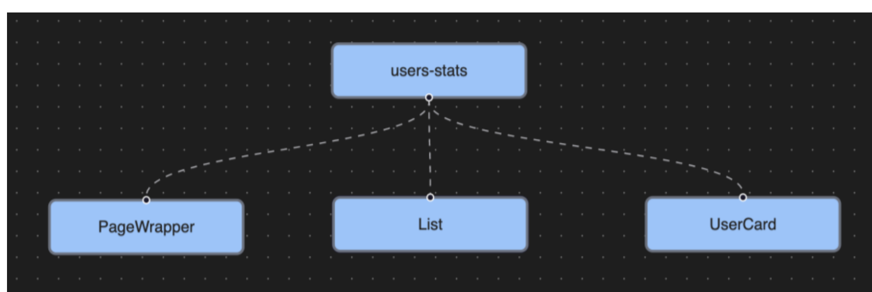
(а) (б)

Рисунок 2.5 – Діаграми послідовності взаємодії користувача з а) сторінкою користувачів; б) сторінкою ігор

За таких умов зв'язки компонентів сторінки статистики ігор та користувачів описано на рис. 2.6.



(а)



(б)

Рисунок 2.6 – Схема компонентів а) сторінки ігор; б) сторінки користувачів

Наявні такі компоненти:

– PageWrapper –адаптивний компонент, що додає основні компоненти сторінки, які є всюди: навігацію, обгортки над контентом сторінки, базові стилі сторінки тощо;

– List –адаптивний компонент списку. Він сприймає масив даних і компонент, що відрисовується на кожний елемент масиву. Також він додає базові стилі, функціональність дозавантаження даних при гортанні сторінки, відображення стану завантаження даних, або помилки при їх отриманні;

– GameCard –компонент, що відображає статистику однієї гри у вигляді картки. Він сприймає об’єкт ігрової статистики і відрисовує всю потрібну користувачу інформацію, використовується у компоненті List;

– UserCard –компонент, що відображає статистику одного гравця у вигляді картки, він сприймає об’єкт статистики гравця й відрисовує всю потрібну користувачу інформацію, використовується у компоненті List;

– games-stats – модуль, що поєднує всі наші компоненти у повноцінну сторінку статистики ігор;

– users-stats – модуль, що поєднує всі наші компоненти у повноцінну сторінку статистики гравців.

У той же час, кінцеві точки, які надає наш бекенд реалізовуються як набір функцій, як показано на схемі кінцевих точок, наведеній на рис. 2.7:

– getUserById(id) приймає ідентифікатор користувача (id) і повертає інформацію про користувача з вказаним ідентифікатором;

– getUsers() повертає список всіх користувачів у системі разом з їхньою інформацією;

– getUserRatio(userId) повертає співвідношення певного показника для користувача з вказаним userId. Наприклад, це може бути співвідношення перемог до поразок у іграх;

– getUserGamesDuration(userId) повертає тривалість ігор для конкретного користувача (userId), можливо, у загальному або середньому значенні;

– getUserLongestAndShortestGames(userId) повертає найдовшу та найкоротшу гру користувача (userId), можливо, з інформацією про їх тривалість чи іншими параметрами;

– getUserGamesStats(userId) надає статистику про ігри користувача (userId), таку як загальна кількість ігор, середня тривалість, середній рівень тощо;

– getGamesStats() повертає загальну статистику про всі ігри у системі, можливо, кількість гравців, середній час гри, загальна кількість ігор тощо.

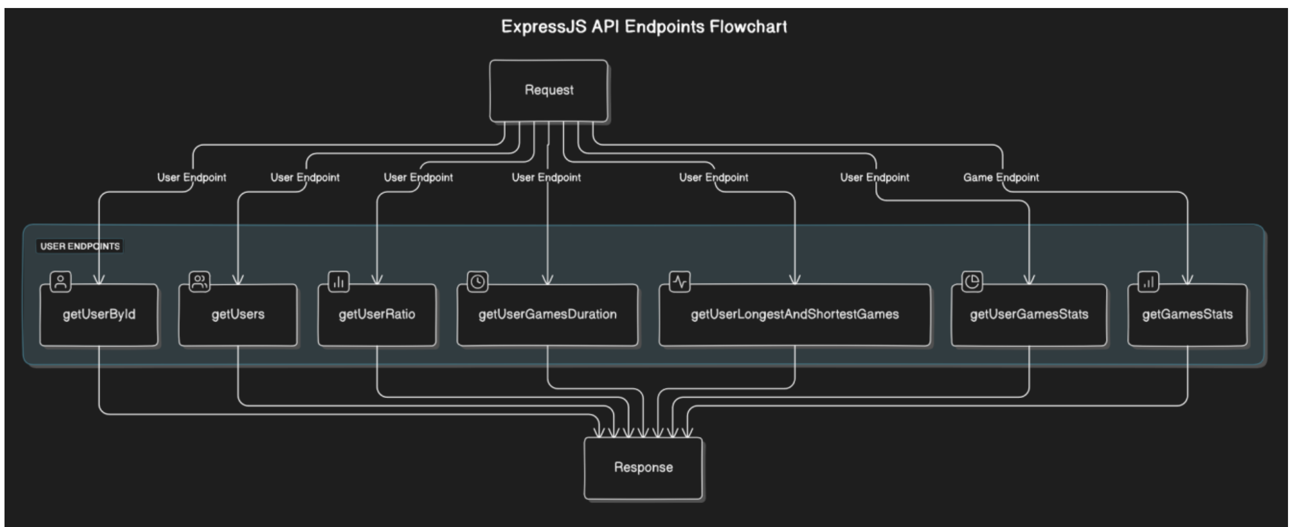


Рисунок 2.7 – Схема кінцевих точок бекенду

У результаті розгляду архітектурних питань побудови демонстраційних додатків для подальшої конвеєризації постачання було визначено основні компоненти, інструменти та технології, що лягають в основу процесу розробки. Отримані працездатні зразки програмного забезпечення виступають базою для подальших комп’ютерних експериментів стосовно автоматизації постачання програмного коду.

РОЗДІЛ 3
ВПРОВАДЖЕННЯ КОНВЕЄРА ПОСТАЧАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У цьому розділі розглянемо ключові етапи побудови конвеєра програмного забезпечення, починаючи з інтеграції коду за допомогою GitHub Actions, упаковки коду в Docker-контейнери, розгортання програмного забезпечення, моніторингу системи, а також перспективи подальшого удосконалення.

3.1. GitHub Actions як інструмент неперервної інтеграції коду

Основні принципи роботи з GitHub Actions включають:

– Тригери. Дії запускаються за допомогою тригерів, таких як push до гілки, створення pull request або за розкладом.

– Робочі процеси (Workflows). Це набори завдань, які виконуються у певній послідовності. Визначаються у файлах з розширенням .yaml, що розміщуються в каталозі .github/workflows вашого репозиторію.

– Завдання (Jobs). Кожен робочий процес складається з одного або більше завдань, які виконуються на окремих віртуальних машинах.

– Кроки (Steps): Кожне завдання складається з кроків, які можуть включати виконання команд, запуск скриптів або використання дій з маркетплейсу GitHub Actions [2].

Візуалізація відповідності між компонентами GitHub Actions наведена на рис. 3.1. У такому ж стилі формуються .yaml-файли, які відтворюють структуру дій та визначають набори команд для виконання на рівні окремих компонентів.

Традиційно при розгортанні додатків будується декілька середовищ, які використовуються по ходу роботи конвеєра постачання програмного забезпечення. У межах розробки CI/CD-процесу для проєкту було вирішено створити два окремих скрипти для середовища розробки (dev) та виробничого середовища (production) відповідно.

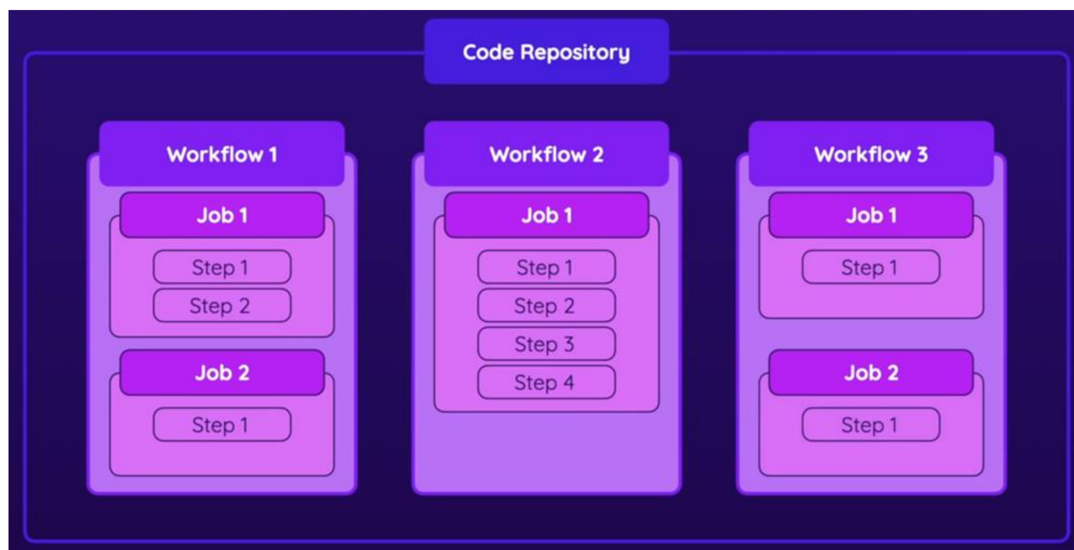


Рисунок 3.1 – Компоненти GitHub Actions

YAML-файл, розроблений для автоматизації робочих процесів у середовищі розробки, наведено в лістингу 3.1. Розпочинаємо з налаштування тригерів для запуску робочого процесу при створенні pull request або push-операції до будь-якої гілки, окрім master. Це дає змогу ізолювати процес розробки від основної (продуктової) гілки, зосереджуючи CI/CD на тестуванні змін у середовищі розробки.

Лістинг 3.1 – Скрипт для середовища розробки (dev)

```
name: dev
on:
 pull_request:
   branches:
     - "!master"
     - "**"
   branches-ignore:
     - master
 push:
   branches:
     - "!master"
     - "**"
   branches-ignore:
     - master
jobs:
 build:
   name: Build and Test
   runs-on: ubuntu-latest
   steps:
     - name: Checkout
       uses: actions/checkout@v3
```

```
    - name: Install Node.js
      uses: actions/setup-node@v3
      with:
        node-version: 18
    - uses: pnpm/action-setup@v2
      name: Install pnpm
      id: pnpm-install
      with:
        version: 7
        run_install: false
    - name: Get pnpm store directory
      id: pnpm-cache
      shell: bash
      run: |
        echo "STORE_PATH=$(pnpm store path)" >> $GITHUB_OUTPUT
    - uses: actions/cache@v3
      name: Setup pnpm cache
      with:
        path: ${{ steps.pnpm-cache.outputs.STORE_PATH }}
        key: ${{ runner.os }}-pnpm-store-${{ hashFiles('**/pnpm-lock.yaml') }}
        restore-keys: |
          ${{ runner.os }}-pnpm-store-
    - name: Install dependencies
      run: pnpm install
    - name: Linters
      run: pnpm lint
    - name: Build
      run: pnpm build
```

Для середовища розробки основним завданням виступає збирання проєкту, тому завдання build розпочинається з налаштування середовища розгортання на базі останньої стабільної версії Ubuntu (ubuntu-latest). У межах завдання маємо такі кроки:

– Checkout. Завантаження коду з репозиторію для подальшої обробки.

– Install Node.js. Встановлення необхідної версії Node.js для виконання проєкту.

– Install pnpm. Встановлення pnpm (менеджера пакетів), який використовується для управління залежностями в проєкті.

– Get pnpm store directory. Визначення директорії кешу pnpm для прискорення побудови подальших збірок.

– Setup pnpm cache. Налаштування кешування для зменшення часу встановлення залежностей.

– Install dependencies. Встановлення залежностей проєкту.

– Linters. Виконання лінтерів для перевірки коду на наявність статичних помилок та відповідність стилю.

– Build. Збірка проєкту.

Для виробничого середовища скрипт фактично буде доповненням попереднього. Початкові кроки такі ж, як у dev: інсталювати залежності, виконати базовий статичний аналіз коду, виконати процес збирання. Наступним завданням є виконання розгортання. Залежно від архітектурного стилю, деплой проєкту має свої відмінності. Уривок скрипта для розгортання монолітної версії додатка наведено в лістингу 3.2. У ньому сформовані такі кроки:

– Set up QEMU. Налаштовує емулятор QEMU, який надає можливість запускати багатоплатформні збірки за допомогою Docker.

– Set up Docker Buildx. Налаштовує Docker Buildx, який дає змогу кросплатформного збирання з використанням розширених функцій Docker.

– Log in to Amazon ECR. Цей крок авторизує розробника в сервісі Amazon ECR (Elastic Container Registry) в зазначеному регіоні (us-west-2).

– Build, tag, and push image to ECR. Виконує збірку Docker-образу, тегує його за допомогою гешу коміту (github.sha), а потім пушить його до вашого ECR репозиторію.

– Deploy to ECS. Примусово оновлює сервіс в Amazon ECS, розгортаючись на базі нового Docker-образа. Використовує AWS CLI.

У межах скрипту 3.2 маємо такі змінні:

– secrets.AWS_ACCOUNT_ID – ідентифікатор вашого AWS акаунта, який зберігається в секретах репозиторію;

– region – регіон AWS, де знаходяться ваші ресурси;

– repository_name – назва вашого репозиторію в Amazon ECR;

– github.sha – геш коміту, який використовується для тегування Docker-образу;

– CLUSTER_NAME – назва вашого ECS кластеру;

– SERVICE_NAME – назва вашого ECS сервісу;

– CONTAINER_NAME – назва вашого ECS контейнера.

Лістинг 3.2 – Уривок скрипту з розгортанням монолітного додатка

```
- name: Set up QEMU
uses: docker/setup-qemu-action@v2
- name: Set up Docker Buildx
uses: docker/setup-buildx-action@v2
- name: Log in to Amazon ECR
id: login-ecr
uses: aws-actions/amazon-ecr-login@v1
with:
  region: us-west-2  # Вкажіть свій регіон
- name: Build, tag, and push image to ECR
env:
  ECR_REGISTRY: ${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.us-west-2.amazonaws.com
  ECR_REPOSITORY: repository_name  # Замініть на ваш ECR репозиторій
  IMAGE_TAG: ${{ github.sha }}
run: |
  docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
  docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
- name: Deploy to ECS
env:
  AWS_REGION: us-west-2  # Вкажіть свій регіон
  CLUSTER_NAME: my-cluster  # Замініть на ваш кластер ECS
  SERVICE_NAME: my-service  # Замініть на ваш сервіс ECS
  CONTAINER_NAME: my-container  # Замініть на ваш контейнер ECS
  ECR_IMAGE: ${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.us-west-
2.amazonaws.com/repository_name:${{ github.sha }}
run: |
  aws ecs update-service --cluster $CLUSTER_NAME --service $SERVICE_NAME --force-
new-deployment --region $AWS_REGION --desired-count 1 --output json
```

Відповідна версія скрипта для мікросервісної реалізації продемонстрована в лістингу 3.3. Структурно цей уривок дуже подібний до лістингу 3.2, проте, зокрема, також включає налаштування AWS credentials, щоб мати можливість працювати з AWS CLI та іншими інструментами AWS, та застосування технології Kubernetes з метою оркестрації образів.

Крок “Update Kubernetes deployment” оновлює деплоймент в Kubernetes (у вашому Amazon EKS кластері) з новим Docker-образом. В ході цього використовується AWS CLI для оновлення конфігурації EKS і kubectl для зміни образу контейнера.

Лістинг 3.3 – Уривок скрипта з розгортанням додатка на основі мікросервісів

```
    - name: Set up QEMU
      uses: docker/setup-qemu-action@v2
    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2
    - name: Log in to Amazon ECR
      id: login-ecr
      uses: aws-actions/amazon-ecr-login@v1
      with:
        region: us-west-2  # Вкажіть свій регіон
    - name: Build, tag, and push image to ECR
      env:
        ECR_REGISTRY: ${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.us-west-2.amazonaws.com
        ECR_REPOSITORY: repository_name  # Замініть на ваш ECR репозиторій
        IMAGE_TAG: ${{ github.sha }}
      run: |
        docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
        docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
    - name: Configure AWS credentials
      uses: aws-actions/configure-aws-credentials@v1
      with:
        aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
        aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
        aws-region: us-west-2  # Вкажіть свій регіон
    - name: Update Kubernetes deployment
      env:
        EKS_CLUSTER_NAME: my-cluster  # Замініть на ваш EKS кластер
        KUBE_DEPLOYMENT_NAME: my-deployment  # Замініть на вашу назву деплойменту
Kubernetes
        CONTAINER_NAME: my-container  # Замініть на ваше ім’я контейнеру
        ECR_IMAGE: ${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.us-west-
2.amazonaws.com/repository_name:${{ github.sha }}
      run: |
        aws eks update-kubeconfig --name $EKS_CLUSTER_NAME --region us-west-2
        kubectl set image deployment/$KUBE_DEPLOYMENT_NAME
$CONTAINER_NAME=$ECR_IMAGE
```

Задіяні в цьому процесі змінні такі:

– secrets.AWS_ACCOUNT_ID – ідентифікатор вашого облікового запису AWS, який зберігається в секретах репозиторію;

– region – регіон AWS, де знаходяться ваші ресурси;

– repository_name – назва вашого репозиторію в Amazon ECR;

– github.sha – геш коміту, який використовується для тегування Docker-образу;

– CLUSTER_NAME – назва вашого EKS кластеру;

– KUBE_DEPLOYMENT_NAME – назва вашого деплойменту в Kubernetes;

– CONTAINER_NAME – назва контейнера, який необхідно оновити в Kubernetes.

3.2. Упаковка програмного коду

Платформа Docker застосовується для розробки, доставки та запуску застосунків у контейнерах. Контейнери дають змогу ізолювати застосунки від середовища, у якому вони працюють, що робить їх більш портативними та легкими для розгортання. Основні компоненти Docker:

– Dockerfile – текстовий файл з інструкціями для створення Docker-образу;

– Docker-образ – непорушний файл, що містить усі залежності та конфігурації для запуску застосунку;

– Docker-контейнер – виконавчий екземпляр образу.

Поетапно опишемо конфігураційні файли, які включають Docker до процесу постачання програмного забезпечення. Файл Dockerfile.api використовується для побудови контейнера, який містить бекенд додаток, готовий для використання у середовищі Docker. Лістинг 3.4 відображає вміст згаданого файлу.

Спочатку визначаємо, що в якості базового для побудови контейнера використовується офіційний образ node:18-alpine. Alpine Linux – легковаговий Linux-дистрибутив, який часто використовують для контейнеризації. На наступному етапі відбувається підготовка середовища (Setup Environment): визначається новий етап setup на основі базового образу, виконуються команди для встановлення пакетів, необхідних для подальшої роботи в контейнері.

Побудова проєкту (Build Project) передбачає визначення робочої директорії і копіювання в неї вихідних файлів проєкту. Тако виконується

встановлення пакету turbo та очистка зайвих файлів і залежностей за допомогою команди turbo prune.

Лістинг 3.4 – Файл Dockerfile.api

```
FROM node:18-alpine AS base
FROM base AS setup
RUN apk add --no-cache libc6-compat
RUN apk update
WORKDIR /app
RUN npm install -g turbo
COPY . .
RUN turbo prune --scope=@stats/api --docker
FROM base AS build
RUN apk add --no-cache libc6-compat
RUN apk update
WORKDIR /app
RUN npm install -g pnpm
COPY .gitignore .gitignore
COPY --from=setup /app/out/json/ .
COPY --from=setup /app/out/pnpm-lock.yaml ./pnpm-lock.yaml
COPY --from=setup /app/out/full/ .
RUN pnpm install
RUN npx turbo run build --filter=@stats/api...
FROM base AS runner
WORKDIR /app
COPY --from=build /app ./
EXPOSE 5530
CMD cd ./packages/prisma && npm run pushDB && cd ../../apps/stats-api && npm run start
```

Збирання проєкту (Project Build) визначено як етап build на основі етапу base. Виконуються аналогічні кроки з налаштування середовища, проте з інсталяцією пакету pnpm, копіюються необхідні файли з попереднього етапу (у цьому випадку з етапу setup). Також виконується встановлення залежностей за допомогою pnpm install і здійснюється збирання проєкту з використанням turbo.

Запуск додатку (Application Run) – це останній етап runner, у ході якого налаштовуємо робочу директорію та копіюємо збірку з попереднього етапу build. Використовується команда EXPOSE, щоб вказати порт, на якому працює додаток. Команда CMD визначає команду, яка виконається при запуску контейнера. У нашому випадку це команда для запуску додатка, яка складається з двох частин: команди для роботи з базою даних (Prisma), та запуску власне API додатка.

Файл Dockerfile.host використовується для побудови контейнера, який містить Host-додаток, готовий для використання у середовищі Docker. Його вміст показано в лістингу 3.5.

Лістинг 3.5 – Вміст файлу Dockerfile.host

```
FROM node:18-alpine AS base
FROM base AS setup
RUN apk add --no-cache libc6-compat
RUN apk update
WORKDIR /app
RUN npm install -g turbo
COPY . .
RUN turbo prune --scope=@stats/host --docker
FROM base AS build
RUN apk add --no-cache libc6-compat
RUN apk update
WORKDIR /app
RUN npm install -g pnpm
COPY .gitignore .gitignore
COPY --from=setup /app/out/json/ .
COPY --from=setup /app/out/pnpm-lock.yaml ./pnpm-lock.yaml
COPY --from=setup /app/out/full/ .
RUN pnpm install
RUN npx turbo run build --filter=@stats/host...
FROM nginx:1.23.1-alpine AS server
WORKDIR /etc/nginx/html
RUN rm -rf ./*
COPY --from=build ./app ./app
COPY --from=build ./app/apps/stats-host/nginx.conf /etc/nginx/nginx.conf
COPY --from=build ./app/apps/stats-host/dist .
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Все, окрім останнього кроку, спільне з попереднім файлом, проте також змінились назви. Тут відбувається запуск сервера Nginx (Running Nginx Server): встановлюється базовий образ nginx:1.23.1-alpine для сервера Nginx, визначається робоча директорія для Nginx, видаляються старі файли з робочої директорії Nginx, копіюються зібрані файли з фронтенду та конфігураційний файл Nginx, відкривається порт 80 для доступу до сервера, визначається команда для запуску Nginx у фоновому режимі.

Файл Dockerfile.remote використовується для побудови контейнера, який містить мікросервіс статистики гравці, готовий для використання у середовищі Docker. Загалом, це копія хосту тільки із зміненими іменами (лістинг 3.6).

Лістинг 3.6 – Вміст файлу Dockerfile.remote

```
FROM node:18-alpine AS base
FROM base AS setup
RUN apk add --no-cache libc6-compat
RUN apk update
WORKDIR /app
RUN npm install -g turbo
COPY . .
RUN turbo prune --scope=@stats/remote --docker
FROM base AS build
RUN apk add --no-cache libc6-compat
RUN apk update
WORKDIR /app
RUN npm install -g pnpm
COPY .gitignore .gitignore
COPY --from=setup /app/out/json/ .
COPY --from=setup /app/out/pnpm-lock.yaml ./pnpm-lock.yaml
COPY --from=setup /app/out/full/ .
RUN pnpm install
RUN npx turbo run build --filter=@stats/remote...
FROM nginx:1.23.1-alpine AS server
WORKDIR /etc/nginx/html
RUN rm -rf ./*
COPY --from=build ./app ./app
COPY --from=build ./app/apps/stats-remote/nginx.conf /etc/nginx/nginx.conf
COPY --from=build ./app/apps/stats-remote/dist .
EXPOSE 80
ENTRYPOINT ["nginx", "-g", "daemon off;"]
```

Файл Dockerfile.monolith застосовується для побудови контейнера, який містить Monolith додаток, готовий для використання у середовищі Docker. Відповідний вміст наведено в лістингу 3.7.

Тут всі базові кроки теж повторюються, крім останнього. Запуск Next.js сервера (Running Next.js Server) визначає робочу директорію та копіює збірку з попереднього етапу installer. Також встановлюються обмеження на запуск процесу в якості користувача nextjs, а не як користувач root, копіюються файли збірки та статичні файли, необхідні для роботи Next.js додатку. Після цього визначається команда для запуску Next.js сервера.

Вище згадані скрипти запускаються в виробничому середовищі, що формує повну картину роботи конвеєра постачання програмного забезпечення. Також конвеєр містить команди для автоматизованого запуску та використання бази даних і менеджера баз даних PostgreSQL.

Лістинг 3.7 – Вміст файлу Dockerfile.monolith

```
FROM node:18-alpine AS base
FROM base AS builder
RUN apk add --no-cache libc6-compat
RUN apk update
WORKDIR /app
RUN npm install -g turbo
COPY . .
RUN turbo prune --scope=@stats/monolith --docker
FROM base AS installer
RUN apk add --no-cache libc6-compat
RUN apk update
WORKDIR /app
RUN npm install -g pnpm
COPY .gitignore .gitignore
COPY --from=builder /app/out/json/ .
COPY --from=builder /app/out/pnpm-lock.yaml ./pnpm-lock.yaml
COPY --from=builder /app/out/full/ .
RUN pnpm install
RUN npx turbo run build --filter=@stats/monolith...
FROM base AS runner
WORKDIR /app
RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nextjs
USER nextjs
COPY --from=installer /app ./
COPY --from=installer --chown=nextjs:nodejs /app/apps/stats-monolith/.next/standalone ./
COPY --from=installer --chown=nextjs:nodejs /app/apps/stats-monolith/.next/static ./apps/stats-monolith/.next/static
COPY --from=installer --chown=nextjs:nodejs /app/apps/stats-monolith/public ./apps/stats-monolith/public
EXPOSE 80
CMD cd ./packages/prisma && npm run pushDB && node ../../apps/stats-monolith/server.js
```

3.3. Моніторинг проєкта

Моніторинг забезпечує контроль за станом програмного забезпечення в виробничому середовищі. Це включає :

– збір та відстеження метрик у вигляді графіків для аналізу продуктивності;

– налаштування тривог (alerts), які сповіщають вас про певні події або значення метрик, що перевищують визначені порогові значення;

– збирання, зберігання та аналіз журналів (logs) з різних джерел;

– відстеження подій та автоматичне реагування на них (наприклад, запуск AWS Lambda функції або активація SNS сповіщення);

– створення дашбордів для візуалізації стану та продуктивності ваших ресурсів у реальному часі.

Аналізуючи метрики та журнали, можна ідентифікувати вузькі місця в продуктивності і оптимізувати ресурси. Сповіщення про проблеми в режимі реального часу допомагають швидко реагувати на неполадки. З кута зору безпеки, можна здійснювати аналіз журналів для виявлення аномалій та потенційних загроз.

При створені додатків AWS автоматично додає моніторинг через CloudWatch. Завдяки цьому отримані моніторинг і логування для наших додатків виглядають так, як показано на рисунках 3.2-3.4. Наприклад, на рисунку 3.2 бачимо логи розробленого та розгорнутого бекенду, який відображає, які саме SQL-запити ми робимо до бази даних.

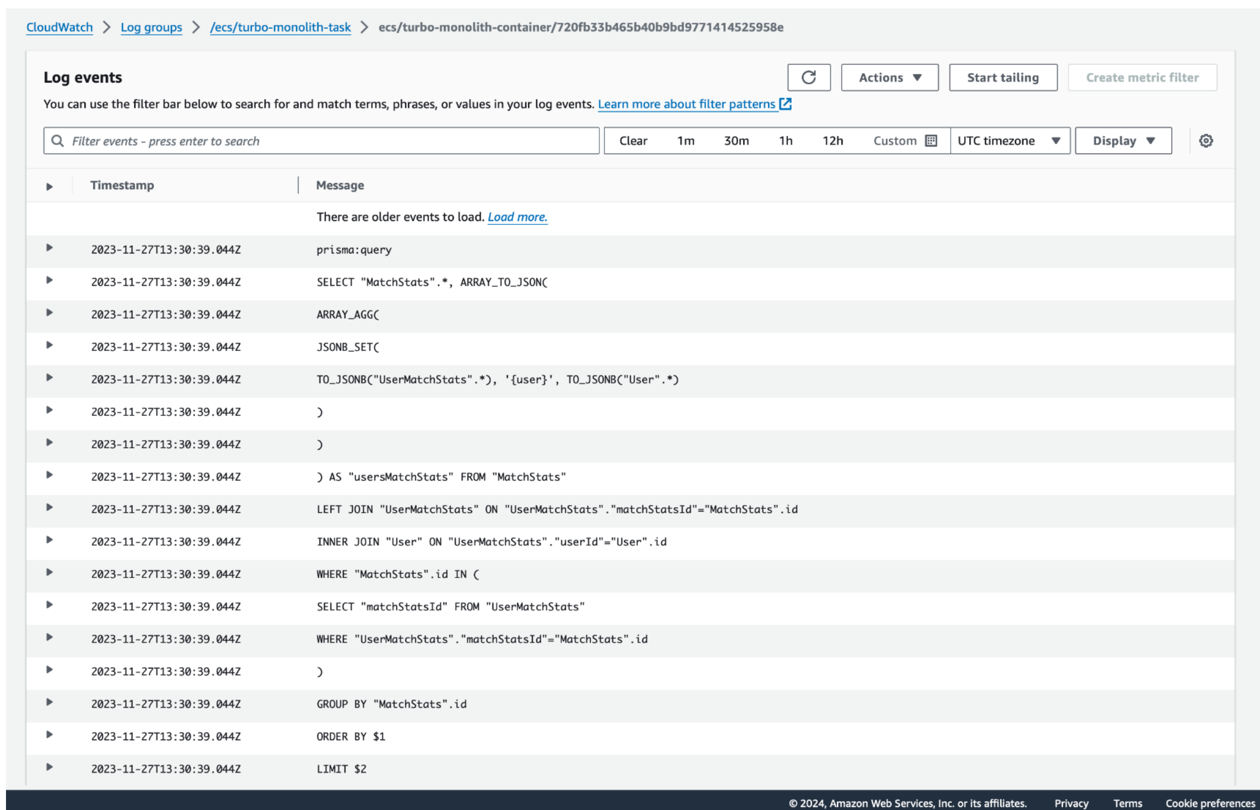


Рисунок 3.2 – Логи проєкту

Решта рисунків демонструють результати групування журналів сервісом CloudWatch та використовувані ним метрики відповідно.

Log streams | Tags | Anomaly detection | Metric filters | Subscription filters | Contributor Insights | Data protection

Log streams (15)

Delete | Create log stream | Search all log streams

Filter log streams or try prefix search

Exact match | Show expired | Info

Log stream	Last event time
ecs/turbo-monolith-container/720fb33b465b40b9bd9771414525958e	2023-11-27 18:49:15 (UTC)
ecs/turbo-monolith-container/c586a71209b44a0f860a6a673cce428f	2023-11-03 07:46:26 (UTC)
ecs/turbo-monolith-container/588070443e8d4b22a26edb7c584bb0f1	2023-11-02 21:29:16 (UTC)
ecs/turbo-monolith-container/a21c2bb00c2046fe97eca037212b88fb	2023-11-02 20:10:57 (UTC)
ecs/turbo-monolith-container/ce717fb6c64b494a993459ec4857a6e6	2023-11-02 19:57:01 (UTC)
ecs/turbo-monolith-container/2b0871bcf22b4e83ac7ffd0824d108d2	2023-11-02 19:56:39 (UTC)
ecs/turbo-monolith-container/7da42c75142044faa83044266375c653	2023-11-02 19:56:21 (UTC)
ecs/turbo-monolith-container/e647bb1e8afb475aacaa3992bf2c54f5	2023-11-02 19:55:57 (UTC)
ecs/turbo-monolith-container/9f5dbaff293049279509000884f7e2ce	2023-11-02 19:55:40 (UTC)
ecs/turbo-monolith-container/9132e0e25beb4211b59df5f97cabf7cb	2023-11-02 19:55:16 (UTC)
ecs/turbo-monolith-container/c2cc0fd935104fb89c90ce77df4bd3c3	2023-11-02 19:54:28 (UTC)
ecs/turbo-monolith-container/8ff7d5efb8b1423f968cb389d32ec403	2023-11-02 19:54:02 (UTC)
ecs/turbo-monolith-container/8171b871b65846398ad573a4986a299a	2023-11-02 19:53:43 (UTC)
ecs/turbo-monolith-container/c649e7a372e5431699149904b53f94fe	2023-11-02 19:53:25 (UTC)
ecs/turbo-monolith-container/f28e585b2b8147a28054824a9eb463cd	2023-11-02 19:53:03 (UTC)

© 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie prefer

Рисунок 3.3 – Список груп логів

CloudWatch > Metrics

Untitled graph

1h 3h 12h 1d 3d 1w Custom | UTC timezone | Actions | Line

Your CloudWatch graph is empty.
Select some metrics to appear here.

1 0.5 0

13:00 13:15 13:30 13:45 14:00 14:15 14:30 14:45 15:00 15:15 15:30 15:45

Browse | Multi source query | Graphed metrics | Options | Source | Add math | Add query

CloudWatch	GetMetricData	API	None	CallCount	No alarms
CloudWatch	GetDashboard	API	None	CallCount	No alarms
EC2	DescribeInstanceTypes	API	None	CallCount	No alarms
EC2	DescribeInstances	API	None	CallCount	No alarms
EC2	DescribeLaunchTemplateVersions	API	None	CallCount	No alarms
EC2	DescribeRegions	API	None	CallCount	No alarms
EC2 Auto Scaling	DescribeAutoScalingGroups	API	None	CallCount	No alarms
EC2 Auto Scaling	DescribePolicies	API	None	CallCount	No alarms
EC2 Auto Scaling	DescribeScalingPolicies	API	None	CallCount	No alarms
Logs	GetLogEvents	API	None	CallCount	No alarms
Logs	DescribeMetricFilters	API	None	CallCount	No alarms
Logs	DescribeLogStreams	API	None	CallCount	No alarms
Logs	DescribeLogGroups	API	None	CallCount	No alarms

Рисунок 3.4 – Вигляд груп метрик сервісу моніторингу

За результатами впровадження конвеєра постачання програмного забезпечення було розроблено набір скриптів автоматизації, які дають змогу виконувати поетапне розгортання програмного забезпечення спочатку в dev-, а потім у prod-середовищах. Процедури неперервної інтеграції можуть передбачати багато інших кроків, спрямованих, у першу чергу, на тестування

програмного забезпечення. Проте, як уже вказувалось, інструменти модульного, інтеграційного та приймального тестування переважно є суто індивідуальними для обраних технологій та мов програмування. Тому першочергово увага приділялась відтворенню базової узагальненої функціональності конвеєра постачання програмного забезпечення. Сформовані скрипти можна доповнювати та розширювати командами для специфічних програмних засобів автоматизації постачання.

ВИСНОВКИ

У результаті проведеного дослідження з аналізу технологій розробки та впровадження програмного забезпечення були виявлені кілька ключових аспектів, які необхідно врахувати для успішного створення та оптимізації конвеєра постачання.

Перш за все, виконаний огляд сучасних технологій дав змогу визначити найбільш ефективні та підходящі інструменти для автоматизації процесів конвеєра постачання. Враховуючи характеристики проекту та потреби команди, було здійснено відповідний вибір інструментів та їх налаштування.

Далі, була проведена розробка та імплементація конвеєра постачання з урахуванням конкретних вимог та характеристик проекту. Це включало в себе розробку не лише технічних аспектів, але і врахування потреб розробників

Після впровадження конвеєра було здійснено тестування та оптимізацію, спрямовану на забезпечення швидкості, надійності та якості випуску програмного забезпечення. Це дозволило виявити можливі проблеми та вдосконалити процеси для досягнення найкращих результатів.

Нарешті, було проведено оцінку ефективності впровадження конвеєра постачання на практиці та визначено його вплив на процес розробки та якість продукту. Це дозволило зробити висновки щодо успішності проекту та ідентифікувати можливі напрямки подальшого вдосконалення.

Основні напрямки для покращення конвеєра програмного забезпечення включають використання таких інструментів, як Terraform, для управління інфраструктурою. Також важливим аспектом є захист та безпека, зокрема шляхом інтеграції інструментів для аналізу безпеки, таких як Snyk або Dependabot, у конвеєр CI/CD.

Отже, впровадження конвеєра постачання є важливим кроком у процесі розробки програмного забезпечення, який дозволяє забезпечити ефективність та якість продукту на кожному етапі його життєвого циклу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. 8 surprising facts about real Docker adoption. Datadog. URL: https://www.datadoghq.com/docker-adoption/ (дата звернення: 10.06.2024).
2. AWS EC2 documentation. URL: https://docs.aws.amazon.com/ec2/?icmpid=docs_homepage_featuredsvcs (дата звернення: 10.06.2024).
3. AWS EC2. URL: https://docs.aws.amazon.com/ (дата звернення: 10.06.2024).
4. Cloud pyramid: IaaS, PaaS and SaaS. GigaCloud: Хмарні Технології та Хмарний Сервіс для Бізнесу. URL: https://gigacloud.ua/en/blog/navchannja/hmarna-piramida-iaas-paas-i-saas (дата звернення: 10.06.2024).
5. Cloud-services-market. URL: https://www.precedenceresearch.com/cloud-services-market (дата звернення: 07.06.2024).
6. Docs | Next.js. Next.js by Vercel - The React Framework. URL: https://nextjs.org/docs (дата звернення: 10.06.2024).
7. Express - Node.js web application framework. Express - Node.js web application framework. URL: https://expressjs.com/ (дата звернення: 10.06.2024).
8. flexera. URL: https://info.flexera.com/CM-REPORT-State-of-the-Cloud?_gl=1*1qqbgnz*_gcl_au*MTMwMDc0Mjk1My4xNzE3OTE0NTM5 (дата звернення: 04.06.2024).
9. Get started with Prisma | Prisma Documentation. Prisma | Simplify working and interacting with databases. URL: https://www.prisma.io/docs/getting-started (дата звернення: 10.06.2024).
10. GitHub Actions documentation - GitHub Docs. GitHub Docs. URL: https://docs.github.com/en/actions (дата звернення: 10.06.2024).
11. Home. Docker Documentation. URL: https://docs.docker.com/ (дата звернення: 10.06.2024).
12. Microservices | decoder. Thoughtworks. URL: https://www.thoughtworks.com/insights/decoder/m/microservices (дата звернення: 10.06.2024).

13. Node.js – Introduction to Node.js. Node.js – Run JavaScript Everywhere. URL: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs (дата звернення: 10.06.2024).
14. Quick start – react. React. URL: https://react.dev/learn (дата звернення: 10.06.2024).
15. Turborepo. Turbo. URL: https://turbo.build/repo (дата звернення: 10.06.2024).