Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Навчально-науковий центр заочної форми навчання
(повна назва)

Кафедра Інформаційних управляючих систем
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів розгортання оточення програмного забезпечення при виконанні ІТ-проекту
(тема)

Виконав:

студент 2 курсу, групи УПГІТзм-20-1

Макарова Ю. О.
(прізвище, ініціали)

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

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

Освітня програма Управління проектами в галузі інформаційних технологій
(повна назва освітньої програми)

Керівник проф. Чала О. В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Петров К.Е.
(підпис) (прізвище, ініціали)

2021 р.

Харківський національний університет радіоелектроніки

Навчально-науковий центр заочної форми навчання

Кафедра Інформаційних управляючих систем

Рівень вищої освіти другий (магістерський)

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

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

Освітня програма Управління проектами в галузі інформаційних технологій
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Макаровій Юліані Олегівні
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів розгортання оточення програмного забезпечення при виконанні ІТ-проекту

затверджена наказом університету від 23 жовтня 2021 р. № 162Стз

2. Термін подання студентом роботи до екзаменаційної комісії 14 грудня 2021 р.

3. Вихідні дані до роботи Науково-технічні публікації та інтернет джерела з тематики кваліфікаційної роботи

4. Перелік питань, що потрібно опрацювати в роботі вступ, аналіз безперервної інтеграції та безперервного постачання програмного забезпечення при виконанні ІТ-проєкту, дослідження технологій розробки ІТ-проєктів, аналіз технологій DevOps, аналіз технологій безперервного постачання, постановка задачі кваліфікаційної роботи, методи розгортання оточень програмного забезпечення, чотирьох-етапний метод розгортання оточення програмного забезпечення, удосконалений метод розгортання оточень програмного забезпечення, планування проєкту розробки удосконаленого методу розгортання оточення, опис додавання нового тестового оточення розгортання у проєкт, статут проєкту, планування проєкту, практичне використання отриманих результатів, розгортання тестового оточення, експериментальна перевірка розробки та тестування нового функціоналу після додавання нового тестового оточення, висновки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз методів та технологій безперервної інтеграції та безперервного постачання програмного забезпечення при виконанні ІТ-проєкту	23.10.2021-11.11.2021	
2	Аналіз технологій DevOps	12.11.2021-16.11.2021	
3	Постановка задачі	17.11.2021-18.11.2021	
4	Дослідження методів розгортання оточень програмного забезпечення	19.11.2021-20.11.2021	
5	Планування проєкту розробки удосконаленого методу розгортання оточення	21.11.2021-26.11.2021	
6	Розгортання тестового оточення	27.11.2021-30.11.2021	
7	Практичне використання удосконаленого методу розгортання оточень програмного забезпечення	01.12.2021-05.12.2021	
8	Оформлення пояснювальної записки	06.12.2021-10.12.2021	
9	Підготовка презентації	11.12.2021-12.12.2021	
10	Подання студентом роботи для перевірки на антиплагіат	14.12.2021	
11	Надання роботи на підпис науковому керівнику	14.12.2021	
12	Надання роботи на рецензію	14.12.2021	
13	Надання роботи на підпис завідувачу кафедри	14.12.2021	
14	Захист кваліфікаційної роботи	15.12.2021	

Дата видачі завдання _25_ _____жовтня____2021 р.

Студент _____
(підпис)

Керівник роботи _____ _____проф. Чала О. В._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи містить: 80 сторінок, 38 рисунків, 1 таблицю, 1 додаток, 25 джерел.

БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ, БЕЗПЕРЕВНЕ ПОСТАЧАННЯ, ОТОЧЕННЯ РОЗГОРТАННЯ, РОЗГОРТАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, DEVOPS

Об’єкт дослідження кваліфікаційної роботи – оточення розгортання при виконанні ІТ-проєктів.

Мета дослідження – ознайомитися с існуючими методами розгортання оточень при виконанні ІТ-проєктів.

Проведено аналіз методів розгортання оточень при виконанні ІТ-проєктів. На підставі проведеного аналізу запропоновано покращений метод оточень розгортання за допомогою додаткового тестового оточення. Розроблена постановка задачі дослідження.

Проведено і виконано опис додавання нового тестового оточення. Виконано експериментальну перевірку удосконаленого методу.

ABSTRACT

The explanatory note to the qualification work: 80 pages, 38 pictures, 1 tables, 1 attachment, 25 sources.

CONTINUOS DELIVERY, CONTINUOS INTEGRATION, DEVELOPMENT OF THE SOFTWARE, DEVELOPMENT ENVIRONMENT, DEVOPS.

The object of research of the qualification work is the deployment environment during the implementation of IT projects.

The purpose of the study is to get acquainted with the existing methods of deploying environments in the implementation of IT projects.

The analysis of methods of deployment of environments at performance of IT projects is carried out. Based on the analysis, an improved method of deployment environments with the help of an additional test environment is proposed. The formulation of the research problem is developed.

A description of adding a new test environment is performed and performed. An experimental test of the improved method was performed.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОКАЗНИКИ

ПЗ – програмне забезпечення

CI – Continues Integration

CD – Continues Delivery

TDD – Test Driven Development

DevOps – Development and Operations

WBS – Work Breakdown Structure

OBS – Organizational Breakdown Structure

DSR – Design Scientific Research

ВСТУП

Сучасний підхід до організації управління складним комплексом робіт, які повинні бути виконані точно в заданий час та в межах визначеної кількості ресурсів базується на науково визначених методологіях і методах управління ІТ-проєктами. Ефективність виконання того чи іншого проєкту в значній мірі залежить від організації планування та управління проєктом в цілому, його ресурсами, сукупностями робіт та окремими роботами.

Одним із головних ресурсів при розробці ІТ-проєктів є наявність декількох оточень розгортання програмного забезпечення. Вони потрібні щоб якість розробки, інтеграції та тестування підтримувалась на високому рівні, а це означає, що і сам проєкт буде якісним та буде задовольняти потреби замовника.

Розгортання програмного забезпечення – це набір операцій, які роблять програмну систему придатною для використання. Цей процес є частиною життєвого циклу програмного забезпечення.

Взагалі кажучи, процес розгортання складається з кількох взаємопов’язаних дій, і між ними можуть бути переходи. Така діяльність може відбуватися з боку виробників і споживачів. Оскільки кожна програмна система унікальна, важко передбачити всі процеси та процедури в процесі розгортання.

Кваліфікаційна робота здійснюється відповідно до вимог з організації та інструкцій щодо їх виконання [1] та національного стандарту [2].

Оформлення переліку джерел посилання відбулося відповідно до національного стандарту [3].

1 АНАЛІЗ МЕТОДІВ ТА ТЕХНОЛОГІЙ БЕЗПЕРЕВНОЇ ІНТЕГРАЦІЇ ТА БЕЗПЕРЕВНОГО ПОСТАЧАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ПРИ ВИКОНАННІ ІТ-ПРОЄКТУ

1.1 Дослідження технологій розробки ІТ-проєктів

На сьогоднішній день розробка програмного забезпечення відбувається у все більш нестабільному бізнес-середовищі. Як правило, це – непередбачувані ринки, складні та мінливі вимоги клієнтів, зміна термінів випуску продукту на ринок для користування потенційним клієнтам і інформаційні технології, що швидко розвиваються.

Для вирішення цієї ситуації гнучкі практики, які підтримують гнучкість, ефективність і швидкість, стають дедалі більше привабливими для підприємств, що спеціалізуються у області розробки програмного забезпечення. Підкреслюючи використання ітерацій та розробку невеликих функцій, agile-практики підвищили здатність компанії, котрі розробляють програмне забезпечення та швидко пристосовуються до вимог клієнтів та мінливим ринкам.

Agile – це гнучкий методологія розробки програмного забезпечення, яка часто застосовується у невеликих командах та у великих організаціях.

Існує багато компаній, основним завданням яких є розробка програмного забезпечення і вони дійсно досягли успіху в застосуванні методів Agile у деяких частинах їх організації. Також є компанії, які досягли успіху у впровадженні гнучкої практики для такого ступеня, що функціональність програмного забезпечення може бути безперервно розгорнута на сайтах клієнтів, для того щоб відгуки клієнтів і дані про використання клієнтів можна ефективно застосовувати в усьому процесі розробки, доставки та розгортання програмного забезпечення [4]. Процес безперервного розгортання програмного забезпечення містить етапи зображені на рисунку 1.1.

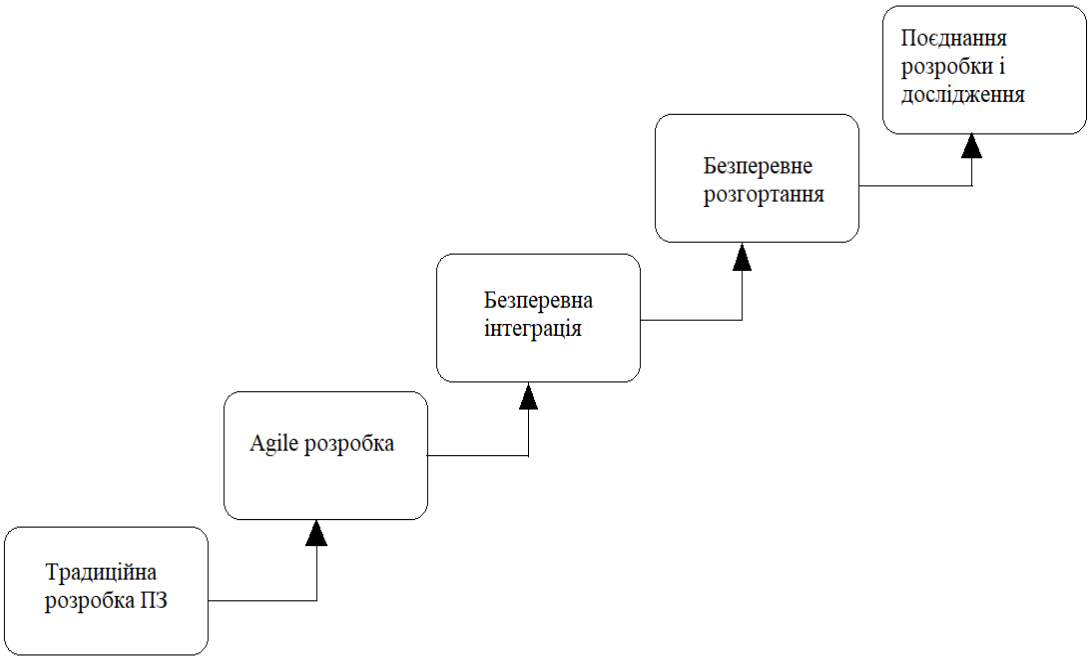

Рисунок 1.1 – Етапи впровадження безперервної доставки

Традиційний розвиток компаній в напрямку безперервного постачання розглядається, як процес розробки програмного забезпечення, що характеризується повільними циклами розвитку, взаємодією у стилі водоспаду між управлінням продукту, його розробкою, системою тестування і відгуків клієнтів[5]. Як правило, це великі проєктні групи, які поділяються на підрозділи, такі як архітектура системи, проектування та тестування.

Розвиток є послідовним із ретельним етапом планування в самому початку кожного проєкту. Як правило, доставка до замовника відбувається в самому кінці проєкту, і після цього, клієнти зможуть надати відгук про роботу нового функціоналу продукту, який вони отримали.

Наступним кроком є Agile розробка – це коли продукт розробляється, тобто компанія, прийняла agile практики, але управління продуктом і система перевірки все ще працюють відповідно до традиційної моделі розробки. Після впровадження agile-практик не завжди існує короткий зворотній зв’язок із користувачами. Команда не повинна вважатися розвинутою, якщо процес

отримання зворотного зв'язку з клієнтами і керівництвом становить шість місяців і більше [6].

Компаніям, які використовують безперервну інтеграцію, вдалося встановити практики, які дозволяють часто робити інтеграція, щоденні збірки та швидке внесення змін, наприклад, автоматизовані збірки та автоматизоване тестування.

Безперервна інтеграція – це розробка програмного забезпечення при якій, члени команди інтегрують свою роботу часто, що призводить до кількох інтеграцій на день. Ідея полягає у автоматизації тестових випадків, збірок, компіляції, покриття коду тощо дозволяє командам щодня тестувати та інтегрувати свій код з основним кодом, це мінімізує час, який є необхідним для створення ідеї щоб втілити її в програмному забезпеченні. На даний момент і розробка продукту, і перевірка системи працюють відповідно до практик agile.

Наступним кроком є безперервне розгортання. На цьому етапі функціональність програмного забезпечення розгортається постійно або, принаймні, частіше на оточенні клієнта. Це дає можливість постійно отримувати зворотній зв'язок від клієнтів, можливість отримувати дані про використання продукту клієнтами, і виключити будь-яку роботу, яка не приносить цінності замовнику. На цьому етапі також проводиться науково-дослідницька робота з управління продуктом оскільки всі клієнти залучені до швидкого, динамічного циклу його розробки.

Останнім кроком є поєднання розробки і досліджень, де вся система реагує і діє на основі відгуків клієнтів і де фактичне розгортання програмного забезпечення розглядається як спосіб експериментування та тестування того, що потрібно клієнту. На цьому етапі розгортання програмного забезпечення розглядається більше як відправна точка для подальшого «налаштування» функціональності, а не доставки кінцевого продукту.

1.2 Аналіз технологій DevOps

DevOps – це набір принципів та практик для покращення співробітництва між розробкою та IT-операціями. Концепція DevOps [4] виникла для того, щоб усунути розрив між розробкою програмного забезпечення та його впровадженням у виробництво у великих підприємств [5]. Основною метою DevOps є використання безперервного процесу розробки програмного забезпечення, такі як безперервна доставка, безперервне розгортання та мікросервіси для підтримки гнучкої розробки програмного забезпечення життєвого циклу.

Інші тенденції в цьому контексті полягають у тому, що програмне забезпечення все частіше постачається через інтернет, або на стороні сервера або як канал для доставки безпосередньо клієнту, і все більшого поширення набувають мобільні платформи та технології, на яких працює це програмне забезпечення [6]. Ці тенденції, що розвиваються, підтримують швидкі і короткі цикли постачання програмного забезпечення в динамічному світі. У зв'язку з цим DevOps був добре прийнятий у спільноті розробників програмного забезпечення

Комунікація та зв'язок між Dev та Ops, дозволяє краще відстежувати наскрізне розгортання виробництва та частіше розгортання кращої якості, що економить гроші компанії. Щоб полегшити цю співпрацю та покращити зв'язок між Dev та Ops, необхідно впровадити кілька ключових елементів у процеси рисунок 1.2.

Рух DevOps базується на трьох осях:

– культура співробітництва: Це сама суть DevOps – той факт, що команди більше не розділені за спеціалізацією (одна команда розробників,одна команда Ops, одна команда тестувальників тощо), а, навпаки, ці люди об'єднуються разом, створюючи багатопрофільні команди, які мають одну й ту саму мету: якнайшвидше створити додану вартість продукту;

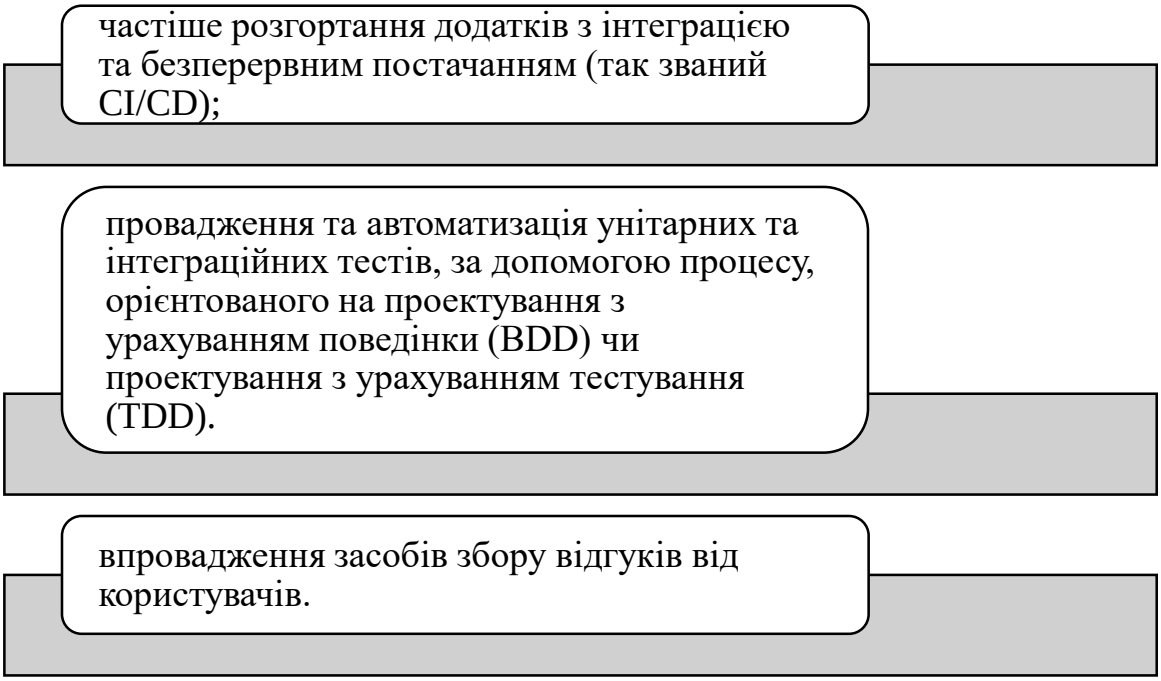

Рисунок 1.2 – Елементи для покращення зв’язку між Dev та Ops

– процеси: щоб очікувати швидкого розгортання, ці команди повинні слідувати процесам розробки ПЗ, заснованих на методології agile, з ітеративними фазами, які дозволяють покращити якість функціональності та швидкий зворотний зв'язок. Ці процеси мають бути не тільки інтегровані в робочий процес розробки з безперервною інтеграцією, а також в робочий процес розгортання за допомогою безперервного постачання та розгортання;

– інструменти: вибір інструментів та продуктів, що використовуються командами, дуже важливий у DevOps. Справді, коли команди були поділені на Dev та Ops, кожна команда використовувала свої специфічні інструменти розгортання для розробників та інструменти інфраструктури Ops – що ще більше збільшувало розрив у комунікаціях.

З командами, що об'єднують розробку та експлуатацію, і з цією культурою єдності, використовувані інструменти повинні бути придатними для використання та експлуатації всіма членами команди.

Розробникам необхідно інтегруватися з інструментами моніторингу, що використовуються операційними командами, щоб виявити проблеми продуктивності якомога раніше і з інструментами безпеки, що надаються операційними службами захисту доступу до різноманітних ресурсів.

Ops, з іншого боку, повинні автоматизувати створення та оновлення інфраструктури та інтегрувати код у менеджер коду; це називається "інфраструктура як код", але це може бути зроблено лише у співпраці з розробниками, які знають інфраструктуру, необхідну для програми. Ops також має бути інтегрований у процеси та інструменти випуску додатків.

На рисунку 1.3 продемонстровано три осі культури DevOps

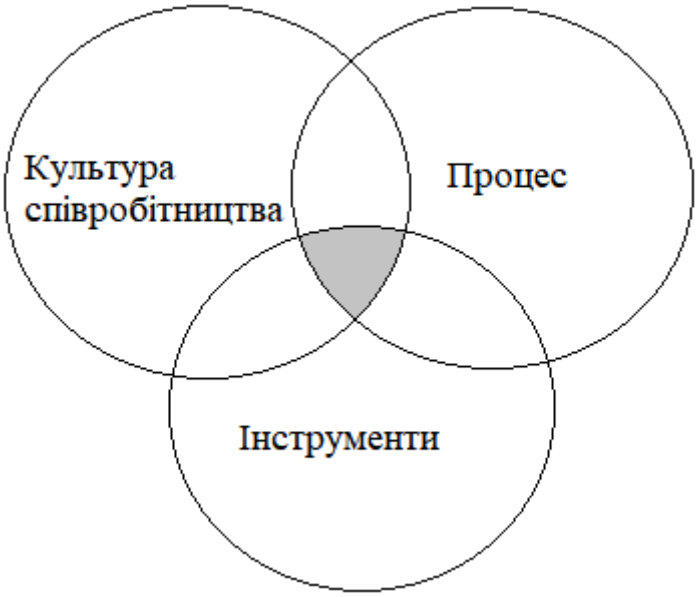

Рисунок 1.3 – Співпраця між Dev та Ops, процеси та використання інструментів.

Безперервна інтеграція – це одна із практик розробки ПЗ. Однією з переваг CI є швидкий зворотний зв'язок, який вона забезпечує. Зворотний зв'язок надає інформацію про стан проєкту кілька разів на день, коли інтеграція виконується безперервно. Швидкий зворотний зв'язок дозволяє розробникам швидко виявити дефекти після їх появи. Таким чином, це

дозволяє скоротити час між виявленням дефекту та його усуненням. Іншими словами, CI може покращити загальну якість програмного забезпечення.

Постійний зворотний зв'язок також дозволяє використовувати такі методи розробки програмного забезпечення, як редагування та розробка на основі тестів, оскільки невеликі зміни можуть бути швидко інтегровані в цілу систему, і розробник може отримати зворотний зв'язок незабаром після кожної фіксації. Крім того, в тій же мірі, якою CI дозволяє TDD – Test Driven Development, можна також помітити, що це працює і в зворотний бік. Розробка через тестування TDD – це технологія розробки ПЗ, яка використовує короткі ітерації розробки, що починаються з попереднього написання тестів, які визначають необхідні покращення або нові функції. TDD допомагає розділити великі зміни на більш дрібні, що дозволяє легко інтегрувати зміни по невеликими частинами.

Якщо TDD практикується, то, можливо, зміни можуть бути інтегровані навіть так часто, як після кожного циклу TDD. Однак, оскільки це є предметом даного дослідження, час зворотного зв'язку може збільшуватися зі збільшенням часу збірки. Швидкий зворотний зв'язок можна розглядати як перевагу CI, але також можна втратити цю перевагу, коли збірка займає кілька годин.

Однією з основних переваг безперервної інтеграції є перенесення частини спілкування між розробниками щодо рівня коду. Якщо інтеграція виконується часто, розробники можуть повідомляти один одному, до якої частини системи вони вносили зміни і таким чином дізнаватися про конфлікти у своєму коді раніше. Конфлікт виникає коли два або більше розробників змінили одні й ті самі кілька рядків коду файлу. Ключ до швидкого усунення проблем - це їхнє швидке виявлення.

Чим раніше розробники дізнаються про конфлікт, тим легше його усунути. З іншого боку, якщо інтеграція здійснюється, наприклад, кожні два тижні, то можливо, що існують невиявлені конфлікти, що ховаються в коді без відома розробників. Ті типи конфліктів, які залишаються невиявленими

протягом декількох тижнів або навіть довше, можуть бути дуже складними для вирішення.

Швидке виявлення конфліктів – це не єдина перевага CI, пов'язана з комунікацією. Отримання інформації про те, що хтось інший зачіпає ту саму ділянку коду, що й ви, може відкрити комунікацію між двома розробниками. У цьому сенсі CI працює як привід для комунікації між людьми та дозволяє людям взаємодіяти та працювати разом.

Безперервна інтеграція складається з 7 етапів, які схематично зображені на рисунку 1.4.

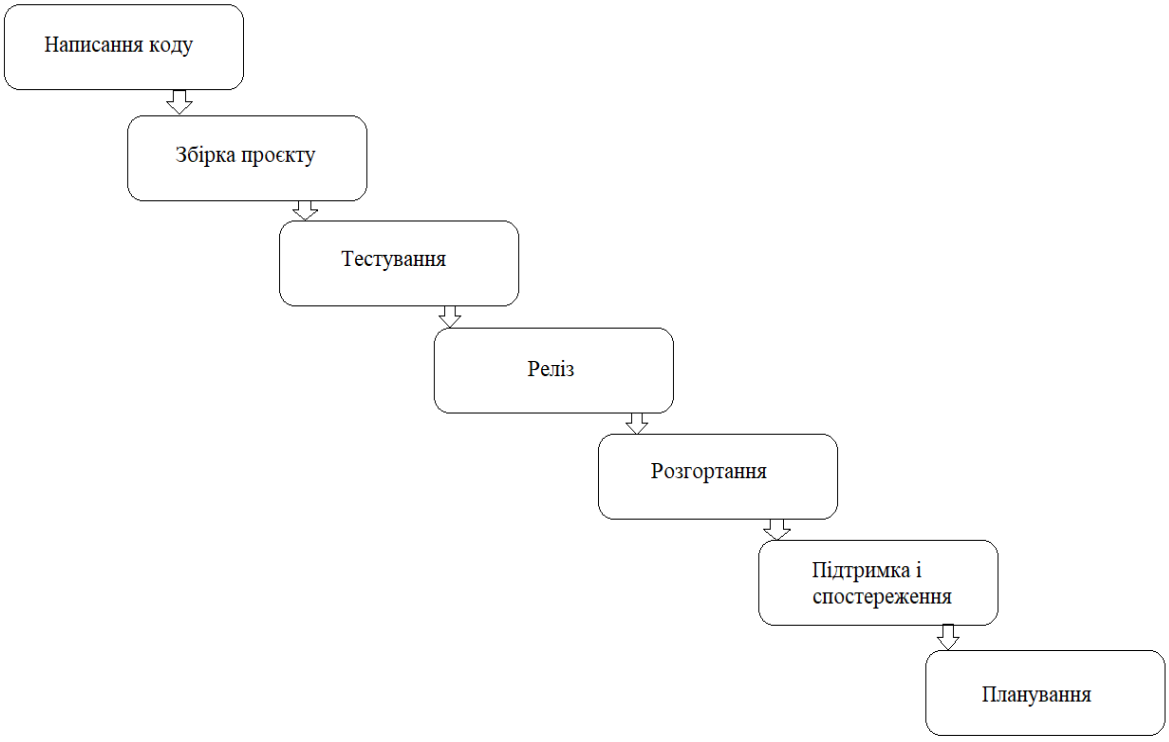

Рисунок 1.4 – Етапи безперервної інтеграції

На етапі написання коду кожен із розробників реалізує свій модуль, тестує його і поєднує результат своєї роботи із основною гілкою. Потім починається наступний етап, тобто відбувається збірка всього проєкту. Коли збірка пройшла успішно починається етап тестування, на якому відбувається

перевірка всього проєкту командою тестувальників. Після успішного завершення тестування настає етап релізу, де збірка отримує кінцеву версію продукту. Потім починається етап розгортання робочої версії проєкту на оточенні кінцевих користувачів. На етапі підтримки і спостереження кінцеві користувачі починають працювати з продуктом, а команда розробників підтримує його та аналізує досвід користувачів. Останнім етапом є планування, де на основі досвіду користувача формуються запити на новий функціонал для продукту. Після чого цикл замикається і переходить у початкову стадію – написання коду.

Середовище в рамках Design Scientific Research формується із трьох суб'єктів: людей, організації та технології. Люди можуть мати різні ролі, можливості та характеристики, які можуть впливати на навколишнє середовище. Стратегії, структура, культура та різні процеси організацій також є тим, що формує середовище. Технічне середовище визначається шляхом опису різних технологій, що використовуються в середовищі, такі як інфраструктура, додатки, комунікаційна архітектура та можливості розробки.

Люди – користувачі створеного додатку - це в основному розробники програмного забезпечення, оскільки саме вони ті, хто перевіряє зміни в коді і в даний час страждає від повільного зворотного зв'язку системи CI. Цей винахід зрештою призведе до менш клопіткої інтеграції програмного забезпечення, оскільки більш швидкий зворотний зв'язок потенційно може дозволити розробникам програмного забезпечення практикувати безперервну інтеграцію.

Розробники, що працюють у цільовій організації, мають досить високу думку про поточну ситуацію в середовищі CI. Крім того, керівництво постійно стурбоване повільним часом зворотного зв'язку та ресурсами апаратного забезпечення для тестування.

Наприклад: продукт, розроблений в одній із компаній, є телекомунікаційним продуктом, і його розробка була розподілена між кількома сайтами, розташованими у кількох країнах. Продукт не є новим і

існує кілька років. Він складається з різних компонентів системного рівня, розробка яких також розподілена на кількох об'єктах. Це дослідження проводиться в лінійній організації всередині компанії-виробника, і вона розробляє один із компонентів системи. Над компонентом системи працюють 8-10 команд. У кожній команді працює близько восьми чоловік, і команди побудовані за крос-функціональним принципом, оскільки в кожній команді є інженери з тестування, інженери з специфікацій та інженери з програмного забезпечення. Команди працюють в основному над обслуговуванням, оптимізацією та новими функціями.

Нинішні команди називаються командами Scrum або командами функцій, і деякі з практики Scrum були прийняті. Деякі з agile-практик програмної інженерії також були впроваджені. Парне програмування та розробка на основі тестування не є нормою, але інженерів заохочують практикувати їх.

Організація також рухається у напрямку створення справжніх команд розробників та колективного володіння кодом, що означає, що буде більше сайтів і більше команд, залучених у розробку компонентів системи, що саме по собі вимагає наявності функціонуючої системи CI [7].

Ще одна організаційна мотивація полягає в тому, що система безперервної інтеграції значною мірою покладається на середовища тестування, і через це потрібна велика кількість апаратних ресурсів для підтримки системи CI. Проблема полягає в тому, що тестові машини коштують дорого та їх кількість обмежена.

Заміна поточної політики повторного тестування всіх продуктів, яка вимагає величезної кількості обладнання для запуску кожного тесту на кожному фіксуванні змін, тести не будуть виконуватися, якщо вони не належать до зафіксованого коду. Це може вплинути на поточну ситуацію, оскільки їм не доведеться виконувати непотрібні тести.

Технологія – це сума будь-яких прийомів, навичок, методів і процесів, що використовуються у виробництві товарів або послуг або для досягнення

цілей, таких як наукове дослідження. Програмний продукт, який виробляє компанія, є великомасштабною системою, що вбудовується, написаною на C++. Інструментом для безперервної інтеграції є Jenkins, який обробляє виконання різних збірок, тестів та статичного аналізу. Зворотній зв'язок від Jenkins можна побачити з його графічного веб-інтерфейсу. Jenkins – це кросплатформна програма для безперервної інтеграції та безперервної доставки, яку можна використовувати для збірки та тестування програмних проєктів. На рисунку 1.5 зображено як середовище CI.

Ідея CI полягає в тому, щоб усі фіксували зміни в одну й ту саму головну гілку. Однак деякі проєкти розподіляють свою розробку за однією або декількома гілками.

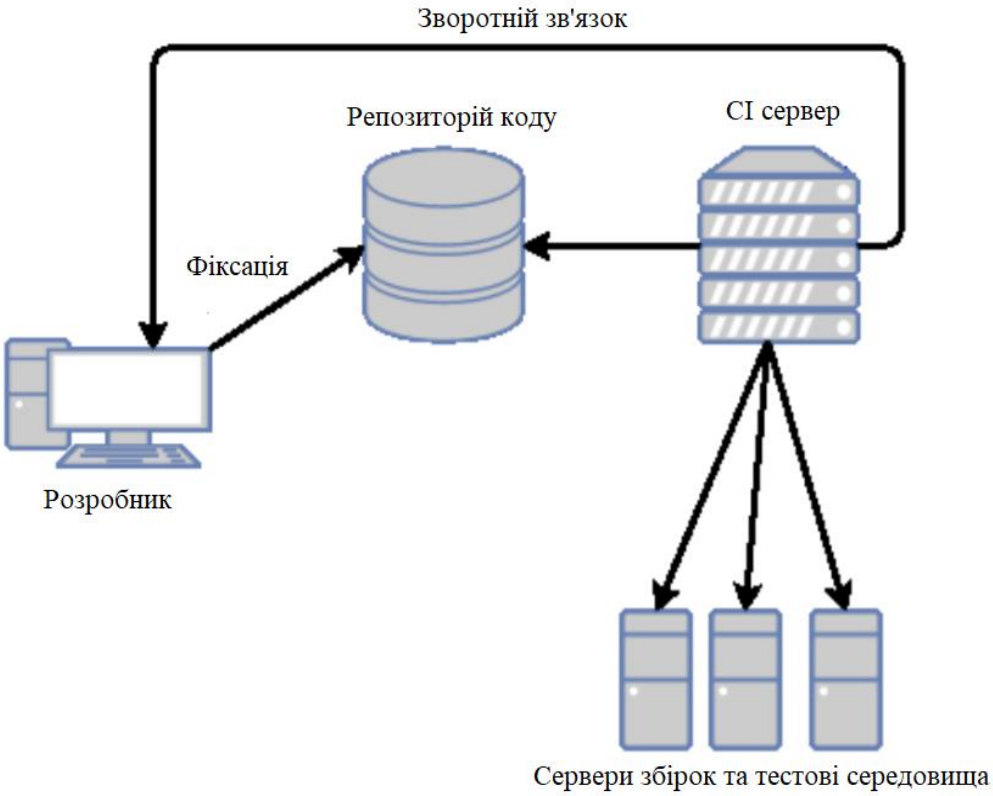

Рисунок 1.5 – Середовище CI

Є кілька причин, через розробники вимушені так виконувати. Розгалуження може бути зроблено за функціями, релізом або командою. Причина також може бути фізичною, оскільки розгалужуються різні файли, компоненти або підсистеми. Розгалуження також може бути зроблено з функціональної конфігурації системи або може бути засноване на збиранні та виконанні. Також можуть бути організаційні мотиви для розгалуження, коли гілки створюються для завдань, підпроєктів, ролей та груп. Причина може бути процедурною, коли гілки створюються для підтримки різних процесів і станів. Найбільші організації можуть мати гілки для різних компонентів та продуктів [8].

Розгалуження в CI пов'язано з тим, що згідно з визначенням, робота над відгалуженням не є безперервною інтеграцією, тому що розробники не інтегруються з основною гілкою. Зміни у сторонньому відгалуженні переносяться в основну гілку шляхом злиття. Злиття означає взяття змін із гілки, над якою ведеться робота та використати їх до основної гілки. Якщо відгалуження не зливається з основною лінією хоча б щодня, воно не може називатися CI. Крім того, існує теорія, що подання змін в основну гілку - це єдиний спосіб виконання CI, який може інтерпретувати розробку в довгоживучих гілках як не зовсім CI [9].

Вочевидь, існують різні погляди те що, що кваліфікується як CI. Рекомендується використовувати гілки якомога рідше. Хоча бувають деякі випадки використання відгалужень, наприклад, відгалуження функцій корисні, коли розробник (або команда) може працювати над функцією ізоляції, при цьому отримуючи поточну версію з основної. Використовуючи функціональні або командні гілки, розробники можуть вирішувати, які функції будуть випущені і злиті з основною версією.

Вони можуть бути використані для складного редагування, коли змінюються деякі основні елементи системи. Це може бути, наприклад, покращення можливостей або перепроектування шарів системи. Незважаючи на те, що розгалуження можуть бути корисними у таких ситуаціях, інший

підхід полягає в тому, щоб вносити зміни невеликими поступовими кроками, зберігаючи при цьому проходження всіх тестів [10].

Проблема з розгалуженням у тому, що ймовірність виникнення конфліктів під час роботи у різних гілках зростає. Якщо конфлікти виникають під час злиття, процес проходить швидко і легко. Однак проблема з розгалуженнями та злиттями виникає, коли зміни, зроблені в бічній гілки, вступають у конфлікт із основною гілкою або навпаки. Якщо злиття відбуваються не часто конфлікт між основною гілкою та відгалуженням може існувати довгий час, і ніхто про це не дізнається. Робота над відгалуженнями потенційно може призвести до дуже складного процесу злиття та інтеграції наприкінці проєкту.

Ще одна проблема з довгоживучими гілками полягає в тому, що страх перед конфліктами при злитті може утримувати розробників від редагування коду. Вони можуть боятися переміщати код між класами або перейменовувати методи, тому що вони не хочуть мати справу з конфліктами злиття, коли гілка буде злита з основною. Можуть бути такі ситуації, коли розробник перейменовував одну змінну, а потім проводив решту дня, виправляючи непрацюючі збірки та вирішуючи конфлікти злиття. Відсутність редагування може призвести до збільшення технічного обов'язку [11].

Небажання займатися редагуванням означає, що розробники в цій організації не можуть дотримуватися "правила бойскауту" [12], згідно з яким ви залишаєте кодову базу в кращому стані, ніж ви її знайшли. Проведення невеликого чищення для кожної перевірки коду та підтримання чистоти кодової бази може стати неможливим через використання відгалужень через страх перед конфліктами злиття, що вони викликають.

Використання гілок прибирає аспект людського спілкування з CI, що, є поганим, оскільки спілкування може розглядатися як ключовий фактор у розробці програмного забезпечення. Якщо воно на основній гілці, розробники можуть підхоплювати зміни один одного будь-коли. Якщо ні, інші розробники

не зможуть підхопити зміни досить швидко. Це може призвести до повторного використання коду та дублювання зусиль.

Великі проєкти можуть мати труднощі з CI. Коли кодова база складається з мільйона рядків коду – це те, що час збирання буде довгим. Кількість часу, на який розробнику доводиться чекати до закінчення складання, може бути критичним для робочого процесу розробника. Десять хвилин – це максимальний час для завершення збирання [13].

Однак при великій кодовій базі необхідний хороший набір тестів для перевірки правильності програмного забезпечення. Це проблема в тому сенсі, що необхідно виконати велику кількість тестів, але в той же час потрібен короткий час складання. Тим не менш, зупинка розробки лише для того, щоб дочекатися зворотного зв'язку, може сповільнити ритм розробки для всіх, хто працює над цим проєктом. Повільні складання можуть навіть вплинути на те, що люди сприймають CI як практику, вони можуть спричинити рідкісну інтеграцію розробника. Таким чином, отримання швидкого відгуку від клієнтів дуже важливе.

Тривалий час збірки також впливає на готовність розробників до редагування. Це може викликати проблеми у команді, його оскільки відсутність призвела до великої кількості технічного обов'язку. Ідеї поліпшення були внесені в задачу, і вони мали бути виконані, коли був час і пріоритет. Коли збірка триває недовго то в таких командах розробники більш готові до серйозного редагування, тому вони не перевантажені технічним боргом і не страждають від вищезгаданих проблем.

Довгі складання призводять до нечастої інтеграції, тоді як короткі складання призводять до дуже частої інтеграції [13]. Середні збірки змушують розробників інтегруватися кілька разів на день. Довгими збірками вважаються ті, що займають більше 30 хвилин, середні - 5-10 хвилин і короткі - менше двох хвилин. Якщо збірки займають більше 30 хвилин, то інтеграція, швидше за все, відбуватиметься раз на тиждень і на процес інтеграції буде виділено цілий день. Довгі збірки роблять зусилля з інтеграції більшими, і коли це

відбувається, весь процес стає перервою, що змушує розробників планувати інтеграцію.

Короткі збірки забезпечують розробнику швидкий відгук від клієнтів, у той час як довгі збірки викликають повільний зворотній зв'язок [14]. Очікування завершення збірки також може відволікати, і якщо час очікування тривалий, то розробник швидше за все займатиметься іншими справами під час очікування. Можна стверджує, що довгі збірки знижують моральний дух команди.

Довгі збірки можуть мати ефекти, які є специфічними для CI, але вони також можуть мати когнітивні та емоційні ефекти. Час тривалості збірки впливає на розмір і частоту фіксування змін, час простою та зусилля з інтеграції. Вони впливають на потік розробки та час зворотнього зв'язку, а також можуть впливати на задоволеність розробників та моральний дух команди. CI дозволяє використовувати такі практики гнучкої розробки програмного забезпечення, як редагування та розробка на основі тестів. Це може бути не так, якщо час збірки занадто тривалий. Можна стверджувати, що тривалі складання можуть вплинути на те, як розробник пише код, і позбавити його деяких переваг, які в ідеалі мають забезпечувати CI.

Коли безперервна інтеграція впроваджується у великі проєкти, можуть виникнути труднощі у них. Якщо розробник отримує електронною поштою повідомлення про непрацюючу збірку кілька разів на день через те, що хтось інший пошкодив збірку, це може стати просто постійним шумом. Нестабільна головна гілка – це важке середовище для прогресу. Слід зазначити, що CI не викликає нестабільності, але вона її оголює [15].

Зростаюча база коду сама по собі збільшує час компіляції, але сам по собі час компіляції не є такою великою проблемою, як кількість тестів, що зростає. Час складання потенційно може збільшуватися експоненціально зі зростанням кількості тестів. Проблеми, пов'язані зі зростанням кодової бази, виявляються, коли код додається до проєкту. Перша проблема полягає в тому, що обсяг коду, який виробляється, збільшується, що призводить до ще більш

серйозних проблем, пов'язаних зі зростанням кодової бази. Інша проблема це зростання кількості людей, що працюють над однією і тією ж кодовою базою, полягає в тому, що з'являється все більше і більше людей, які залежать від працюючої збірки, тому поломка торкнеться великою кількості людей. У CI фіксування поверх зламаної збірки заборонено. Важливо постійно підтримувати тести CI, і якщо хтось ламає збірку, вся команда повинна зупинитися і зосередитися на її виправленні. Насправді хоча в деяких випадках люди можуть ігнорувати зламану збірку і фіксувати свої зміни, незважаючи на те, що це заборонено. Це може викликати ряд проблем, оскільки неможливо перевірити правильність програмного забезпечення. Якщо така поведінка стане звичайною, то розробники можуть розглядати непрацюючу збірку як ситуацію "вільного вибору", коли всі просто скидають свої зміни поверх непрацюючої збірки.

Тим не менш, зламана збірка – це серйозна проблема в процесі розробки. Швидше за все, вона підриває продуктивність розробників. Поширеним підходом є примусове використання суворої процедури попередньої комісії, яка максимально ускладнює порушення складання. Така процедура попередньої фіксації може складатися із запуску всієї інтеграційної збірки на локальній машині, і тільки якщо вона проходить, дозволяється фіксація зміни. Це ефективний спосіб зберегти збірку у робочому стані, але він також збільшує час інтеграції.

Для певної перспективи такий спосіб мислення можна порівняти з ідеєю [10] , що інтеграція може виконуватися так само часто, як кожен цикл TDD. Очевидно, що кожен цикл TDD не може складатися із запуску всієї інтеграційної збірки, тому такі політики попереднього фіксування можуть зменшити частоту інтеграції або унеможливити часту інтеграцію. Порушення складання має бути прийнятним, тому що сервер інтеграції забезпечує зворотний зв'язок, якщо щось йде не так зі зміною коду. Перетворення локальної машини кожного розробника на інтеграційну машину - неефективний спосіб інтеграції, оскільки це може зашкодити продуктивності

та морального стану розробників. Тестування надто великої кількості речей на локальній машині перед фіксацією може стати вузьким місцем для всього процесу розробки та CI [16].

1.3 Аналіз технологій безперервного постачання

Безперервне постачання – це набір практик і принципів, що дозволяють випускати програмне забезпечення швидше та з більш високою частотою [17]. Це дозволяє командам виробляти програмне забезпечення протягом коротких циклів, а програмне забезпечення перебуває у стані, придатному для випуску будь-коли. Цього можна досягти за рахунок дисципліни та автоматизації самої поставки, включаючи створення, тестування та розгортання програмного забезпечення.

Безперервне постачання не слід плутати з безперервним розгортанням. Як було сказано вище, у CD акцент робиться на тому, що програмне забезпечення перебуває в стані, придатному для випуску, у той час як при безперервному розгортанні програмне забезпечення постійно випускається. Відмінність між цими двома методами полягає в тому, що при безперервному постачанні компанія може вирішувати, чи випускати програмне забезпечення, коли воно перебуває в стані, придатному для випуску. Безперервне постачання складається декількох етапів, які зображені на рисунку 1.6.

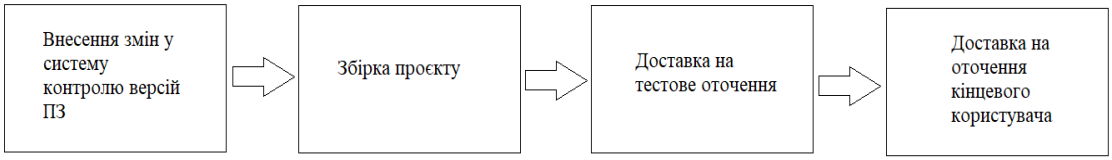

Рисунок 1.6 – Етапи безперервного постачання

Розробник зливає свою гілку з основною, потім відбувається збірка проєкту, якщо вона пройшла успішно, то відбувається розгортання на тестовому оточенні, де відбувається тестування всього проєкту. Коли тестування пройшло успішно, продукт доставляється на оточення кінцевого користувача.

Перспективи CD полягають у тому, що він дозволяє організаціям виводити на ринок удосконалені послуги та випереджати конкурентів у швидкій, ефективній та надійній формі. Хоча те, чого бажане кожної організації, використання CD може бути складним завданням. Особливо для великих підприємств, які вже мають середовище для випуску, це складно запровадити цю практику [18] . Крім того, можливість випускати продукцію безперервно часто блокується різними вузькими місцями в процесі доставки.

Незважаючи на те, що використання CD може бути складним, воно було успішно впроваджено в деяких організаціях. Були й такі компанії, які зіштовхнулися із труднощами при впровадженні CD. Прийняття CD складним завданням. Незважаючи на те, що були повідомлення про проблеми в адаптації CD, не було проведено досліджень, які б сфокусувалися на причинних механізмах цих проблем [19]. Вони вивчили адаптацію CD за принципом "знизу нагору" і повідомили про проблеми, з якими зіткнулися залучені в цей процес люди.

Як показано на рисунку 1.7, повільний зворотний є прямою ознакою дисфункціональної практики CD.

Існує три основні причини, які призводять до кількох механізмів процесу, що викликають повільний зворотній зв'язок. Корінні причини повільного зворотнього зв'язку можна відстежити до стадії-виходу процесу розробки та відсутність стратегії тестування. В даному випадку це призвело до затримки інтеграції, повільного складання та обмеженості апаратних ресурсів. Обмежені апаратні ресурси та затримка інтеграції можуть бути пов'язані з використанням кількох гілок. Повільні складання викликані

дублюючим тестуванням, яке є наслідком відсутності стратегії тестування [19].

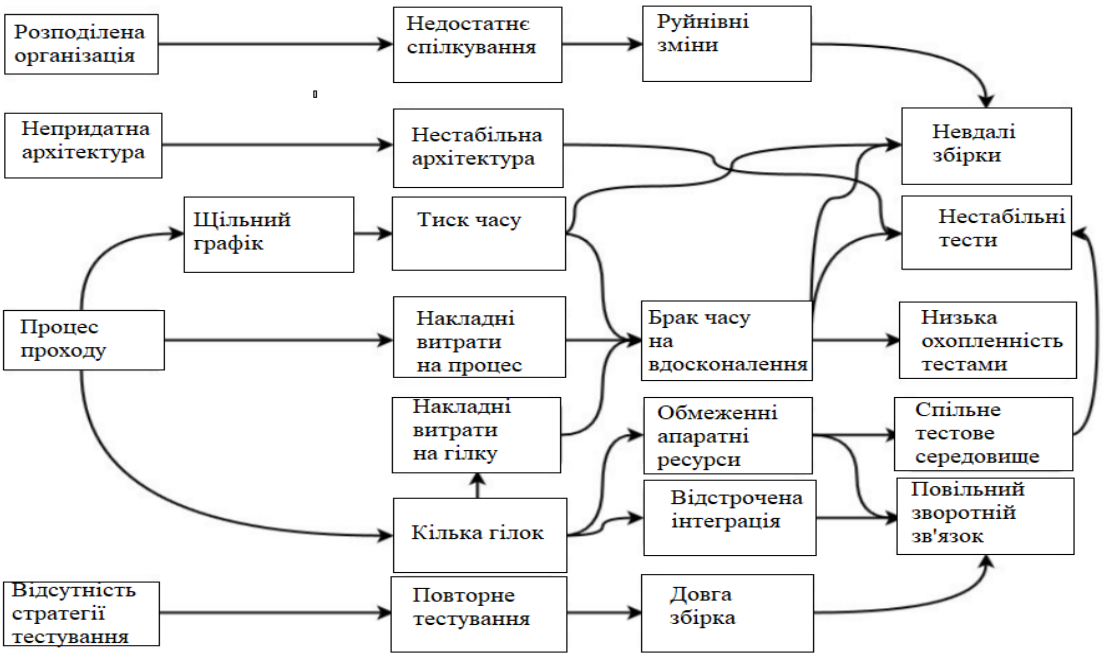

Рисунок 1.7 – Огляд прямих ознак, основних причин та механізму між ними

1.4 Постановка задачі дослідження

Об’єктом дослідження в рамках магістерської атестаційної роботи є процес безперервного постачання програмного забезпечення при виконанні IT-проєкту

Предметом дослідження є методи розгортання оточень програмного забезпечення при виконанні IT-проєкту.

Метою даної роботи є дослідження методів розгортання оточень програмного забезпечення для підвищення ефективності його безперервного постачання програмного при виконанні IT-проєкту.

Для досягнення мети, необхідно досліджувати наступні питання:

– методи розгортання оточень програмного забезпечення при виконанні ІТ-проєту;

– проблематика;

– опис додавання тестового оточення розгортання у проєкт;

– статут проєкту;

– планування проєкту;

– розгортання тестового оточення;

– експериментальна перевірка розробки та тестування нового функціоналу після додавання нового тестового оточення.

2 МЕТОДИ РОЗГОРТАННЯ ОТОЧЕНЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Чотирьох-етапний метод розгортання оточення програмного забезпечення

Розгортання програмного забезпечення – це набір дій, які роблять програмний продукт доступним для використання кінцевими користувачами. Загальний процес розгортання складається із декількох взаємопов'язаних дій з можливими переходами між ними. Ці дії можуть відбуватися як з боку виробника, так і збоку споживача або на обох відразу. Оскільки кожна програмна система унікальна, важко визначити точні процеси або процедури в рамках кожної діяльності. Тому «розгортання» можна трактувати як загальний процес відповідно до певних вимог та характеристик. Розгортання може здійснюватися програмістом і під час розробки програмного забезпечення [20] рисунок 2.1.

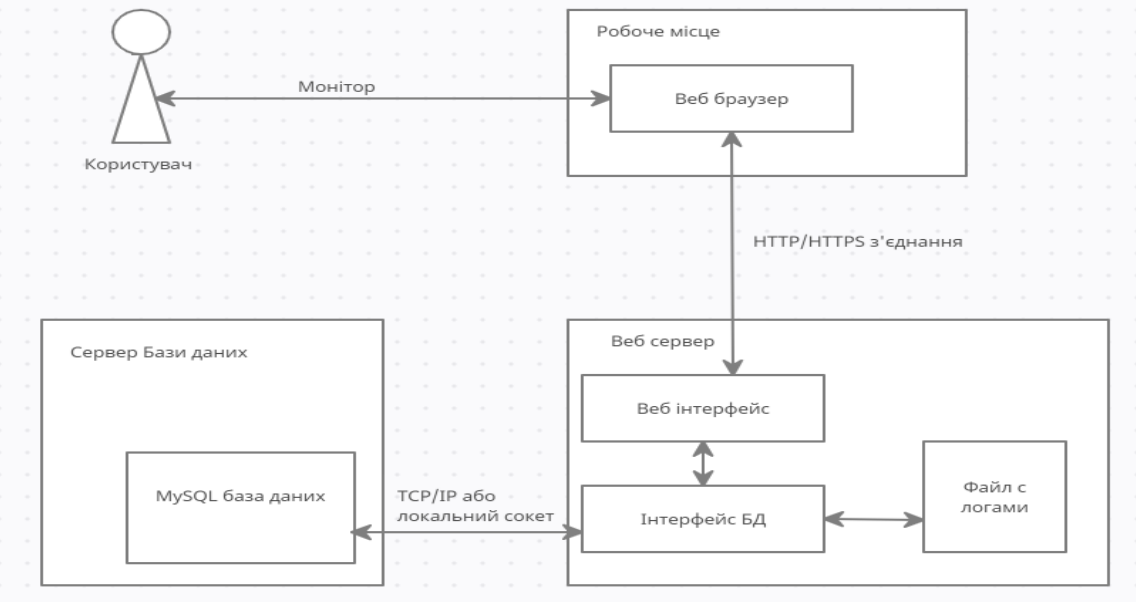

Рисунок 2.1 – Процес розгортання оточення

У простому випадку, цей процес виконується на тій же машині, що і зберігається. При промисловій розробці виділяються наступні види оточень [21] рисунок 2.2.

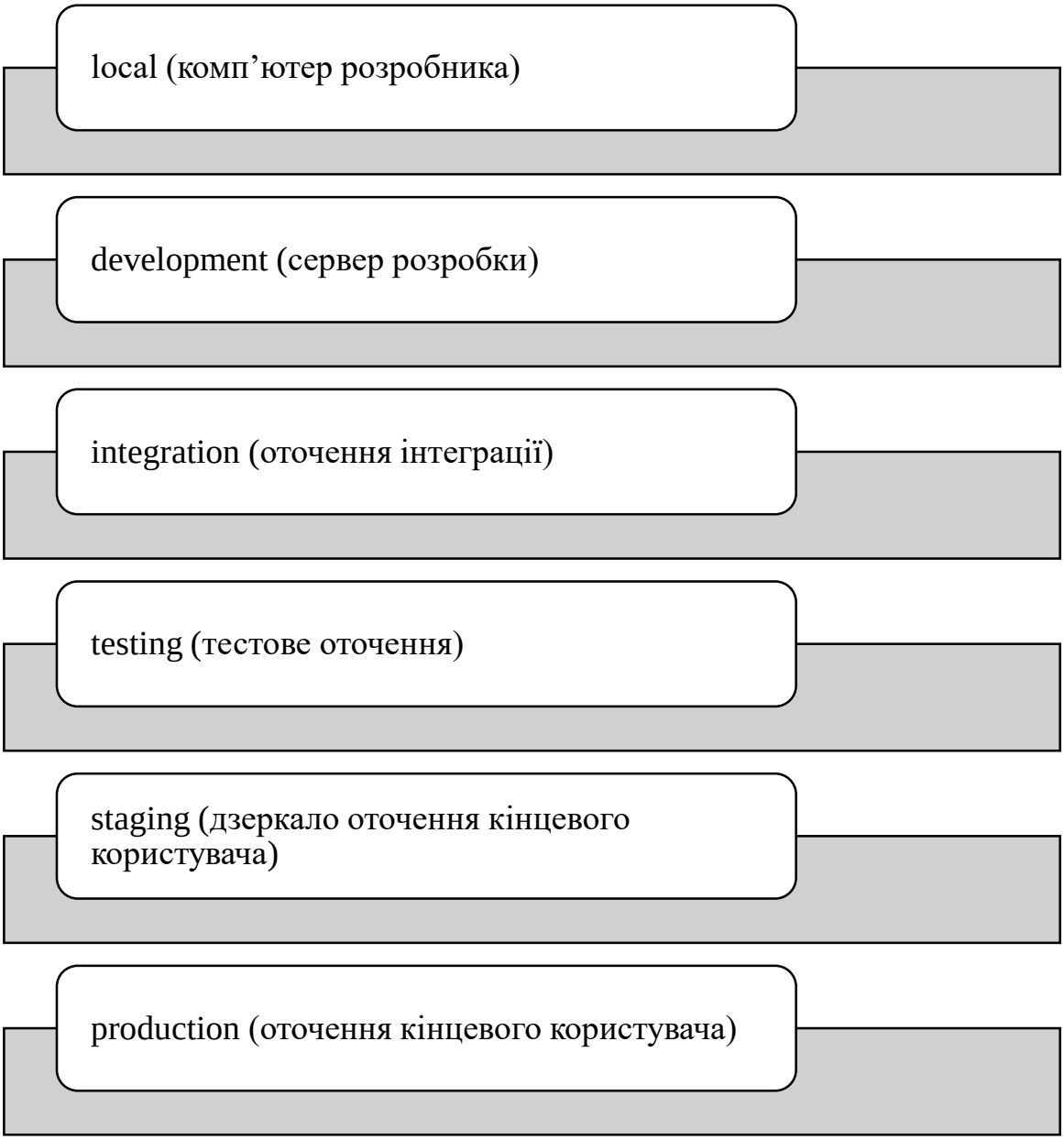

Рисунок 2.2 – Види оточень

Локальне оточення (local) – це оточення в якому розробляється софт, найчастіше це просто комп’ютер розробника.

У контексті управління версіями, особливо при участі великої кількості розробників, проводяться більші тонкі відмінності: розробник має робочу копію вихідного тексту на своїй машині і зміни вносяться в репозиторій, будучи зафіксованими в гілці.

Оточення на окремій робочій станції, на якій зміни відпрацьовані і випробувані, може називатися оточенням розробки (development) або пісочницею. Збірка копії вихідного коду репозиторія у чистому оточення є окремим етапом інтеграції, і це оточення може називатися інтеграційним оточення або оточення розробника. При неперервній інтеграції це робиться часто, так же часто, як і для кожної версії продукту.

Локальне оточення і оточення розробки – призначені для регулярного використання під час розробки. Часто в них є багато заглушок і особливостей, щоб здешевити і полегшити роботу з ними. Їх розробники можуть ламати і робити інші дії не турбуючись про наслідки.

Integration оточення — це основа для побудування неперервної інтеграції. Безперервна інтеграція (Continuous Integration) спрямована на автоматизовану перевірку інтеграції між змінами розробника і іншим кодом. У цей процес може входити статичний аналіз коду на уразливості і невідповідність загальним практикам або правилам розробки, збірка програми і автоматизоване тестування с динамічною перевіркою на уразливості.

Ціллю тестового оточення (testing) полягає в тому, щоб дозволити людям, які проводять тестування, пропускати новий і змінений код або через автоматизовані перевірки, або через ручні методи тестування. Після того, як розробник пропускає новий код и конфігурації через модульне тестування в середовищі розробки, код переноситься на одне або декілька тестових середовищ. Після неуспішного тестування, тестове середовище може видалити помилковий код із тестових платформ або повернутися до минулої версії, зв’язатися з DevOps і надати звіт про тестування і результати.

Якщо тестування пройде успішно, тестове середовище або фреймворк неперервної інтеграції, яке контролює тести, може автоматично перенести код у наступне оточення розгортання.

Дзеркало оточення кінцевого користувача (staging) – це оточення для тестування, яке в точності схоже на продакшен-оточення. Воно прагне якомога точніше відобразити реальне оточення кінцевого користувача і може підключатися до інших продакшен-сервісам і даним, таких як бази даних.

Основним призначенням staging-оточення полягає в тестуванні всіх сценаріїв установки, конфігурації, переміщення скриптів і процедур, перш ніж вони будуть застосовані на production-оточенні. Це гарантує, що всі існуючі і незначні оновлення продакшен-оточення будуть завершені якісно, без помилок і в мінімальні строки. Іншим важливим використанням staging-оточення використовується для тестування продуктивності, тестування навантаження, так як це часто відчутно для оточення.

Оточення кінцевого користувача (production) – це оточення з яким взаємодіють кінцеві користувачі. Розгортання в виробничому середовищі є найбільш чутливим кроком. Це можна виконати шляхом розгортання нового коду або шляхом розгортання змін конфігурації. Це може приймати різні форми рисунок 2.3.

Точні визначення і межі між оточеннями варіюються – тестове оточення (test) може розглядатися як частина оточення розробки (dev), оточення приймання може розглядатися як частина тестового, частина дзеркала оточення кінцевого користувача (stage), або бути окремою і так далі.

Існує багато методів розгортання програмного забезпечення, відрізняються вони кількістю оточень. Вона залежить від масштабу IT-проєкта, або від правил компанії. Але всі вони починаються з локального оточення і завершуються оточенням кінцевого користувача.

Поширеними є архітектури, які складаються із 4-х оточень development, testing, acceptance and production (DTAP) [22] і 3-х оточень development, testing, production (DTP) [23].

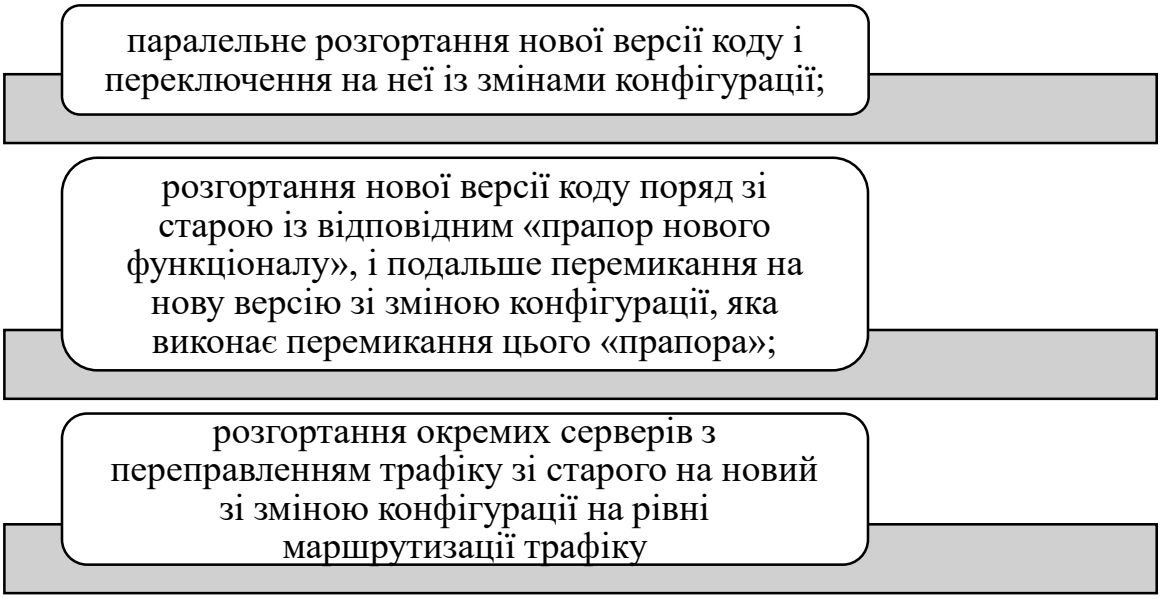

Рисунок 2.3 – Способи розгортання ПЗ на оточенні кінцевого користувача

Такий поділ зокрема підходить для серверних програм, коли сервера працюють у віддалених дата-центрах; для коду який працює на кінцевих пристроях користувача, наприклад додатків або клієнтів, останній ярус позначають як оточення користувача (USER) а перше – як локальне оточення (LOCAL).

На рисунку. 2.4 схематично зображено поділ на оточення 3-х ярусної архітектури.

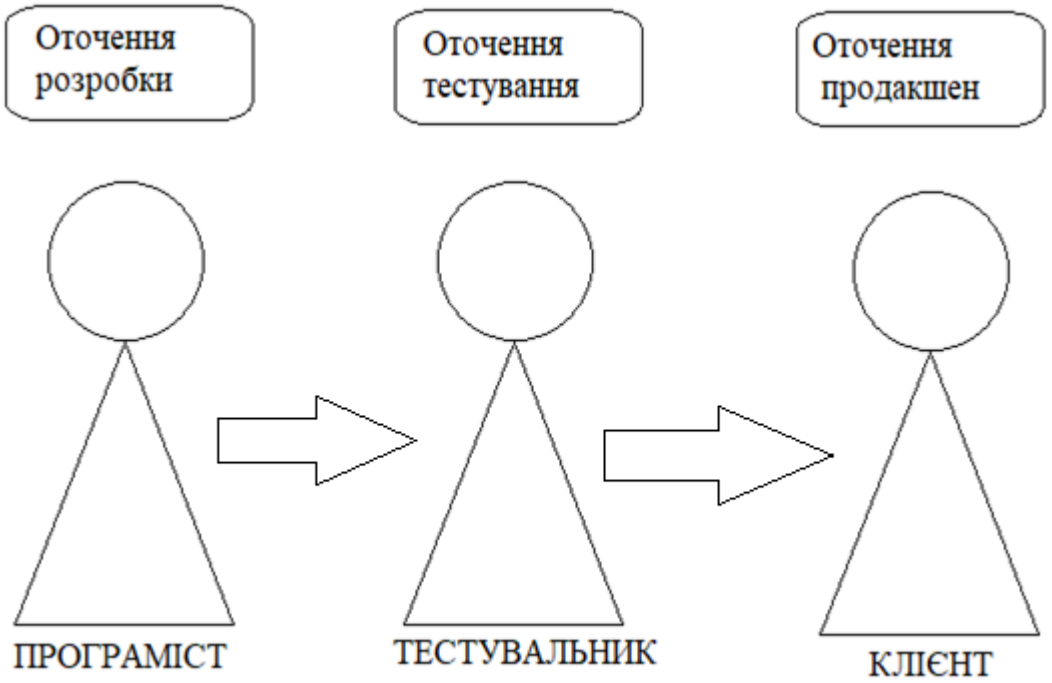

Рисунок 2.4 – Схематичне зображення поділу на оточення 3-х ярусної архітектури

Ця архітектура необхідна для забезпечення стримувань и противаг, необхідних для ефективної роботи виробничого серверного середовища з високою доступністю. Основна концепція оточень DTP проста. Розробники тестують свій код на оточені, яке призначене для розробки, щоб побачити, чи буде цей додаток працювати з іншим кодом. Простіше кажучи, це оточення представляє собою коробку для розробки.

Як тільки програміст переконається, що його частина коду готова до роботи в прайм-тайм, вона переміщується на проміжне оточення (оточення тестування). Проміжне оточення є дзеркальною копією оточення кінцевого користувача; його основною метою – протестувати завершений новий функціонал на дзеркальній копії виробничого середовища, щоб упевнитися, що він не порушує роботу існуючих частин проєкту виробничого оточення.

Ніяка фактична розробка коду не повинна відбуватися на проміжному оточенні – тільки незначне налаштування параметрів ОС або налаштування програм.

Проміжне оточення – це останній крок перед тим, як проєкт буде готовий до розгортання на продакшен оточенні. У багатьох середовищах процес остаточного затвердження слідує за етапом, щоб удосконалитися, що всі зацікавлені сторони підписуються до того, як проєкт буде запущено у виробництво. Коли додаток отримає схвалення, воно перейде у виробництво. Якщо переміщення пройшло успішно, додаток стає частиною оточення кінцевого користувача.

Тобто оточення тестування у даній архітектури виступає у ролі пісочниці і у ролі оточення для тестування, це не дуже добре, тому що на нього постійно виливається якась частина коду від розробника, що призводить до нестабільної роботи оточення. А коли воно є нестабільним, якісно протестувати новий функціонал не вийде. А отже можуть бути проблеми на оточенні кінцевого користувача.

Наступним поширеним є метод, в основі якого лежить DTAP архітектура. Тобто він має 4 оточення розгортання програмного забезпечення: development, testing, acceptance, production. Кожне оточення має своє призначення, а саме:

Development – оточення розробки. У більшості сучасних систем розробка відбувається на персональному комп’ютері окремого розробника. Це нормально, про умові, що проєкт забезпечений належною системою контролю версій. Програмістам буде потрібне повне середовище, якщо воно урізане. Усі частини системи повинні бути частиною їх системи. Для розробника, який створює веб-додаток, це означає, що йому буде потрібне як мінімум: веб-сервер та база даних, які використовуються на оточенні продакшен. Це середовище може не мати можливостей для тестування.

Testing – оточення тестування. Як тільки розробник упевнився, що його частина коду готова, вона копіюється у середовище тестування, щоб

перевірити, що він працює так як потрібно. Тут відбувається модульне тестування, потім відбувається інтеграція і регресійне тестування, щоб переконатися, що всі частини підходять один до одного і нічого раніше не було порушено.

Acceptance – оточення приймального тестування. Це оточення повинне відображати оточення продакшен. Деякі команди повторно запускають комплекс інтеграційного і регресійного тестування на цьому етапі у якості останньої перевірки, перш ніж клієнт побачить код. Приймальний тест для веб-додатків має бути там, де ваш клієнт дійсно бачить код і тестує його, використовуючи свій власний план тестування.

Production – оточення кінцевого користувача. Це останній етап розробки програмного забезпечення. Якщо замовник продукту приймає його, він розгортається у виробничому оточення, роблячи його доступним всім користувачів системи.

На рисунку 2.5 схематично зображено поділ на оточення 4-х ярусної архітектури.

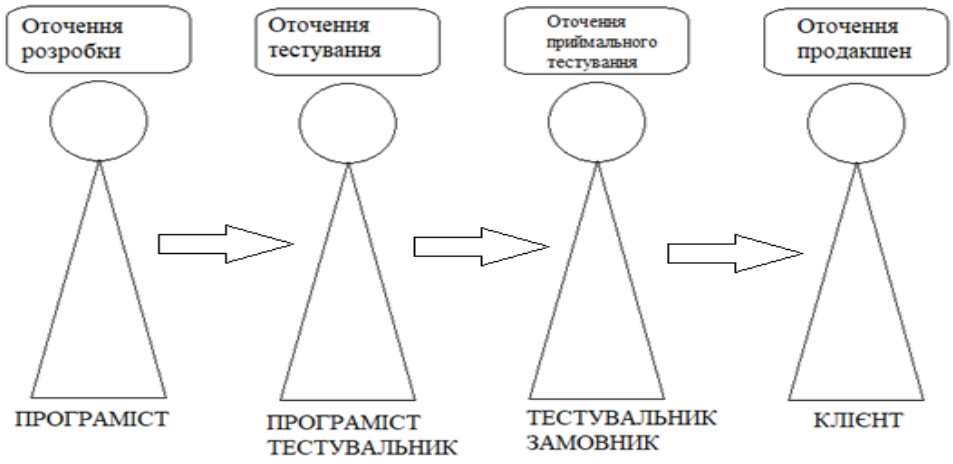

Рисунок 2.5 – Схематичне зображення DTAP архітектури

Оточення розробки та тестування однакові (дзеркально відображені) і однаково враховують приймання та виробництво. Розгорнути середовище DTAP легко, тому що на результат DTAP впливає багато факторів. Одним із

факторів є ціна та вартість ліцензування. Часто дану архітектуру використовують у навчальних цілях.

Проблема полягає в тому, що кожен реліз не виконується у визначений термін. Оточення стейджинг використовується у 2 напрямках, тестування і стабілізація. Команда тестувальників приступає до роботи тільки на оточенні стейджинг, тому що тестувати на оточенні розробки на має сенсу, через те, що воно нестабільне, тому що розробники постійно оновлюють його. Тобто, якщо тестувальник знайшов помилку, розробник її виправив і швидко доставити його на оточення стейджинг не може бо в нього не має до нього доступу. Через це і затримується процес розробки.

2.2 Удосконалений метод розгортання оточень програмного забезпечення

Удосконалений метод розгортання оточення полягає у додатковому оточенні, яке називається тестовим [24]. Метод буде складатися з таких етапів рисунок 2.6. Доступ до нього будуть мати всі розробники та тестувальники. Воно буде слугувати для тестування нового функціоналу. Таким чином оточення стейджинг буде розвантажено і воно буде використовуватися для стабілізації роботи продукту та для регресійного тестування. Таким чином повинна покращитися якість продукту та пришвидшитися розробка.

Проєкт буде складатися з :

– локальне оточення, буде слугувати як комп’ютер розробника, на якому буде реалізовуватися певний модуль якогось функціоналу. Обмеженнями даного оточення є пам'ять комп'ютера кожного розробника, його потужність а також відсутність підключення до БД;

Рисунок 2.6 – Етапи удосконаленого методу розгортання оточень

– оточення розробки, яке буде слугувати для інтеграції модулів системи. До нього будуть мати доступ усі розробники, які у будь-який момент можуть вилити свою частину коду. Обмеженнями відсутність підключення до БД та запасом пам'яті на сервері;

– тестове оточення, на якому буде відбуватися тестування нового функціоналу. Доступ до цього оточення будуть мати всі розробники і тестувальники. Обмеженнями даного оточення є відсутність підключення до реальної БД та відсутні інтеграції з деякими сервісами, їх місці розробники зазвичай роблять заглушки;

– стейджинг оточення, призначене для регресивного тестування щоб стабілізувати проєкт. На це оточення має право виливати нову версію продукт лише тех. лід. команди розробників. Тут активно працюють тестувальники. Це оточення має бути максимально схожим на оточення кінцевого користувача. Можуть буди відсутні інтеграції з деякими сервісами або їх потрібно переключити с оточення кінцевого користувача на стейджинг оточення;

– оточення кінцевого користувача, тут кінцеві клієнти вже користуються цим продуктом. Тестувальники та розробники мають до нього доступ, але потрібно бути обережними, щоб не перевантажити реальну БД тестовими даними.

Додавання нового тестового оточення складається з наступних етапів рисунок 2.7.

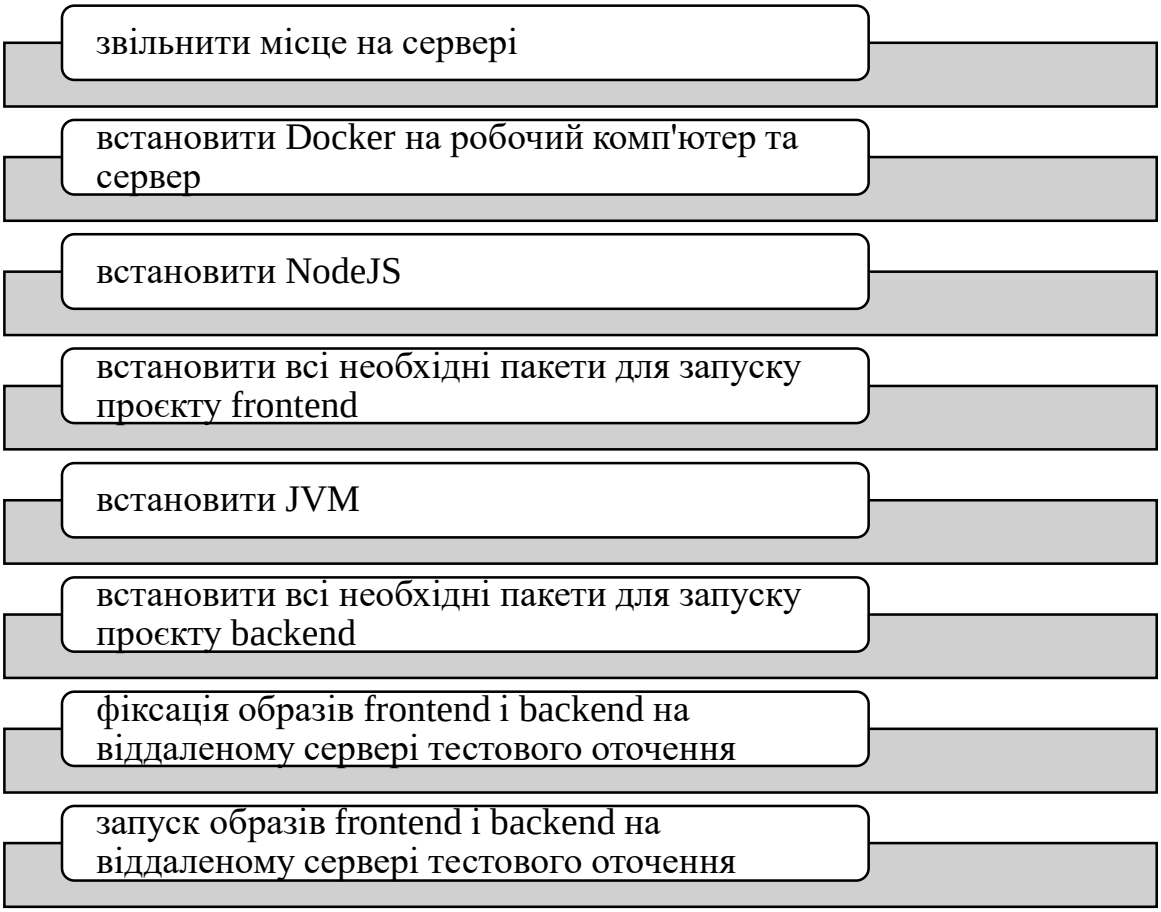

Рисунок 2.7 - Етапи додавання тестового оточення та розгортання на ньому проєкту

Додавання тестового оточення буде виконано за умови виділення оперативної пам'яті на сервері:

$$v_s \geq 2 * v_p, \tag{2.1}$$

де v_s – об'єм пам'яті сервера;

v_p – об'єм пам'яті, потрібний проекту.

Тестове оточення буде розгорнуто на сервері на якому розташовано оточення розробки, для цього DevOps повинен забезпечити місце на сервері для двох оточень. У випадку, якщо на сервері розміщується оточення розробки та тестове оточення, то мінімальний об'єм пам'яті серверу становить:

$$v_s \geq 3 * v_p, \tag{2.2}$$

3 ПЛАНУВАННЯ ПРОЄКТУ РОЗРОБКИ УДОСКОНАЛЕНОГО МЕТОДУ РОЗГОРТАННЯ ОТОЧЕННЯ

3.1 Опис додавання тестового оточення розгортання у проєкт

У додаванні нового тестового оточення розгортання будуть приймати участь наступні спеціалісти рисунок 3.1.

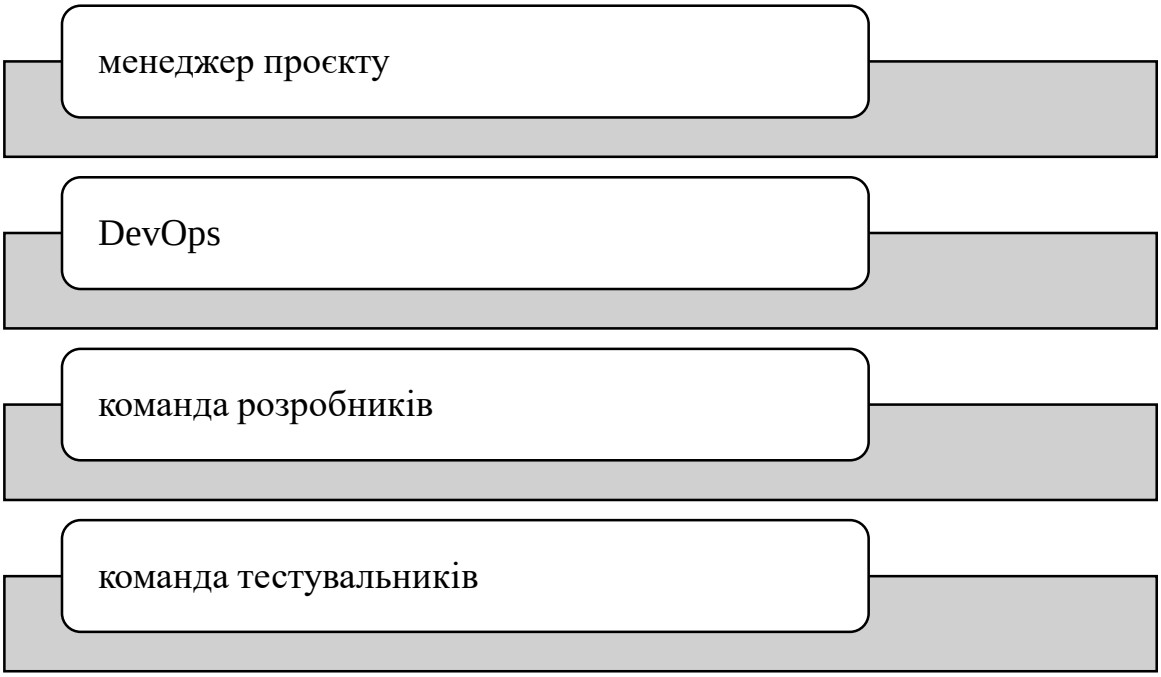

Рисунок 3.1 – Члени команди по додаванню нового оточення

Тривалість проєкту буде тривати приблизно 39 робочих днів. Використовується Agile-метолодогія розробки програмного забезпечення. Проєкт будо декомпазовано на такі, обов’язкові задачі:

– налаштування тестового оточення;

– розгортання проєкту на новому оточенні;

– тестування проєкту на оточенні;

– планування задач;

– розробка нового функціоналу;

– тестування нового функціоналу на тестовому оточенні;

– проведення регресійного тестування на staging оточенні;

– доставка нового функціоналу на оточення кінцевих користувачів;

– поверхневе тестування на оточенні кінцевих користувачів.

3.2 Статут проєкту

1. Причини додавання тестового оточення у проєкт

На сьогоднішній день на проєкту існує вагома причина, через яку розробка нового функціоналу на проєкті триває досить довго. Це пов’язано с тим, що оточення staging використовується і як тестове, і як дзеркало оточення кінцевого користувача. Основною причиною додавання додаткового оточення, покращить якість продукту та пришвидшити його розробку та тестування.

2. Сенсом додавання нового тестового оточення та спосіб його використання для розв’язання конкретної проблеми

Провівши аналаіз методів оточень розгортання програмного забезпечення усі вони мають один суттєвий недолік: усі вони використовують одне оточення у декількох цілях. Але кожне оточення повинно мати тільки одне призначення. Саме тому, було вирішено додати оточення тестування, на якому буде відбуватися первинне тестування нового функціоналу та розвантажити staging оточення, яке буде використовуватися для стабілізації проєкту.

3. Мета проєкту: додати нове оточення, покращити якість продукту, пришвидшити розробку нового функціоналу та розвантажити оточення staging.

4. Обмеження проєкту

Початок проєкту – не пізніше 1 вересня 2021 року. Завершення не пізніше 27 жовтня 2021 року.

5. Завдання проєкту та критерії їх досягнення

– додавання тестового оточення. Критерієм досягнення вважається підвищення мінімізація помилок, які потравлять на staging оточення. Пришвидшення часу випуску нового функціоналу за рахунок тестування;

– розвантаження на staging оточення. Критерієм вважаємо стабільну роботу оточення та зменшення кількості помилок на цьому середовищі.

6. Межі (зміст) проєкту

Межі проєкту визначаються переліком робіт від початку налаштування нового оточення до доставки нового функціоналу на оточення кінцевого користувача.

3.3 Планування проєкту

Щоб проєкт був успішно виконаний необхідно його грамотно спланувати. Планування – це перша функція, виконувана менеджерами, що визначає схему дій, необхідні вирішення ситуацій у майбутньому щоб досягнути цілі які поставила перед собою організація. Плани - це заздалегідь певні курси дій, що складаються у цьому, щоб спрямовувати майбутню реалізацію досягнення очікуваних цілей організації.

Таким чином, плани та планування є засобами, за допомогою яких менеджери можуть впливати на майбутнє організації. Проте планування також можна визначити як процес, з допомогою якого менеджери аналізують поточні умови, щоб визначити можливі шляхи досягнення очікуваного майбутнього стану. Планування також є функцією управління, яка створює та поєднує цілі, політику та стратегії. Зі сказаного вище, планування визначається як процес

прийняття рішення про те, які цілі будуть переслідуватися в майбутньому тимчасовому інтервалі і що буде зроблено для досягнення цих цілей.

Основною метою планування є створення універсальної підтримки та розуміння цілей, а також впровадження операційних процесів, які спрямовують організацію до їх досягнення. Результатом етапу планування є статут проєкту, створена структурна декомпозиція робіт проєкту, організаційна структура проєкту та діаграма Ганта.

При плануванні виділяємо такі об’єкти: ресурси, предметна область, якість, час виконання проєкту, вартість, ризики, планування і комунікація.

Одностайна робота всіх членів команди, які приймають участь у проєкті організовується за допомогою календарних планів або розкладів робіт проєкту, основними характеристиками яких є: терміни виконання робіт, ключові дати, тривалість робіт і ін.

Календарне планування – це процес складання й коригування розкладу, де роботи, які виконуються різними членами проєкту, пов’язані між собою в часі і з можливостю забезпечення їх різними видами трудових та апаратних ресурсів. Під час календарного планування необхідно враховувати обмеження оптимального розподілу ресурсів. [25]

Етап проекту	Початок	Тривалість	Затримка	Кінець
Постановка задачі	01.09.2021	1	0	01.09.2021
Написання ТЗ	02.09.2021	11	0	12.09.2021
Розгортання додаткового оточення	13.09.2021	3	-1	16.09.2021
Налаштування оточення	17.09.2021	5	0	22.09.2021
Тестування	23.09.2021	4	0	27.09.2021
Планування задач	28.09.2021	1	0	28.09.2021
Розробка нового функціоналу	29.09.2021	15	3	15.10.2021
Тестування нового функціоналу на новому оточенні	16.10.2021	2	-1	18.10.2021
Проведення регресійного тестування на оточенні staging	19.10.2021	5	-2	24.10.2021
Доставка нового функціоналу на оточення кінцевого користувача	25.10.2021	1	0	25.10.2021
Поверхневе тестування на оточенні кінцевих користувачів	25.10.2021	1	0	25.10.2021
Складання звіту	26.10.2021	1	0	26.10.2021

Рисунок 3.2 - Декомпозиція робіт проєкту (WBS)

Рисунок 3.2 відображає перелік робіт додавання тестового оточення. Структура декомпозиції робіт (WBS) – це орієнтована на результат ієрархічна декомпозиція робіт, які мають бути виконані членами проєкту для успішного досягнення цілей проєкт. WBS є наріжним каменем ефективного планування, виконання, контролю, моніторингу та звітності проєкту. Усі роботи, що містяться у WBS, мають бути визначені, оцінені, заплановані та закладені до бюджету.

Власники компаній та менеджери проєктів використовують WBS, щоб зробити складні проєкти більш керованими. Вона розроблена для того, щоб допомогти розбити проєкт на керовані частини, які можна ефективно оцінювати та контролювати.

Структура розбивки робіт (WBS) розробляється для встановлення загального розуміння масштабів проєкту. Вона є ієрархічним описом робіт, які мають бути виконані для завершення результатів проєкту. Кожен низхідний рівень WBS є все більш докладний опис результатів проєкту.

Підсумки – це бажані результати проєкту, такі як продукт, результат або послуга, і їх можна точно передбачити. Події, з іншого боку, може бути важко точно передбачити. Добре розроблена WBS дозволяє легко призначати елементи WBS для будь-якої діяльності проєкту. Хороша WBS повинна мати наступні характеристики:

– визначеність (може бути описана та легко зрозуміла учасникам проєкту);

– керованість (осмислена одиниця роботи, де конкретна відповідальність та повноваження можуть бути закріплені за відповідальною особою);

– можливість бути оціненою (тривалість може бути оцінена у часі, необхідному для завершення, а вартість може бути оцінена у ресурсах, необхідних для завершення);

– незалежність (мінімальна взаємодія з іншими поточними елементами або залежність від них, тобто може бути призначена на один контрольний рахунок та чітко відділена від інших пакетів робіт);

– інтегрованість (інтегрується з іншими робочими елементами проєкту та з кошторисами та графіками вищого рівня, щоб увімкнути весь проєкт);

– можливість бути вимірюваною (може бути використаний для вимірювання прогресу, має дати початку та завершення та вимірні проміжні етапи);

– адаптованість (досить гнучка, щоб додавання/видалення обсягу робіт можна було легко врахувати у структурі WBS).

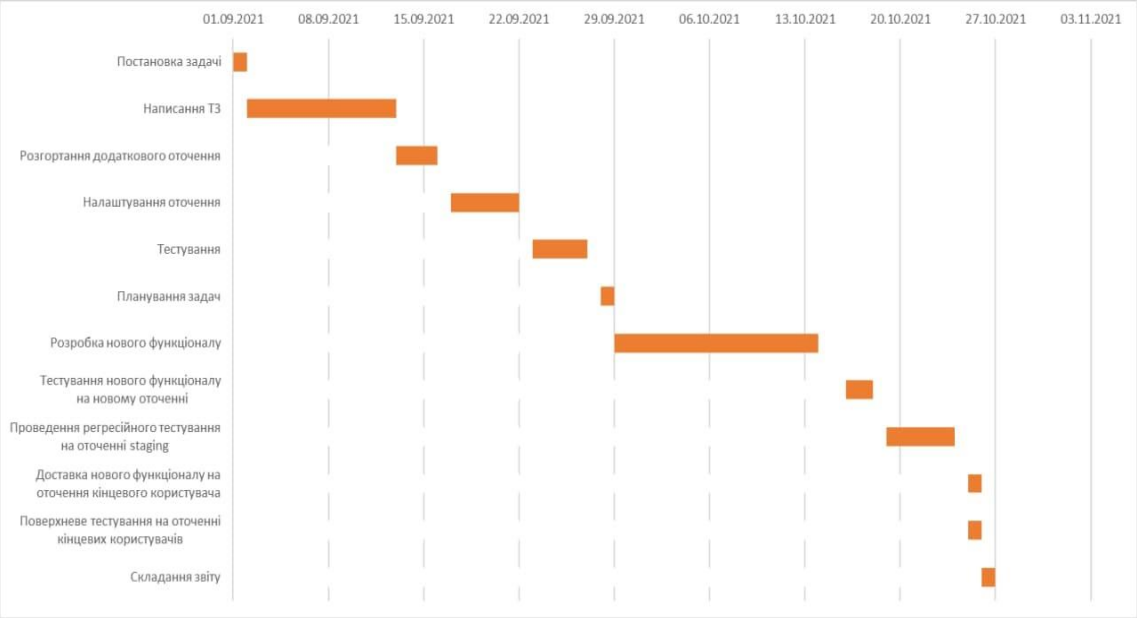

Рисунок 3.3 – Діаграма Ганта із критичним шляхом

Рисунок 3.3 містить діаграму Ганта. Діаграма Ганта – це інструмент управління проєктами, який наглядно демонструє послідовність трудових функцій на проєкті, їх тривалість та закінчення. Діаграма Ганта – це горизонтальна гістограма, що показує дати початку та закінчення робіт, їх взаємозв’язок, планування та кінцеві терміни. Це приносить корисно, в підтримці завдань, якщо змінюється тривалість робіт, коли є велика колектив та багато зацікавлених сторін.

Оскільки це формат гістограми, ви можете швидко перевірити прогрес. Ви можете легко побачити:

– візуальне відображення всього проєкту;

– терміни виконання всіх завдань;

– зв’язки та залежності між різними видами діяльності;

– стадії проєкту.

Діаграми Ганта, дають менеджерам можливість бачити робочі навантаження кожного члена команди, а також поточну та майбутню доступність, що дозволяє точніше планувати.

Деякі завдання в проєкті можна виконувати паралельно, але вони тривіальні і тому їх можна ігнорувати. Послідовність усіх завдань проєкту становить критичний шлях. Найдовший безперервний ланцюг операцій у проєкті називається критичним шляхом. Якщо послідовність завдань на цьому шляху порушує тимчасові положення, дата завершення всього проєкту буде перенесена.

Критичний шлях проєкту також є одним із ключових методів управління проєктом. У західній практиці управління він називається методом критичного шляху (CPM), а в радянській практиці управління методом критичного шляху (CPM) разом із подібними системами PERT використовується в загальній системі мережевого планування та методів управління.

Рисунок 3.4 містить зображення організаційної структури проєкту. Це тимчасова структура, що включає всіх його учасників, з урахуванням їх ролей та відносин підпорядкованості, що створюється для виконання проєкту. У ході планування проєкту OBS використовується для співвіднесення організаційних одиниць, що входять до її складу, з пакетами робіт і роботами зі складу WBS.

Так як планується додавання нового оточення, необхідно визначити людей, які будуть нести відповідальність за певні роботи, матимуть свої обов’язки, а також встановити правила взаємодії кожного учасника проєкту, відносини підпорядкованості. Спеціалісти можуть мати як повну так і часткову занятість.

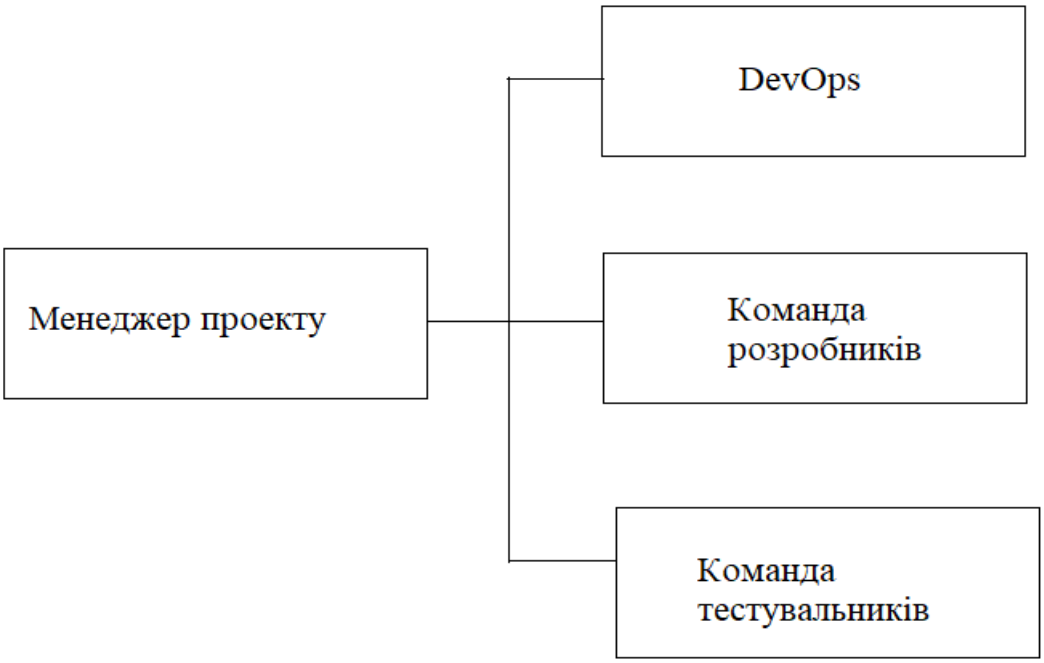

Рисунок 3.4 – Організаційна структура проєкту (OBS)

4 ПРАКТИЧНЕ ВИКОРИСТАННЯ ОТРИМАНИХ РЕЗУЛЬТАТІВ

4.1 Розгортання тестового оточення

Проєкт мав чотири оточення та 3 сервери. Сервер – це комп’ютер, який підключений до Інтернету і постійно увімкнений. Основними цілями серверу є зберігання даних та забезпечення взаємодії. Коли застосунку потрібна якась інформація, він звертається до серверу. Або коли додатку необхідно зв’язатися з іншими користувачами цього додатку, саме сервер забезпечує їх взаємодію.

Слід зазначити, що кожне оточення може бути розташоване на окремому сервері, або на сервері може одночасно розгортатися два оточення. У нашому випадку оточення розробки і тестове будуть розгорнені на одному сервері, а той сервер, який був виділений компанією буде містити в собі оточення стейджинг. Це зроблено для того щоб оточення стейджинг працювало швидко та ефективно проходила стабілізація продукту. А також, щоб зекономити кошти компанії, щоб вона не купувала нове оточення.

Розгортанням нового тестового оточення буде займатися DevOps. Він повинен проаналізувати ресурси кожного серверу та кожного оточення, щоб все працювало злагоджено, та не було перевантаження на серверах. Також на його плечі лягає відповідальність за всі інтеграції для тестового середовища (під’єднання БД з тестовими даними тощо). Також DevOps повинен кожному розробнику створити свій особистий акаунт для того, щоб вони мали доступ до тестового оточення.

Поділимо наш продукту на дві частини: frontend і backend. Де frontend написаний на ReactJS – це бібліотека JavaScript, яка розробки інтерфейсів користувача. Вона має відкритий код. Розробка на ReactJS буде відбуватися у NodeJS – це середовище виконання коду, яка перетворює JavaScript на машинну мову. Backend написаний на мові програмування Java. Java – об’єкто-орієнтована та мультиплатформена мова програмування.

Розгортання буде виконуватися за допомогою програмного забезпечення Docker, яке призначене для розгортання програмних продуктів на сервері. Його робота полягає у тому, що він створює образ віртуальної машини зі встановленими на ній та необхідними програмними забезпеченнями.

Перше, що нам необхідно – це встановити Docker. Далі необхідно зібрати образи та запустити контейнери. Першим образом буде частина frontend. Необхідно виконати такі кроки рисунок 4.1.

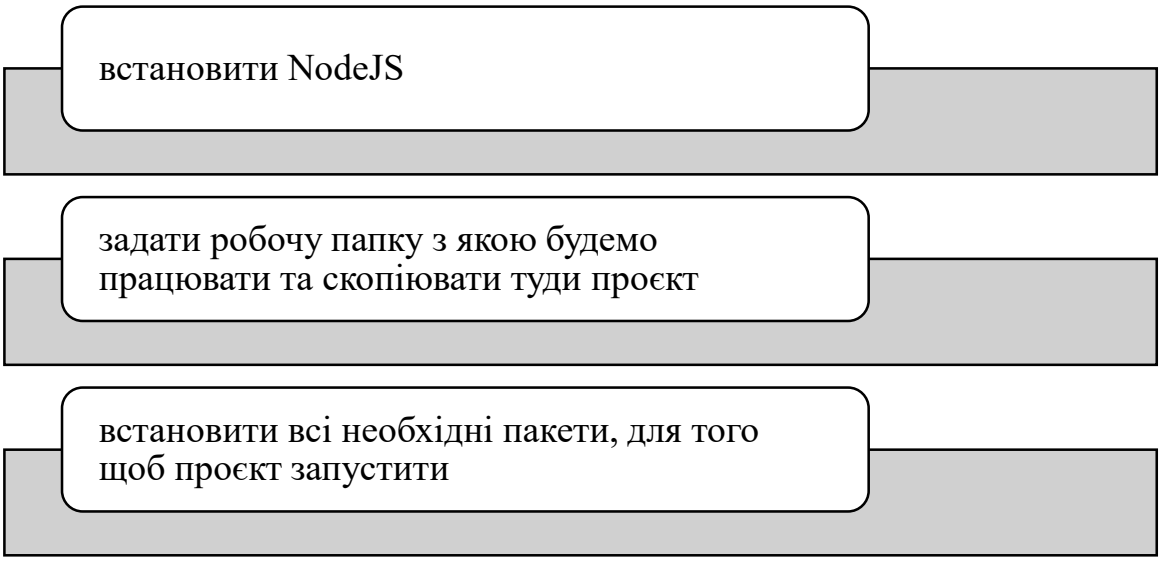

Рисунок 4.1 – Послідовність дій при створенні образу frontend

Всі кроки, які згадувалися вище, виконуються за допомогою команд, які зображені на рисунку 4.2

```
1  FROM node:alpine
2  WORKDIR /usr/app/front
3  EXPOSE 3000
4  COPY ./ ./
5  RUN npm install
6  CMD ["npm", "start"]
```

Рисунок 4.2 – Встановлення необхідних пакетів та запуск частини frontend проєкту

Далі збираємо frontend за допомогою команди рисунок 4.3.

```
4  docker build -t rebounder-chain-frontend .
```

Рисунок 4.3 – Збірка frontend

Якщо все виконано вірно, збірка проєкту має виглядати так рисунок 4.4

```
Step 7/7 : CMD ["npm", "start"]
 ---> Running in ee0e8a9066dc
Removing intermediate container ee0e8a9066dc
 ---> b208c4184766
Successfully built b208c4184766
Successfully tagged rebounder-chain-frontend:latest
```

Рисунок 4.4 – Результат збірки

Отже збірка frontend виконана успішно, отже образ сформовано. Перевірити це можна за допомогою команди рисунок 4.5. Результат даної команди рисунок 4.6.

```
4  docker run -p 8080:3000 rebounder-chain-frontend
```

Рисунок 4.5 – Запуск образу frontend

```
Auz-Yuliana:rebounder-chain-frontend xpendence$ docker run -p 8080:3000 rebounder-chain-frontend

> rebounder-chain-frontend@0.1.0 start /usr/app/front
> react-scripts start

Starting the development server...

Compiled successfully!

You can now view rebounder-chain-frontend in the browser.

  Local:            http://localhost:3000/
  On Your Network:  http://172.17.0.2:3000/

Note that the development build is not optimized.
To create a production build, use npm run build.
```

Рисунок 4.6 – Результат виконання запуску образу

Далі необхідно виконати те ж саме із частиною backend рисунок 4.7.

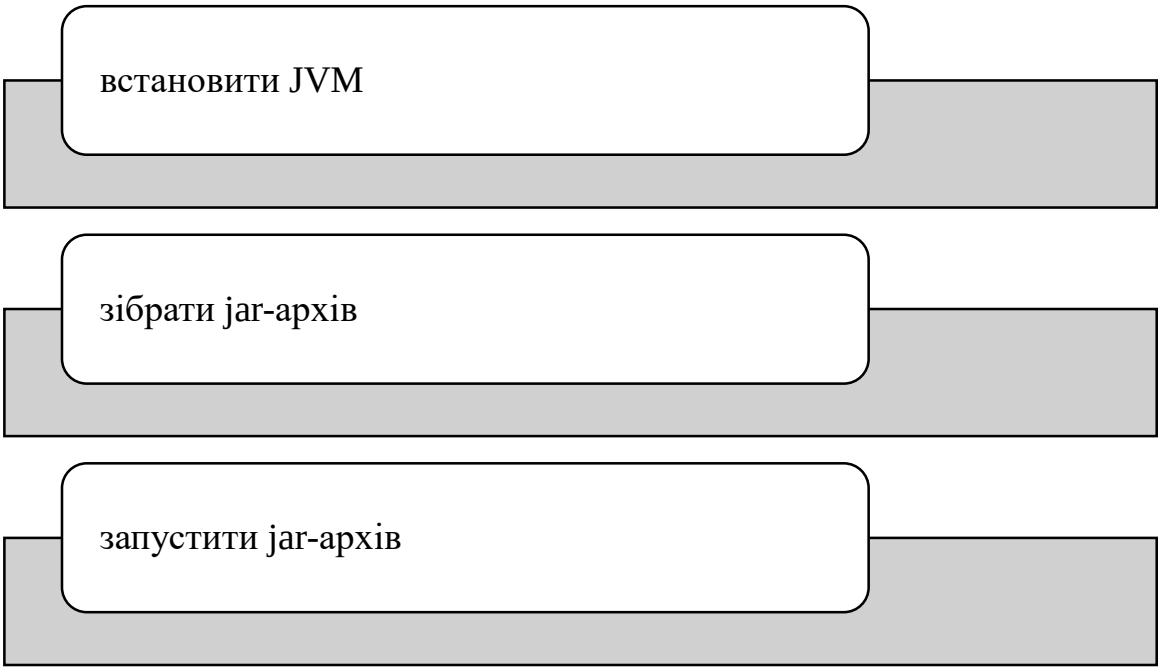

Рисунок 4.7 - Послідовність дій при створенні образу backend

Описані кроки вище зображено на рисунку 4.8 – 4.9.

```
1  FROM openjdk:8-jdk-alpine
2  LABEL maintainer="2262288@gmail.com"
3  VOLUME /tmp
4  EXPOSE 8099
5  ARG JAR_FILE=build/libs/rebounder-chain-backend-0.0.2.jar
6  ADD ${JAR_FILE} rebounder-chain-backend.jar
7  ENTRYPOINT ["java","-jar","/rebounder-chain-backend.jar"]
8
```

Рисунок 4.8 – Встановлення необхідних пакетів частини backend проєкту

```
1  docker build -t rebounder-chain-backend .
2  docker run -p 8099:8099 rebounder-chain-backend
```

Рисунок 4.9 – Збірка та запуск backend частини проєкту

Далі необхідно зібрати образи та запустити контейнери на віддаленому сервері. На сервері також повинен бути встановлений Docker, щоб усе працювало як необхідно. Потрібно авторизуватися у Docker. Першою справою необхідно кожному образу присвоїти тег рисунок 4.10. та зафіксувати у Docker рисунок 4.11.

```
1  docker tag rebounder-chain-backend xpendence/rebounder-chain-backend:0.0.2
2  docker tag rebounder-chain-frontend xpendence/rebounder-chain-frontend:0.0.1
```

Рисунок 4.10 – Команда присвоєння тега образам

```
4  docker push xpendence/rebounder-chain-backend:0.0.2
5  docker push xpendence/rebounder-chain-frontend:0.0.1
6
```

Рисунок 4.11 – Команда фіксації образів у Docker

Образи успішно додані у Docker рисунок 4.12.

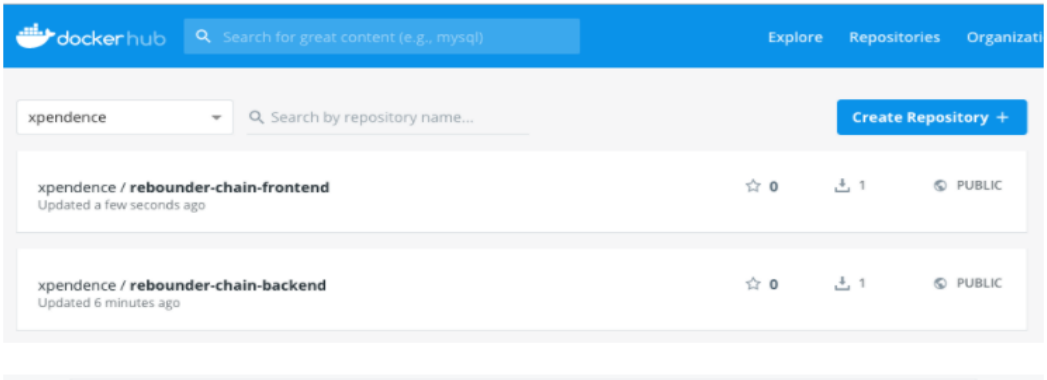

Рисунок 4.12 – Відображення доданих образів у Docker

Запускаємо образи за допомогою команди рисунок 4.13

```
1  docker run -d -p 8099:8099 xpendence/rebounder-chain-backend:0.0.2
2  docker run -d -p 8080:3000 xpendence/rebounder-chain-frontend:0.0.1
```

Рисунок 4.13 – Запуск образів на віддаленому сервері

Результат успішного розгортання проєкту на тестовому оточенні рисунок 4.14

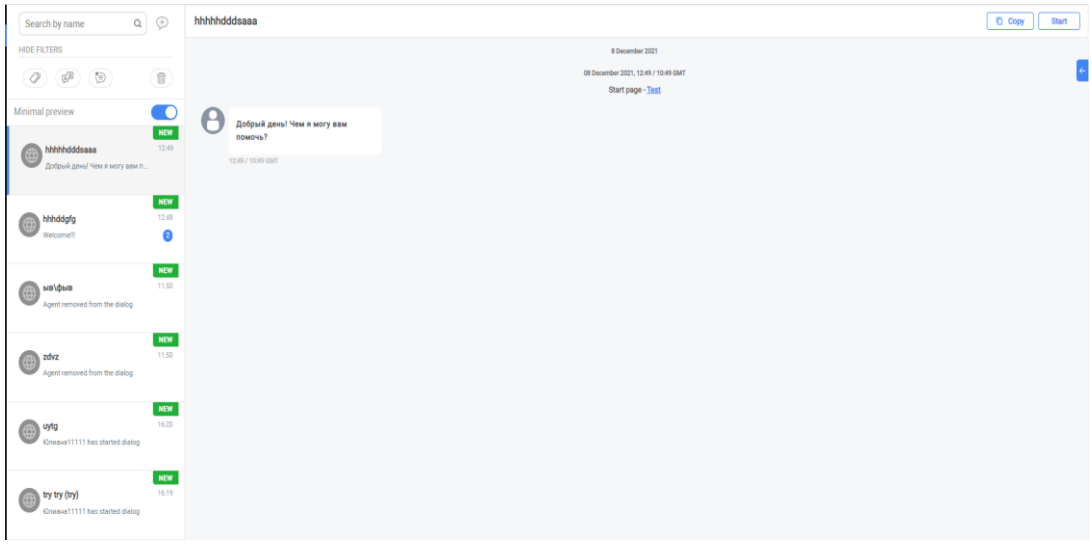

Рисунок 4.14 – Успішне розгортання проєкту на тестову оточенні

Після успішного розгортання необхідно протестувати продукт, щоб упевнитися, що він працює правильно. Тестування проводять і розробники і тестувальники. У разі виникнення проблем їх необхідно якомога швидко виправити. Перевіряти необхідно і тестове оточення, і оточення розробки, і стейджинг. Бо саме з ними відбулися зміни.

Коли тестування завершилося і всі оточення були стабілізовані, команда провела планування та обрала на майбутню ітерацію три задачі для реалізації, які були схожі по складністю реалізації.

4.2 Експериментальна перевірка розробки та тестування нового функціоналу після додавання тестового оточення

Після додавання тестового оточення, команда пройшла одну ітерацію по новому процесу оточень розгортання ПЗ. Після цього було зроблено аналіз по показникам, які наведені у Таблиці 4.1. Бачимо, що терміни на доставку нового функціоналу на оточення кінцевого користувача були прострочені на всього на 3 дні у порівнянні з минулою ітерацією.

Кількість помилок піл час тестування була значно меншою, вони швидко виправлялися розробниками та швидше перевірялися тестувальниками, бо вони були швидше доставлені на тестове оточення, так як кожний розробник має доступ до тестового оточення.

Зменшився час на тестування нового функціоналу та на регресивне тестування, через те що ці види тестувань проходили на різних оточеннях. Раніше ці етапи виконувалися паралельно. На оточення кінцевого користувача потрапила лише 1 критична помилка, було прийнято рішення виходити у реліз з нею, бо користувачам поки недоступний функціонал до якого помилка відноситься. На рисунках 4.15 – 4.20 продемонстрована динаміка зміни показників.

Таблиця 4.1 – Порівняння ітерацій до додавання тестового оточення і після

Показник	До оптимізації	Після оптимізації
Терміни ітерації	Затримка на 7 днів	Затримка на 3 дні
Кількість помилок	33	24
Час тестування нового функціоналу	5 днів	2 дні

Кінець таблиці 4.1

Показник	До оптимізації	Після оптимізації
Час регресійного тестування	5 днів	4 дні
Кількість помилок, які потрапили на оточення кінцевого користувача	2	1
Додаткові релізи	2 релізи	Відсутні

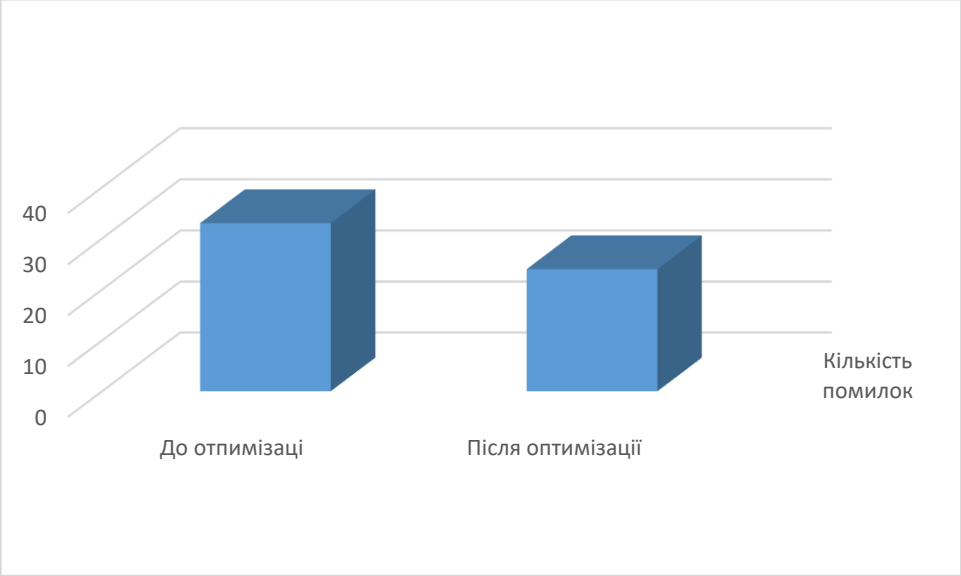

Рисунок 4.15 – Динаміка зміни кількості помилок

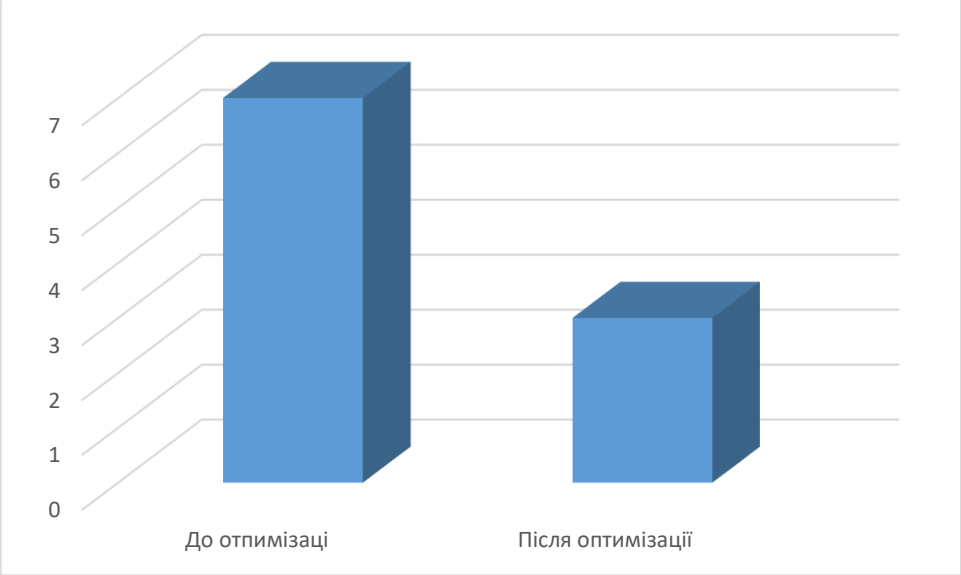

Рисунок 4.16 – Динаміка зміни затримки випуску нового функціоналу до користувача

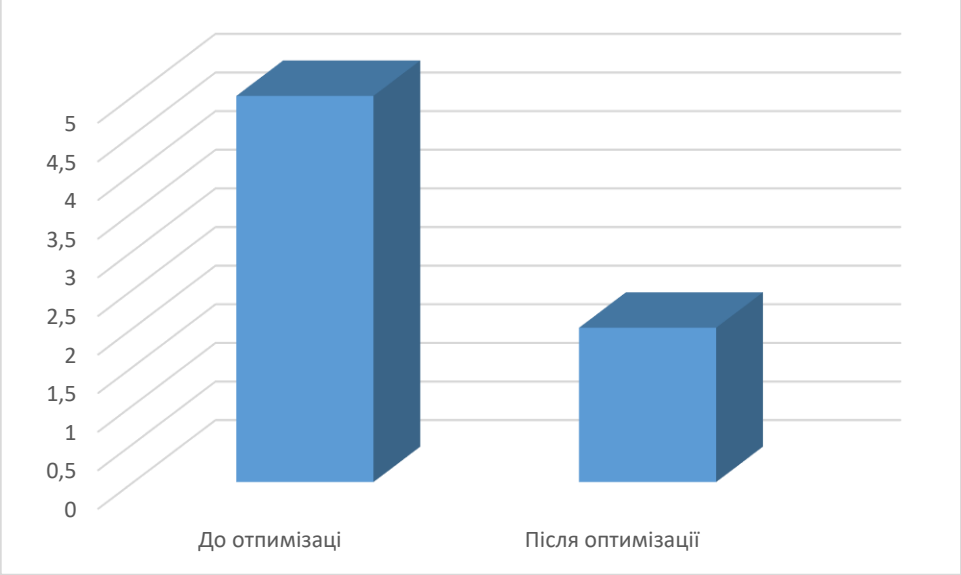

Рисунок 4.17 – Динаміка зміни часу тестування нового функціоналу

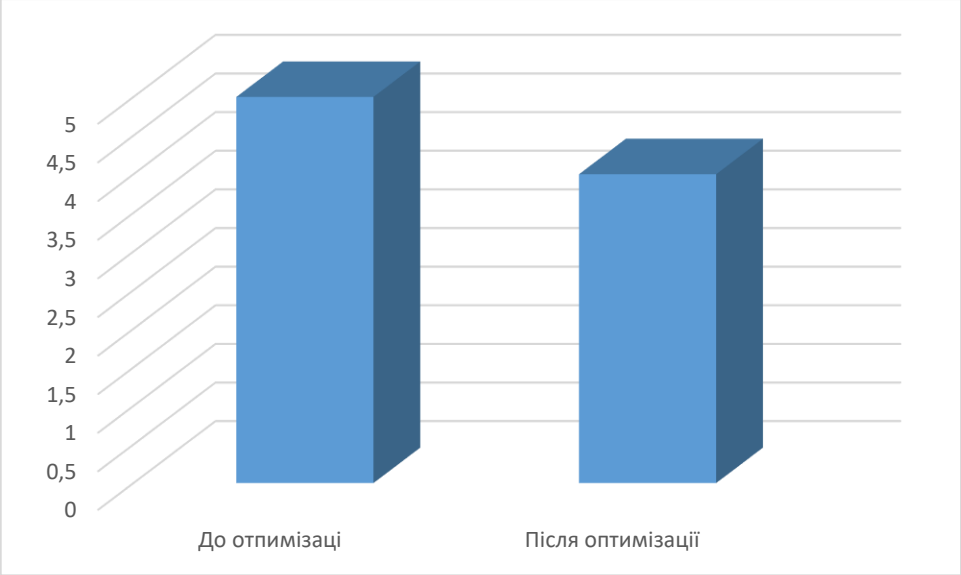

Рисунок 4.18 – Динаміка зміни часу регресійного тестування

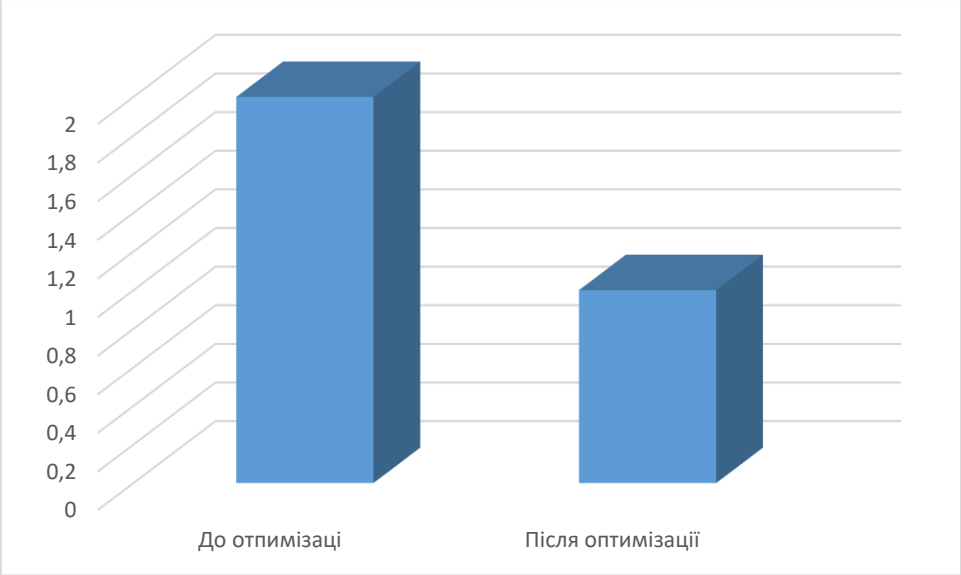

Рисунок 4.19 – Динаміка зміни кількості помилок, які потрапили на оточення користувачів

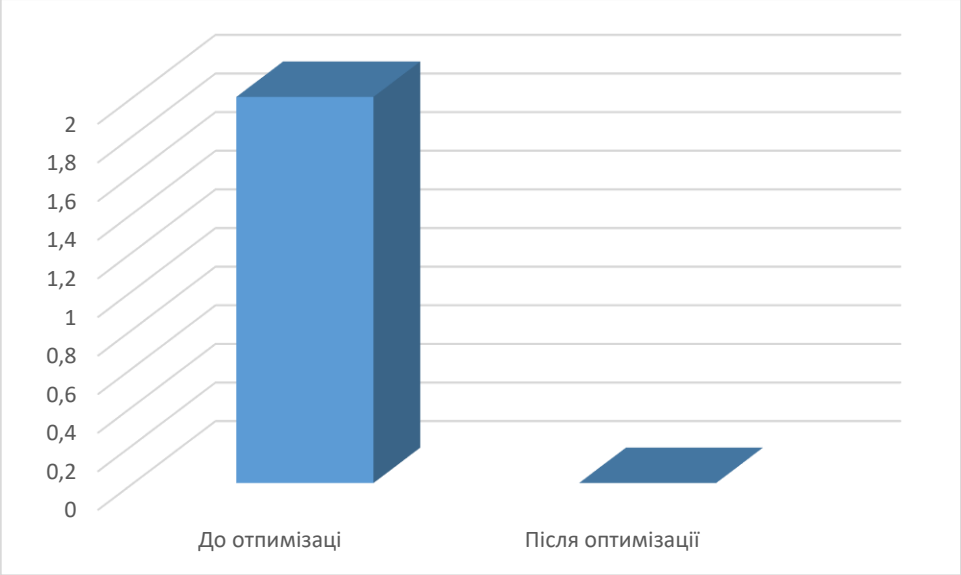

Рисунок 4.20 – Динаміка зміни кількості додаткових релізів

Можливо після додавання нового тестового оточення показники не досить вражаючі, але не все одразу. Це була перша ітерація після оптимізації. Потрібно ще декілька ітерацій щоб стабілізувати роботу усіх оточень, та щоб команда звикла до нового алгоритму роботи.

ВИСНОВКИ

В ході роботи було досліджено методи оточень розгортання при виконанні IT-проєктів. Дослідження показало, що кожне оточення повинно мати своє призначення.

Було проведено аналіз технологій безперервного постачання, безперервної інтеграції, технологій DevOps і самого процесу розгортання оточення.

Було вирушено додати нове тестове оточення у проєкт, яке буде слугувати лише для тестування нового функціоналу продукту.

Для реалізації був спланований проєкт, що включає DevOps, менеджера проєкту, команду розробників та команду тестувальників. Результатом успішного виконання було скорочення терміну реалізації нового функціоналу, пришвидшення тестування та зменшення кількості помилок.

Виконано експериментальну перевірку нового процесу розгортання проєкту, у вигляді ітерації, на основі гнучкої методології розробки програмного забезпечення.

Експериментальна перевірка показала, що пришвидшилось тестування продукту, зменшилась кількість помилок як за час тестування так і котрі потрапили на оточення кінцевого користувача. В наступних ітераціях показники повинні покращуватися.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Методичні вказівки щодо розробки та оформлення кваліфікаційної роботи (для студентів усіх форм навчання другого (магістерського) рівня вищої освіти спеціальності 122 Комп'ютерні науки освітньо-професійної програми «Управління проектами в галузі інформаційних технологій») / Упоряд.: Петров К.Е., Левикін В.М., Чалий С.Ф., Євланов М.В., Саєнко В.І., Міхнов Д.К., Міхнова А.В., Чала О.В. – Харків: ХНУРЕ, 2021. – 30 с.
2. ДСТУ 3008:2015. Державний стандарт України. Інформація та документація. Звіти у сфері науки і техніки. Структура та правила оформлення. – К.: ДП «УкрНДНЦ», 2016. – 31 с
3. ДСТУ 8302:2015. Бібліографічне посилання. Загальні положення та правила складання. / Видання офіційне. – К.: ДП «УкрНДНЦ», 2016 – 20 с.
4. А. Шеллоуэй, Дж. Р. Тротт і Г. Бівер, Lean-Agile програмне забезпечення Розробка: Досягнення гнучкості підприємства, Addison-Wesley, 2009 р. – 365 с.
5. І. Соммервіль, Програмна інженерія, 6-е видання, Pearson Education. 2001 р. – 267 с.
6. Дж. Хайсміт і А. Кокберн, Гнучка розробка програмного забезпечення: бізнес інновацій, Комп'ютер. 2001 рік. – 274 с.
7. Fowler, M., & Foemmel, M. (2006). Continuous integration. Thought-Works http://www. thoughtworks.com/Continuous Integration
8. Duvall, P. M. (2007). Continuous integration. Pearson Education India.
9. Humble, J., & Farley, D., Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education. 2010. – 315 с.

10. Larman, C., & Vodde, B. (2010). Practices for scaling lean & Agile development: large, multisite, and offshore product development with large-scale scrum. Pearson Education.
11. Naik, V. (2015). Enabling Trunk Based Development with Deployment Pipelines. https://blog.snap-ci.com/blog/2015/10/06/enabling-trunk-based-development/ (Дата звернення 01.11.2021)
12. Martin, R. C., Clean code: a handbook of agile software craftsmanship. Pearson Education. 2009. – 296 с.
13. Rasmusson, J., Long build trouble shooting guide. In Conference on Extreme Programming and Agile Methods. Springer Berlin Heidelberg. 2004. – 76 с.
14. Rogers, R. O., Scaling continuous integration. In International Conference on Extreme Programming and Agile Processes in Software Engineering. 2008. – 76 с.
15. Poole, D., Multi-Stage Continuous Integration. http://www.drdobbs.com/tools/multi-stage-continuous-integration/212201506 (Дата звернення 03.11.2021)
16. Elbaum, S., Rothermel, G., & Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM. 2014. – 245 с.
17. Krusche, S., & Alperowitz, L., Introduction of continuous delivery in multi-customer project courses. In Companion Proceedings of the 36th International Conference on Software Engineering. ACM. 2014. - 343 с.
18. Chen, L. (2015). Continuous delivery: Huge benefits, but challenges too. IEEE Software, 2015. – 54с.
19. Laukkanen, E., Paasivaara, M., & Arvonen, T., Stakeholder Perceptions of the Adoption of Continuous Integration--A Case Study. In Agile Conference (AGILE), IEEE. 2015. - 54с.
20. Розгортання програмного забезпечення https://en.wikipedia.org/wiki/Software_deployment

21. Оточення розгортання програмного забезпечення https://en.wikipedia.org/wiki/Deployment_environment (Дата звернення 02.11.2021)
22. Опис методу DTAP заснованому на 4-х ярусній архітектурі https://en.wikipedia.org/wiki/Development,_testing,_acceptance_and_production (Дата звернення 02.11.2021)
23. Опис методу DTP https://www.itprotoday.com/devops-and-software-development/development-staging-and-production-model (Дата звернення 05.11.2021)
24. Призначення тестового оточення https://radar4site.ru/blog/86-chto-takoe-testovoe-okruzhenie-v-testirovanii.html (Дата звернення 07.11.2021)
25. Що таке WBS і посібник для правильного малювання https://www.tacticalprojectmanager.com/basics/work-breakdown-structures-definition-example/ (Дата звернення 10.11.2021)