

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ДЕРЖАВНИЙ ЗАКЛАД
„ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА”

Навчально-науковий інститут фізики, математики та інформаційних
технологій

Кафедра інформаційних технологій та систем

Нелєпа Олексій Володимирович

**Інтеграційне тестування оркестратора контейнерів на прикладі
kubernetes**

кваліфікаційна робота

**здобувача вищої освіти першого (бакалаврського) рівня
освітньої програми „Інженерія програмного забезпечення”
за спеціальністю 121 Інженерія програмного забезпечення**

Особистий підпис _____ Олексій НЕЛЄПА

Науковий керівник _____ Галина КОЗУБ,
кандидат технічних наук, доцент
кафедри інформаційних технологій та
систем

Завідувач кафедри _____ Микола СЕМЕНОВ,
кандидат педагогічних наук, доцент
кафедри інформаційних технологій та
систем

Полтава – 2023

Міністерство освіти і науки України
Державний заклад „Луганський національний університет
імені Тараса Шевченка”

Факультет (інститут)

Навчально-науковий інститут фізики,
математики та інформаційних технологій

Кафедра

Інформаційних технологій та систем

Рівень освіти

перший (бакалаврський)

Спеціальність

121 „Інженерія програмного забезпечення”

(код, назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри ІТС

Микола СЕМЕНОВ

(підпис)

(ім'я, прізвище)

“ ”

2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Нелєпа Олексій Володимирович

(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) Розробка інтеграційного сценарію на прикладі
kubernetes

Керівник кваліфікаційної роботи

Козуб Г.О.

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджена наказом по університету

Від“ ” 2023 року №

2. Строк подання студентом проєкту (роботи)

3. Вихідні дані до роботи (проєкту) Дослідження інтеграційного

тестування оркестратора контейнерів на прикладі kubernetes. У результаті
виконання роботи повинно бути розроблено інтеграційні тестові сценарії

(визначаються кількісні або (та) якісні показники, яким повинен відповідати об'єкт розробки)

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно
розробити) аналіз предметної області.

вибір програмного забезпечення.

розробка інтеграційного сценарію.

(визначаються назви розділів або (та) перелік питань, які повинні увійти до тексту ПЗ)

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових
креслень)

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання „_____” _____ 2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/ п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
	Вибір теми роботи, вивчення наукової літератури, затвердження теми та керівника.	До 24 жовтня	
	Аналіз літературних джерел за темою роботи. Розробка та апробація методики дослідно-експериментальної роботи. Подання структури теоретичної частини роботи та плану експериментальних досліджень.	Другий тиждень листопада (10 листопада)	
	Робота над теоретичною частиною. Подання теоретичної частини роботи для першого читання науковим керівником.	До 15 грудня	
	Усунення зауважень, урахування рекомендацій наукового керівника. Подання теоретичної частини роботи на друге читання.	До 28 січня	
	Проведення експериментальної роботи. Поетапний аналіз та обговорення її результатів. Перевірка стану виконання роботи.	Перший тиждень березня	
	Урахування рекомендацій наукового керівника, усунення недоліків, підготовка варіанта роботи до передзахисту. Розробка презентації.	До 31 березня	
	Попередній захист роботи на кафедрі	квітень	
	Доопрацювання роботи з урахуванням рекомендацій після передзахисту. Подання роботи науковому керівникові та рецензентові на підготовку відгуку та рецензії	За 10 днів до державної атестації	
	Подання на кафедру остаточного варіанта роботи, переплетеного та підписаного автором, науковим керівником і рецензентом.	За 5 днів до державної атестації	

Студент

підпис

Олексій НЕЛЕСА

Керівник роботи

підпис

Галина КОЗУБ

АНОТАЦІЯ

Нелєпа О.В.

Тема: Інтеграційне тестування оркестратора контейнерів на прикладі kubernetes.

Спеціальність: 121 “Інженерія програмного забезпечення”

Установа: ДЗ ЛНУ імені Тараса Шевченка, 2023 р.

Бакалаврська робота містить: 64 с., 3 рис., 3 додат., 23 джерел.

Об’єкт дослідження - інтеграційне тестування оркестратора контейнерів на прикладі Kubernetes.

Предмет дослідження - інтеграційне тестування оркестратора контейнерів, зокрема на прикладі Kubernetes.

Мета роботи - дослідження інтеграційного тестування оркестратора контейнерів на прикладі Kubernetes.

Результати роботи. В роботі розглянуто різні методи інтеграційного тестування, особливості тестування оркестратора Kubernetes, а також архітектуру Kubernetes та її вплив на інтеграційне тестування. Було проведено аналіз інструментів інтеграційного тестування Kubernetes.

Висновок. В результаті розробки було отримано програмне забезпечення, яке використовується для тестування продуктивності та працездатності системи управління контейнерами кubernetes, та дозволяє легко його розширювати додатковими функціями, конфігураціями та додатковими тестовими сценаріями.

Ключові слова. ІНТЕГРАЦІЙНЕ ТЕСТУВАННЯ, ОРКЕСТРАТОР КОНТЕЙНЕРІВ, KUBERNETES, GOLANG, АРХІТЕКТУРА KUBERNE

Abstract

Nelepa O. V.

Theme: Integration testing of the container orchestrator using kubernetes as an example.

Speciality: 121 “Software Engineering”

Institution: Luhansk Taras Shevchenko National University (LTSNU), 2023.

Diploma work contains: 64 p., 3 fig., 3 appendices., 23 sources.

A research object: integration testing of the container orchestrator using Kubernetes as an example.

Subject of research: integration testing of the container orchestrator, in particular on the example of Kubernetes

The article of research is research of integration testing of the container orchestrator on the example of Kubernetes.

Job performances. The paper discusses various methods of integration testing, features of testing the Kubernetes orchestrator, as well as the Kubernetes architecture and its impact on integration testing. Kubernetes integration testing tools were analyzed.

Conclusion. As a result of the development, software was obtained that is used to test the performance and performance of the Kubernetes container management system, and allows you to easily extend it with additional functions, configurations, and additional test scenarios.

Keywords. INTEGRATION TESTING, CONTAINER ORCHESTRATOR, KUBERNETES, GOLANG, KUBERNE TE ARCHITEC

Міністерство освіти і науки України
Державний заклад “Луганський національний університет
імені Тараса Шевченка”

Факультет (інститут)	Навчально-науковий інститут фізики, математики та інформаційних технологій <small>(повна назва)</small>
Кафедра	Інформаційних технологій та систем <small>(повна назва)</small>

ТЕХНІЧНЕ ЗАВДАННЯ
на виконання програмної розробки (ПР):
“ІНТЕГРАЦІЙНЕ ТЕСТУВАННЯ ОРКЕСТРАТОРА
КОНТЕЙНЕРІВ НА ПРИКЛАДІ KUBERNETES”

Полтава 2023

ЗМІСТ

ВСТУП	8
1. ХАРАКТЕРИСТИКА ОБ'ЄКТА	8
2. ПРИЗНАЧЕННЯ ТОВАРІВ	8
3. ОСНОВНІ ВИМОГИ ДО ПРОГРАМНОГО КОМПЛЕКСУ	8
4. ТЕХНІКО - ЕКОНОМІЧНІ ВИМОГ ДО КІНЦЕВОГО ПРОДУКТУ	9
5. ВИМОГИ ДО МАТЕРІАЛІВ І КОМПЛЕКТУЮЧИХ	9
6. ЕТАПИ ВИКОНАННЯ ПР	9
7. ПРИЙОМ	10
8. ПОРЯДОК ВНЕСЕННЯ ЗМІН ДО ТЕХНІЧНЕ ЗАВДАННЯ, ЩО ЗАТВЕРДЖЕНО	11

ВСТУП

1.1 Найменування: Інтеграційне тестування оркестратора контейнерів.

1.2 Шифр ПР: АДР -1

1.3 Підстава для виконання ПР: Підставою для виконання даної розробки є заява від замовника.

1.4 Терміни розробки:

1.4.1 Початок 15 жовтня 2022 р.

1.4.2 Закінчення 20 квітня 2023 р.

1.5 Фінансується за рахунок коштів замовника. Умови фінансування - за договором № 13 / а і протоколу узгодження ціни № 13 / б.

1. ХАРАКТЕРИСТИКА ОБ'ЄКТА

1.1. Розроблений товар має працювати правильно.

1.2. До вхідної інформації належать вимоги замовника щодо додатку.

2. ПРИЗНАЧЕННЯ ТОВАРІВ

2.1. Призначення: перевірка на працездатність.

2.2. Основні критерії ефективності

2.2.1. Працездатність.

3. ОСНОВНІ ВИМОГИ ДО ПРОГРАМНОГО КОМПЛЕКСУ

3.1. Загальні вимоги

3.1.1. Повинно працювати на системах macos і ubuntu;

3.1.2. Має працювати правильно та без багів.

3.2. Додаткові вимоги

3.2.1. Вимоги до ліцензійного ПЗ не передбачаються і вирішуються замовником

3.3. Вимоги до складу і архітектури

3.3.1. Розробник самостійно вибирає склад і виконує розробку архітектури ПР

3.3.2. Особливих умов до складу та архітектури ПР не передбачено

3.4. Вимоги до якості і надійності

3.4.1. Повинен надійно працювати.

3.4.3. Розробник гарантує роботу без збоїв та переналаштувань.

3.5. Вимоги до експлуатації

3.5.1. Розробник використовує macos або ubuntu що надійно працює.

4. ТЕХНІКО - ЕКОНОМІЧНІ ВИМОГ ДО КІНЦЕВОГО ПРОДУКТУ

Вартість робіт по розробці даної ПР визначається згідно з договором на розробку. Вартість пропонованих аналогів повинна забезпечити економічну доцільність їх застосування.

5. ВИМОГИ ДО МАТЕРІАЛІВ І КОМПЛЕКТУЮЧИХ

5.1. Вимоги до екологічної безпеки при експлуатації.

Не пред'являються.

5.2. Спеціальні вимоги до кінцевого продукту.

Не пред'являються.

5.3. Вимоги до безпеки для населення при експлуатації продукції.

Не пред'являються.

6. ЕТАПИ ВИКОНАННЯ ПР

Етапи виконання ПР можуть уточнюватися згідно календарного плану робіт за погодженням між замовником і виконавцем

№	Етапи виконання роботи	Термін виконання і обсяг робіт	Звітні матеріали
1	Аналіз розробки програмного комплексу та розробка першої версії. Аналіз вимог. Розробка структури. Попереднє тестування.		Фрагмент програмного комплексу на ЕОМ замовника, який виконує всі основні функції і звітна документація п.8.2
2	Коригування структури. Розробка допоміжних функцій. Розробка остаточної версії програмного комплексу і його обробки. Тестування.		Готовий програмний комплекс на ЕОМ замовника і звітна документація п.8.2
3	Доопрацювання окремих модулів і навчання користувачів. Розробка звітних матеріалів по п.8 цього ТЗ		Звітні матеріали згідно з пунктом 8.

7. ПРИЙОМ

7.1. Необхідні вимоги для впровадження ПР і завершення робіт.

Оцінка результатів розробки і доцільність її продовження здійснюється замовником за поданням наступних матеріалів:

- встановлено програмний комплекс на ЕОМ замовника;
- перелік файлів на резервному носії;
- короткий опис роботи ПР і опис всіх файлів, які необхідні для роботи ПР.
- Технічне завдання
- Пояснювальна записка
- Методика тестування

7.2. Перелік звітних документів, необхідних для прийняття етапів роботи:

- короткий опис результатів етапу у вигляді анотованого звіту (для 1 та 2 етапів);
- частковий програмний комплекс на ЕОМ замовника згідно календарного плану робіт;
- акт приймання продукції.

Звітні матеріали подаються у вигляді звітів на папері відповідно до "ДСТУ 3008-2015. Державний стандарт України. Документація. Звіти в сфері науки і техніки. Структура та правила оформлення."

7.3. Загальний перелік до прийому звітних документів, макетів, експериментальних зразків.

До прийому пред'являються: акт здачі-приймання продукції, акт впровадження ПР.

7.4. Тестування ПР

Тестування виконується в розділі "Методика тестування", яка розробляється виконавцем і затверджується замовником.

8. ПОРЯДОК ВНЕСЕННЯ ЗМІН ДО ТЕХНІЧНОГО ЗАВДАННЯ, ЩО ЗАТВЕРДЖЕНО

Дане технічне завдання може уточнюватися в процесі розробки ПР при узгодженні сторін з оформленням доповнень до ТЗ.

Міністерство освіти і науки України
Державний заклад “Луганський національний університет
імені Тараса Шевченка”

Факультет (інститут)	Навчально-науковий інститут фізики, математики та інформаційних технологій
	<hr/> (повна назва)
Кафедра	Інформаційних технологій та систем
	<hr/> (повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА

до дипломного проєкту (роботи)

БАКАЛАВРА

(освітній ступень)

“ІНТЕГРАЦІЙНЕ ТЕСТУВАННЯ ОРКЕСТРАТОРА КОНТЕЙНЕРІВ
НА ПРИКЛАДІ KUBERNETES”

Полтава 2023

ЗМІСТ

ВСТУП.....	14
РОЗДІЛ 1. АНАЛІЗ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	16
1.1. Огляд наявних робіт у галузі інтеграційного тестування Kubernetes ...	17
1.2. Огляд технології контейнеризації та оркестратора Kubernetes	18
1.3. Інтеграційне тестування	19
1.4. Основні підходи до інтеграційного тестування	21
1.5. Особливості інтеграційного тестування оркестратора Kubernetes.....	22
1.6. Архітектура Kubernetes та її вплив на інтеграційне тестування.....	24
1.7. Інструменти для інтеграційного тестування Kubernetes	28
1.8. Висновки до розділу 1	29
РОЗДІЛ 2. ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ	30
2.1. Мова програмування і програмне середовище.....	30
2.2. Docker як інструментарій для управління контейнерами	31
2.2. Висновки до розділу 2	34
РОЗДІЛ 3. ПРОЄКТУВАННЯ ТА КОНСТРУЮВАННЯ ТЕСТІВ	35
3.1. Розробка тестів для інтеграційного тестування Kubernetes.....	35
3.2. Процес проведення інтеграційного тестування Kubernetes	35
3.3. Аналіз результатів інтеграційного тестування Kubernetes	37
3.4. Порівняння результатів із результатами інших досліджень.....	41
3.5. Основні результати дослідження	42
3.6 Висновки до розділу 3	43
ВИСНОВКИ	45
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	46
ДОДАТКИ	48

ВСТУП

В даний час широкий спектр додатків, включаючи хмарні обчислення, масштабовані веб-додатки, розробку і тестування програмного забезпечення тощо, використовують технології контейнеризації та оркестратори контейнерів. Kubernetes, один з найпоширеніших оркестраторів контейнерів, пропонує гнучке та надійне рішення для керування додатками в контейнерах на кластері серверів. Однак, щоб переконатися, що вся система працює належним чином, необхідне ретельне перевірка, в тому числі тестування інтеграції, оскільки розмір кластера і кількість активних додатків зростають.

Об'єкт дослідження оркестратор контейнерів Kubernetes, що широко розглядається як галузевий стандарт для управління контейнерними робочими навантаженнями, є основним об'єктом цього дослідження. Мета дослідження - вивчити численні інтеграційні можливості Kubernetes та зрозуміти, як він працює з іншими компонентами та сервісами розподіленої системи.

Предмет дослідження розглядається тестування інтеграції Kubernetes, наскільки добре вона може координувати контейнерні додатки, контролювати мережу, працювати зі сховищем і взаємодіяти із зовнішніми сервісами. У дослідженні розглядаються точки інтеграції, які необхідно ретельно протестувати, щоб гарантувати оптимальну продуктивність і відмовостійкість, заглиблюючись у внутрішні аспекти архітектури Kubernetes.

Мета роботи є дослідження тестування інтеграції в середовищі Kubernetes та покращення знань про нього. Ця робота прагне представити ефективні методології та методи оцінки інтеграційних аспектів Kubernetes, тим самим підвищуючи загальну стабільність і надійність контейнерних систем шляхом виявлення критичних точок інтеграції та потенційних труднощів.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

1. Аналіз архітектури Kubernetes і точок інтеграції;
2. Розробка варіантів використання і сценаріїв.

Методи дослідження - техніко-економічний з використанням комп'ютерних технологій, технічний аналіз, методи моделювання інформаційних процесів.

Структура і обсяг роботи.

Робота складається з вступу, трьох розділів, висновків списку використаних джерел, додатків. Обсяг роботи становить 64 сторінки, обсяг використаної літератури – 23 джерела.

Перший розділ «аналіз предметної області» визначає поняття концепції. Досліджено можливості інтеграційного тестування оркестратора та складністю його складових частин. Також було прийнято до уваги інструменти для автоматизації інтеграційного тестування Kubernetes.

Другий розділ присвячено вибору необхідної мови програмування. При виборі враховувалися ряд вимог які сформульовано в першому розділі.

У третьому розділі, розглядається процес розробки та реалізації інтеграційного тестування оркестратора контейнерів на прикладі Kubernetes. В цьому розділі детально описуються етапи та кроки, які були здійснені для підготовки середовища та створення тестових сценаріїв.

Додатки містять методику тестування, керівництво користувача на виконання програмної розробки та коди сценаріїв тестування.

- Методика тестування: В цьому додатку наведена вичерпна методика, що розкриває всі аспекти тестування оркестратора контейнерів. Описані різні типи тестів, їх послідовність, параметри та вимоги до тестування.
- Керівництво користувача: У цьому додатку надано детальні вказівки щодо розробки програмного коду для інтеграційного тестування оркестратора контейнерів з використанням Kubernetes.
- Коди сценаріїв тестування: У цьому додатку подані реалізовані коди сценаріїв тестування для оркестратора контейнерів Kubernetes.

РОЗДІЛ 1

АНАЛІЗ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Однією з найпопулярніших технологій у розробці програмного забезпечення є контейнеризація. Ви можете ізолювати певні програми та їхні залежності, оскільки це забезпечує повну переносимість у різних умовах. Одним з найпопулярніших інструментів контейнеризації є Docker.

Система управління контейнерами Kubernetes orchestrator автоматизує розгортання, масштабування та обслуговування контейнерів. Властивості розподілених мереж і сховищ даних забезпечують чудову масштабованість і надійність. Контейнеризація та Kubernetes, які також значно підвищують надійність та масштабованість програм, дозволяють швидше та ефективніше розгорнути додатки в різних контекстах.

Ми розглянемо вплив Kubernetes та контейнеризації на інтеграційне тестування більш детально в наступних розділах. Одним з найкращих інструментів для оркестрування контейнерів є Coober. Ви можете запускати контейнерні програми, керувати ними, розвивати та автоматизувати процес. Здатність динамічно створювати програми залежно від попиту є однією з ключових переваг Kubernetes.

Для керування контейнерами та компонентами Kubernetes використовує інтерфейс API. За допомогою цього API ви можете керувати веб-налаштуванням, обмеженнями ресурсів та іншими адміністративними діями. Крім того, Kubernetes дозволяє розгорнути програми на декількох хостах і надає рішення для архітектури мікросервісів.

Додатки масштабуються і розгортаються в умовах високої доступності за допомогою технології контейнеризації. Вони можуть функціонувати незалежно в різних середовищах, розділяючи свої залежності за допомогою контейнерів. Без належного рішення управління та підтримка декількох контейнерів може бути складним завданням.

Розробники програмного забезпечення та інженери повинні бути обізнані з технологіями оркестрування та контейнеризації, які компанії все частіше використовують для доставки своїх додатків.

1.1 Огляд наявних робіт у галузі інтеграційного тестування Kubernetes

Створення програм у контейнерному середовищі вимагає інтеграційного тестування. У цій галузі було проведено багато практичних досліджень і досліджень. Багато авторів зосереджуються на розробці методів тестування, які підвищують надійність і якість системи через складність Kubernetes. Згідно з одним дослідженням команди розробників Google, віртуальні клієнти можна використовувати для перевірки інтеграції. Використання цієї тактики дозволяє швидше виконувати тести та ефективніше керувати ресурсами.

Дослідження, проведене членами команди розробників Red Hat, зосереджено на розробці автоматизованого фреймворку тестування Кубера. Щоб охопити широкий спектр системних функцій і сценаріїв, вони запропонували використовувати метод, заснований на створенні довільних тестових сценаріїв інших дослідженнях досліджуються різні частини Kubernetes, такі як управління мережею та кластери.

Загалом, створення ефективних методів тестування стає все більш важливим під час розробки розподілених систем і додатків. Коли мова заходить про перевірку Кубера, багато статей зосереджені на функціональному тестуванні компонентів системи, таких як планувальник, контролери реплікації та сервер API. Окремі компоненти повинні добре працювати один з одним і працювати разом, що називається тестуванням цілісності.

На офіційному сайті Kubernetes також доступна інструкція з тестування системи. У цьому есе розглядаються інструменти для випробування, такі як Ginkgo і Gomega, і розглядаються різні методи випробування, включаючи

інтеграційне тестування. Крім того, було проведено дослідження інтеграційного тестування контейнерних систем, таких як Docker і Kubernetes.

У статті «Порівняльне дослідження Docker Swarm та Kubernetes» автори розглянули Docker Swarm і Kubernetes з точки зору різних компонентів, таких як масштабованість, управління та безпеку. Крім того, є дослідження, які зосереджуються на різних аспектах випробування Kubernetes, як-от Ренато Лосарі «KubeCon + CloudNativeCon Europe 2021 — Тестування безперервної інтеграції нативних додатків Кубера за допомогою GitHub Actions та E2E-тестів».

1.2 Огляд технології контейнеризації та оркестратора Kubernetes

Контейнеризація є однією з найпоширеніших технологій у розробці програмного забезпечення. Вона гарантує повну переносимість у різних середовищах, що дозволяє відокремити певні програми та їхні залежності. Docker є одним із найвідоміших інструментів для контейнеризації.

Оркестратор Кубера, система керування контейнерами, автоматизує розгортання, масштабування та обслуговування контейнерів. Характеристики зберігання даних і розподіленої мережі забезпечують виняткову масштабованість і надійність. Додавання можна розгортати швидше та ефективніше в багатьох умовах завдяки контейнеризації, яка також забезпечує чудову стабільність і масштабованість програм.

У наступних розділах ми розглянемо більш детально вплив Кубер і контейнеризації на тестування інтеграції.

Kubernetes використовує інтерфейс API, щоб керувати контейнерами та компонентами. Це дозволяє вам виконувати завдання адміністрування, такі як онлайн-конфігурація, обмеження ресурсів тощо. Крім того, він пропонує рішення для мікросервісної архітектури та дозволяє розгортати програми на кількох хостах.

За допомогою технологій контейнеризації додатки можна масштабувати та розгортати в середовищах з високою доступністю. За допомогою

контейнерів компоненти та їхні залежності можуть бути захищені, що дозволяє працювати незалежно в різних умовах. Тим не менш, обслуговування та управління численними контейнерами в цьому випадку може бути складним без відповідного рішення. Інженери та розробники програмного забезпечення повинні знати про технологію контейнеризації та оркестру Kubernetes, яку використовує все більша кількість компаній для розгортання своїх додатків.

1.3 Інтеграційне тестування

Тестування програмного забезпечення під назвою "інтеграційне тестування" передбачає перевірку того, як окремі компоненти системи, що залежать один від одного або мають спільну функціональність, взаємодіють один з одним. Основна мета інтеграційного тестування - перевірити, що окремі компоненти програми взаємодіють і сумісні один з одним належним чином.

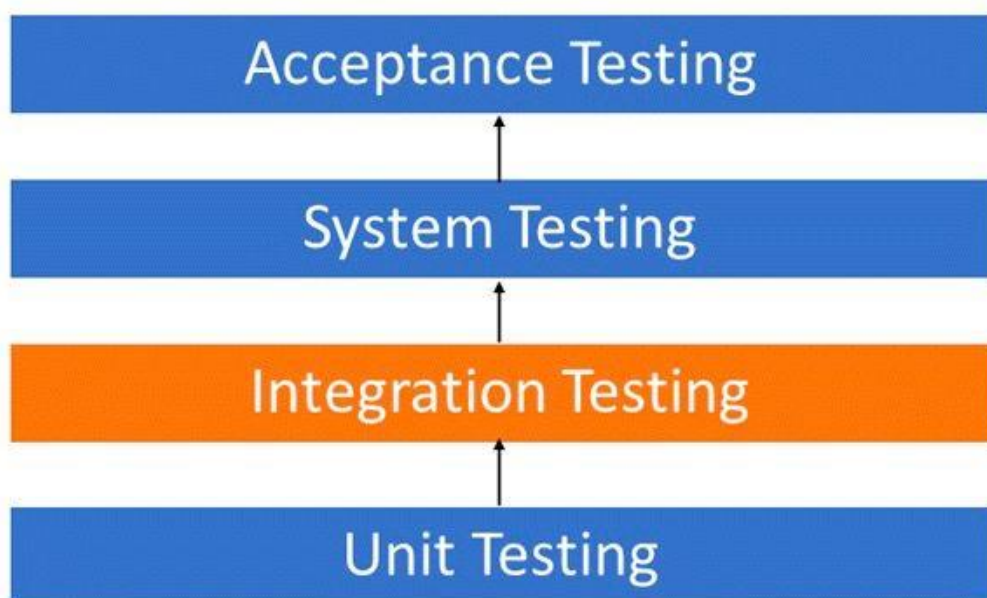


Рис. 1.1 Типи тестування

Для інтеграційного випробування часто використовують автоматизовані набори тестів для запуску складних тестових сценаріїв і пошуку проблем у

взаємодії між компонентами. У цій ситуації випробування може виконуватися як на системі в цілому, так і на окремих її компонентах.

Інтеграційне тестування в контексті оркестратора контейнерів може включати оцінку того, наскільки добре контейнери взаємодіють один з одним, а також з іншими елементами системи, включаючи бази даних, мережеві ресурси та інші сервіси. Щоб правильно виконати інтеграційне тестування, ви повинні добре розуміти роботу оркестратора контейнерів.

знання архітектури та функцій оркестратора, а також його частин.

Інтеграційне тестування передбачає вивчення того, як взаємодіють різні компоненти програмної системи, щоб з'ясувати, чи працюють вони коректно відносно решти системи. Це відбувається не на рівні окремих функцій, а на рівні міжкомпонентної взаємодії. Отже, мета інтеграційного тестування - перевірити, як компоненти взаємодіють один з одним, а також виявити і виправити будь-які проблеми, пов'язані з взаємодією.

Інтеграційне тестування в контексті Кубер передбачає підтвердження того, як різні компоненти, такі як сервер API, контролери, планувальник, kubelet та інші, взаємодіють один з одним. Тестування інтеграції, наприклад, може перевірити функціональність планувальника при запуску та зупинці контейнерів, функціональність API-сервера при відповіді на запити від різних компонентів, а також сумісність різних компонентів під час обходу відмови та інших сценаріїв.

Оскільки воно дозволяє виявити проблеми взаємодії між компонентами до того, як вони будуть виявлені під час тестування на рівні компонентів, інтеграційне тестування є важливим компонентом тестування Кубер.

Процес тестування програмного забезпечення включає інтеграційне випробування, яке має вирішальне значення. Цей вид тестування передбачає спільну перевірку різних програмних компонентів для забезпечення належної роботи та дотримання як функціональних, так і нефункціональних критеріїв.

Інтеграційне тестування дозволяє виявити проблеми взаємодії програмних компонентів, такі як невідповідність API, проблеми залежностей

і конфігурації, помилки обміну даними тощо. Цей вид тестування також допомагає знизити ризики на стадії виробництва, коли труднощі з інтероперабельністю можуть спричинити значні проблеми та збої в роботі системи.

Інтеграційне тестування використовується для перевірки того, як різні компоненти Кубера, такі як контейнери, кластери та інші сервіси, взаємодіють один з одним. Для цього необхідно враховувати особливості архітектури та контейнеризації.

Для проведення ефективного інтеграційного тестування необхідно використовувати ряд методологій, включаючи тестування інтерфейсу користувача, тестування API та ін. За допомогою таких інструментів, як Sonobuoy, KubeTest та інших, ви можете перевірити, як різні компоненти Kubernetes взаємодіють один з одним.

Основні методи інтеграційного випробування і те, як вони використовуються в середовищі, будуть розглянуті в наступних параграфах.

Перевірка взаємодії між різними компонентами системи є ще одним важливим кроком в інтеграційному тестуванні. Наприклад, в залежності від того, як налаштовано Kubernetes, можна використовувати декілька типів мереж, але вони повинні функціонувати належним чином разом. Також важливо переконатися, що різні версії програмного забезпечення системи сумісні одна з одною і добре працюють разом.

Загалом, ретельне планування, глибокий аналіз системи та врахування численних елементів, які можуть вплинути на роботу системи, є необхідними для інтеграційного тестування.

1.4 Основні підходи до інтеграційного тестування

Розробка може використовувати різні стратегії тестування інтеграції:

- 1 Bottom-up підхід - полягає в тому, що тести створюються для окремих компонентів, які пізніше об'єднуються в єдину систему. Цей підхід

дозволяє розробникам перевірити роботу кожного компонента окремо, але не забезпечує повного покриття функціональності системи в цілому.

- 2 Top-down підхід - полягає в тому, що тести створюються для системи в цілому, а потім детально перевіряється робота кожного компонента. Цей підхід забезпечує повне покриття функціональності системи, але може бути складним для виконання через необхідність створення складних тестів для системи в цілому.
- 3 Hybrid підхід - поєднує в собі bottom-up та top-down підходи. Починаючи з top-down підходу, тести створюються для системи в цілому, а потім детально перевіряється робота кожного компонента. Цей підхід дозволяє забезпечити повне покриття функціональності системи, а також перевірити роботу окремих компонентів.

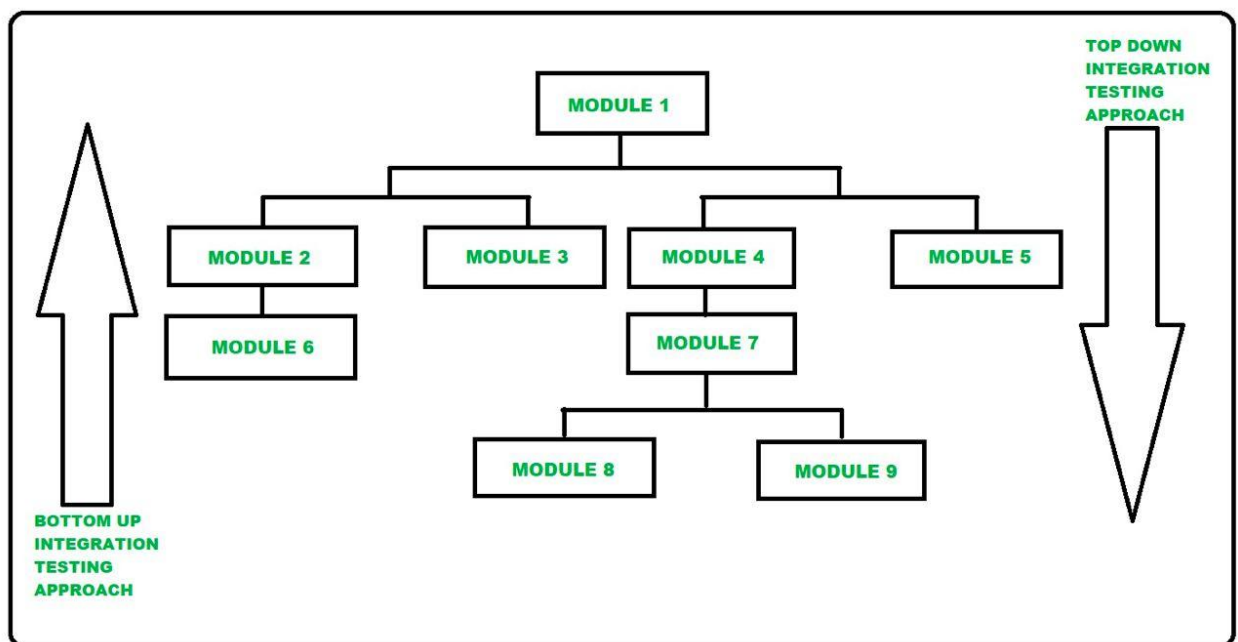


Рис. 1.2 Різниця між інтеграційним тестуванням зверху вниз та інтеграційним тестуванням знизу вгору

1.5 Особливості інтеграційного тестування оркестратора Kubernetes

Інтеграційне тестування повинно враховувати унікальні особливості оркестратора. Нижче наведено деякі з найважливіших характеристик:

- Поділ додатків на невеликі, незалежні сервіси. Мікросервіси можуть бути розгорнуті за допомогою Kubernetes і розміщені на різних вузлах. Це означає, що при тестуванні слід враховувати ймовірність взаємодій і залежностей між мікросервісами.
- Наявність кластерів. Коли один з вузлів кластера виходить з ладу, Kubernetes може мати проблеми з доступністю. При тестуванні необхідно враховувати використання кластерів і перевіряти доступність систем аварійного відновлення.
- Наявність контейнерів. Додатки та сервіси в Kubernetes розгортаються з використанням контейнерів, що може вплинути на тестування. Наприклад, може знадобитися запуск тестів у певних контейнерах із залежностями. Також слід враховувати можливість виконання контейнерів на різних вузлах.
- Доступні сервіси. За допомогою Кубера ви можете створювати сервіси, які дозволяють взаємодіяти з різними мікросервісами та контейнерами. Під час тестування необхідно перевірити доступність та функціональність сервісів.
- Масштабованість. Кубер може задовольнити потреби програм і сервісів. Внаслідок цього можуть виникати проблеми з доступом та складна взаємодія між компонентами системи.

Також, через складний дизайн цієї технології, інтеграційне тестування оркестратора Kubernetes має свої особливості. При тестуванні Кубера, слід враховувати наступні особливості:

- Кількість компонентів: Кубер має значну кількість компонентів, які взаємодіють між собою. Це означає, що інтеграційне тестування повинно охоплювати кожен з цих елементів і те, як вони працюють разом.
- Взаємозалежність компонентів: Різні компоненти Кубера можуть залежати один від одного. Як наслідок, якщо одна частина системи

неефективна, вся система може вийти з ладу. Необхідно враховувати всі потенційні залежності компонентів під час інтеграційного тестування.

- Масштабованість: Кубер може обробляти десятки тисяч вузлів і контейнерів. Як наслідок, інтеграційне тестування повинно бути масштабованим і здатним тестувати систему у великих кількостях.
- Швидкість: щоб уникнути уповільнення процесу розробки, інтеграційне тестування повинно бути виконано правильно і швидко.
- Відмовостійкість: Кубер має системи для виявлення помилок та повернення системи до роботи. Відмовостійкість і реакція системи на збої повинні бути протестовані як частина процесу інтеграції.

При підготовці до інтеграційного тестування оркестратора Kubernetes слід враховувати всі ці аспекти.

Важливою потребою інтеграційної перевірки оркестратора є те, що тести повинні враховувати широкий спектр потенційних мережевих конфігурацій і сценаріїв розгортання контейнерів. Це означає, що тести повинні бути стійкими і ефективними на будь-яких налаштуваннях для Кубера.

Тести повинні бути розроблені таким чином, щоб бути сумісними з будь-якою платформою, оскільки Кубер може бути встановлений на різних платформах, включаючи Amazon, Google Cloud або локально. Також при написанні тестів важливо враховувати будь-які проблеми, які можуть виникнути під час розгортання контейнерів на кластері Кубера, такі як збій мережі, недоступність ресурсів та інші.

Важливо перевірити сценарії балансування навантаження та аварійного відновлення. Загалом, інтеграційну перевірку слід проводити, оскільки Кубер є дуже складною системою, враховуючи всі її особливості та забезпечуючи перевірку на багатьох платформах та конфігураціях.

1.6 Архітектура Kubernetes та її вплив на інтеграційне тестування

Для керування контейнерними програмами у Кубер використовується низка основних абстракцій, зокрема

Вузли: Реальні або віртуальні комп'ютери, на яких працюють ваші контейнери, називаються вузлами. На вузлах виконується середовище виконання Кубера, яке керує контейнерами на вузлі.

Поді: У Кубері, контейнер є найменшою одиницею, що розгортається. Він являє собою один екземпляр процесу, який наразі працює у кластері. Один або декілька контейнерів, які мають спільний мережевий простір імен і можуть з'єднуватися один з одним через локальний хост, можуть бути знайдені в поді.

Сервіси: Група подів, що виконують один і той самий додаток, представлена сервісом, який є абстракцією. Навіть коли под створюється або видаляється, сервіси пропонують постійну IP-адресу та DNS-ім'я, які можуть бути використані для доступу до додатка. Ці абстракції, відомі як контролери реплікації та набори реплік, використовуються для того, щоб гарантувати, що в будь-який момент часу буде активна необхідна кількість реплік (екземплярів) одного з подів. За необхідності вони автоматично коригують кількість копій.

Розгортання: Розгортання - це більш просунута абстракція, яка контролює декілька наборів реплік та подів і пропонує такі функції, як оновлення та відкоти.

Карти конфігурацій: Карта конфігурації - це об'єкт Кубера, який дозволяє зберігати дані конфігурації у форматі ключ-значення. Потім ці дані можна використовувати у вигляді конфігураційних файлів або змінних оточення.

Секрети: Секрет ідентичний ConfigMap, за винятком того, що він призначений для зберігання приватної інформації, такої як ключі API, паролі та інші конфіденційні дані. За допомогою просторів імен можна розділити один кластер на декілька віртуальних кластерів.

Ресурси всередині кластера можна впорядкувати та ізолювати за допомогою просторів імен. Це деякі з ключових абстракцій, і багато інших об'єктів, таких як CronJobs, StatefulSets і Jobs, будуються на них, щоб запропонувати ще більш складні функціональні можливості. На інтеграційне

перевірка впливає архітектура Кубера через складну топологію системи та взаємодію з численними компонентами. Для успішної перевірки необхідно розуміти основні компоненти та їх функції.

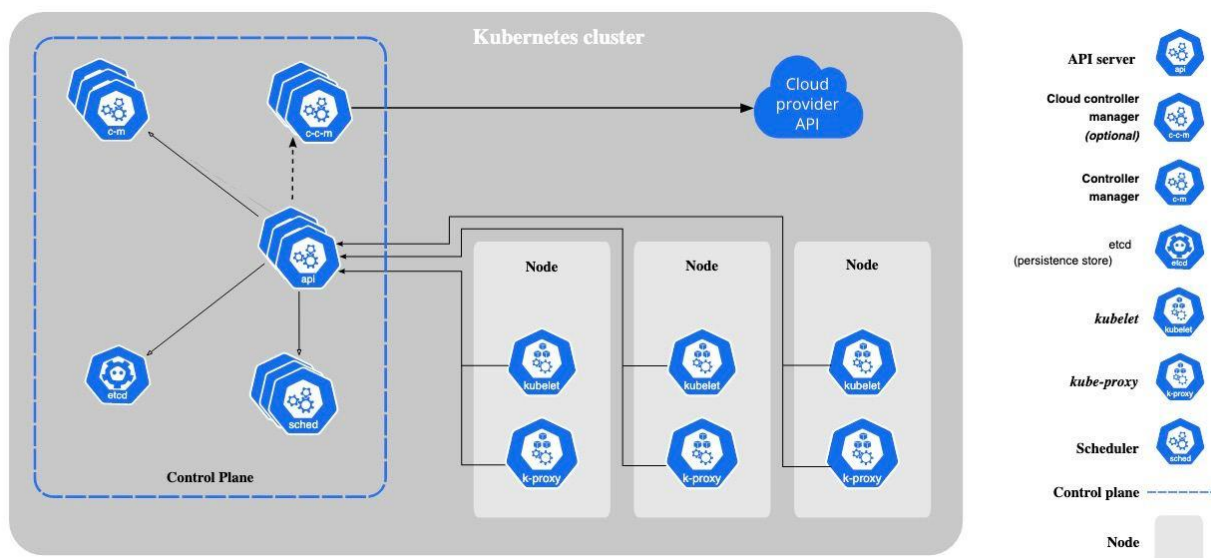


Рис. 1.3 Структура кластера кубернетеса

Головний вузол, який контролює всі інші вузли, є основною частиною Кубера. Контролер-менеджер, планувальник та сервер API - ось деякі з компонентів, що входять до складу головного вузла. Контролер-менеджер керує життєвим циклом ресурсів, планувальник розподіляє робочі навантаження між доступними вузлами, а API-сервер пропонує інтерфейс для управління кластером.

Серед додаткових компонентів Кубера є робочі вузли, які виконують власне обробку даних та виконання завдань. На кожному робочому вузлі присутній кубелет, який з'єднується з головним вузлом і запускає та зупиняє контейнери. Куб-проксі, який забезпечує зв'язок між контейнерами, також присутній на кожному робочому вузлі. Залежно від налаштувань, Кубер може включати інші компоненти, такі як etcd, який пропонує сховище для конфігурації кластера, або DNS-сервер, який дозволяє використовувати доменні імена для зв'язку між контейнерами. Взаємодія всіх цих компонентів

між собою та з іншими системами має бути врахована під час інтеграційної перевірки.

Перевірка різних налаштувань кластера та забезпечення точності документації також є важливими. Більше того, Кубер дуже автоматизований, тому перевірка також має бути автоматизованою. Ви можете досягти цього, використовуючи такі технології, як Ansible, Puppet, Chef або Terraform, які полегшують налаштування тестового середовища. Крім того, під час перевірки Кубера необхідно враховувати численні архітектурні рішення, включаючи мережеві політики, заходи безпеки, рівні доступу до ресурсів та функції різних компонентів Кубера. Щоб перевірити правила мережі, ви можете використовувати, наприклад, ситцеву або плетену сітку. Важливо випробувати можливості резервного копіювання та відновлення кластера. Ви можете досягти цього за допомогою різних інструментів, таких як Velero або Neptio Ark.

Крім того, Кубер володіє складними можливостями мережевого тестування, які вкрай важливі для запуску мікросервісних додатків. Їм можуть розділяти мережеві ресурси для кожної програми та зберігати їх стан за допомогою Kubernetes, який може автоматично створювати віртуальні мережі для кожної розгорнутої програми. Доступність додатків з різних мереж, обмеження пропускної здатності, обмеження швидкості з'єднання та інші правила мережі - все це можна налаштувати в Кубері.

Можливість створювати тестові середовища, які ідеально повторюють фактичне виробниче середовище, в якому працює додаток, є ще одним важливим аспектом дизайну Kubernetes. Це дозволяє виявити потенційні проблеми до розгортання та запуску програми у виробничому середовищі.

Як результат, Архітектура має значний вплив на інтеграційне тестування, оскільки вона пропонує передові можливості для тестування декількох компонентів додатків та можливість створювати тестові середовища, максимально наближені до реальних виробничих налаштувань.

1.7 Інструменти для інтеграційного тестування Kubernetes

Інтеграційне тестування може проводитися з використанням найрізноманітніших технологій. Найбільш типовими прикладами є:

- 1 Kube-test - фреймворк для тестування Kubernetes, який забезпечує автоматизоване тестування різних аспектів кластера. Він дозволяє створювати тести для перевірки працездатності кластера, включаючи різні сценарії відновлення після збоїв, тести підвищення масштабування та багато іншого.
- 2 Kubetest - інструмент для автоматизованого тестування Kubernetes. Він дозволяє запускати тести на різних платформах, включаючи локальні тестові середовища, контрольовані кластери та хмарні рішення.
- 3 Sonobuoy - інструмент для виконання конформаційних тестів Кубера. Він дозволяє перевірити відповідність кластера стандартам Kubernetes, включаючи аспекти безпеки, доступність та надійність.
- 4 Ginkgo - фреймворк для тестування, який забезпечує підтримку BDD (Behaviour Driven Development) тестування в Go. Він може використовуватись для написання тестів для Kubernetes.
- 5 Kubernetes e2e - набір тестів на рівні кінцевої точки, які дозволяють перевірити різні аспекти роботи Кубера, включаючи масштабування, безпеку та доступність.
- 6 Kubeadm - інструмент, що дозволяє автоматизувати процес встановлення Kubernetes. Він також може використовуватись для створення тестового кластера для проведення інтеграційного тестування.
- 7 Minikube - це інструмент для локального розгортання і тестування кластерів Kubernetes. Він дозволяє вам створювати одновузловий кластер Kubernetes на вашому локальному комп'ютері, що дозволяє вам відтворювати та експериментувати з роботою Kubernetes без необхідності розгортання повноцінного кластеру.

Ми повинні врахувати певні вимоги та характеристики, перш ніж вибрати один із цих інструментів, оскільки кожен з них має свої переваги та недоліки.

1.8 Висновки до розділу 1

Дослідження вимог до програмного забезпечення включало розгляд технологій контейнеризації та оркестратора, а також можливостей і методів інтеграційного тестування цих технологій. Було виявлено, що інтеграційне тестування Kubernetes оркестратора перевіряло багато елементів, таких як топологія мережі, зберігання, балансування навантаження, автоматичне масштабування та безпеку.

Досліджено можливості інтеграційного тестування оркестраторів Kubernetes через дизайн і складність їхніх компонентів. Крім того, були розглянуті інструменти, які мають на меті автоматизувати інтеграційне тестування Kubernetes. Багато методологій, включаючи підходи «зверху вниз» і «знизу вгору», були враховані під час проведення інтеграційних тестів.

Таким чином, очевидно, що інтеграційне тестування Kubernetes оркестратора є важливим для забезпечення безпеки, надійності та ефективності програм, що використовуються в середовищі Kubernetes. Оркестратор має складну архітектуру та функції, тому для успішного інтеграційного тестування необхідні відповідні інструменти та методології.

РОЗДІЛ 2

ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Мова програмування і програмне середовище

Для створення своєї проєктної роботи використовувалася мова Програмування Go — це швидка та ефективна мову розробки програмного забезпечення. Як відомо, синтаксис Go простий, що полегшує розуміння та підтримку коду. Крім того, підтримка конкурентного програмування вбудована в нього, що полегшує роботу з розподіленими та багатопотоковими системами.

Крім того, процес розробки прискорюється завдяки використанню інтегрованого середовища розробки GoLand. У GoLand є інструменти для завершення коду, налагодження, профілювання та інші корисні функції, які підтримують мову Go. Я зміг швидко та ефективно писати, тестувати та налагоджувати свій код завдяки цьому інтегрованому середовищу розробки.

Для тестування рекомендується використовувати мову програмування Go (також відому як Golang) та інтегроване середовище розробки GoLand, оскільки:

1. Продуктивність: Мова Go була створена спеціально для роботи з розподіленими та багатопотоковими системами. Це ідеальний варіант для тестування контейнерного оркеструвальника, такого як Kubernetes, завдяки його високій продуктивності та низькій затримці.
2. Go має зрозумілий синтаксис, який спрощує написання та розуміння коду. Це має вирішальне значення для тестування, оскільки тестовий код має бути простим для читання та оновлення.
3. Go містить вбудовану підтримку gorutin та каналів, що спрощує виконання одночасних процесів та комунікацію між ними. Це підтримує конкурентне програмування. Це робить тестування у ворожому середовищі простим і дозволяє легко імітувати розподілені середовища.

4. Багата бібліотека стандартів: Мова програмування Go пропонує безліч різних стандартних бібліотек, які полегшують створення тестових кейсів. Наприклад, при тестуванні Kubernetes може бути корисною підтримка HTTP-запитів, роботи з файлами, роботи з мережевими протоколами та багато іншого.
5. Інструментарій: Популярне інтегроване середовище розробки (IDE) для мови програмування Go називається GoLand. Воно пропонує комплексну допомогу для створення, аналізу та запуску тестів. GoLand включає в себе кілька інструментів, які полегшують тестування, в тому числі функції для автоматичного завершення коду, налагодження та профілювання.
6. Кросплатформеність: Windows, macOS і Linux - це лише деякі з операційних систем, на яких підтримуються Go і GoLand. Це дозволяє проводити крос-платформне тестування і розробку, забезпечуючи широку сумісність.

2.2 Docker як інструментарій для управління контейнерами

Docker є потужним інструментом, який створює просте та ефективне середовище для розгортання, управління та ізоляції програмного забезпечення в контейнерах. Він незамінний для управління контейнерами завдяки своїм багатьом перевагам.

Одним із головних переваг Docker є здатність упаковувати програмне забезпечення та всі його залежності в контейнери. Програми можуть працювати в окремому середовищі, не взаємодіючи з іншими програмами або системними ресурсами завдяки цьому контейнеру. Це дозволяє легко запускати програми на різних серверах або в хмарних середовищах, не налаштовуючи залежностей ручним чином.

Однією з основних переваг Docker є портативність. Можна запускати контейнери Docker на будь-якому сервері або хмарному середовищі, що підтримує Docker. Це надає розробникам гнучкість і можливість легко

переміщати свої програми між різними середовищами, не змінюючи саме додаток.

Здатність масштабувати управління контейнерами є великою перевагою Docker. Для створення контейнерних кластерів можна використовувати звичайні інструменти оркестрації, такі як Docker Kubernetes, які гарантують високу надійність і автоматичний розподіл навантаження між вузлами. Це дозволяє розробникам легко адаптувати свої програми до потреб і гарантує безперебійну роботу програм навіть при збільшенні обсягу оброблюваної інформації.

Докер пропонує як інтерфейс командного рядка, так і графічний інтерфейс управління контейнерами. За допомогою простих команд або веб-інтерфейсу розробники можуть створювати, запускати, зупиняти та відображати контейнери. Програмне забезпечення Docker також дозволяє керувати контейнерами та керувати їх використанням.

Головні переваги Docker:

- Легкість розгортання: Docker дозволяє розгортати програми швидко та просто. У ньому містяться всі необхідні залежності та конфігурації, що полегшує перенесення програм у різні середовища.
- Ізоляція: висока ізоляція контейнерів Docker дозволяє запускати багато програм і послуг в ізольованих місцях. Це запобігає впливу проблем у одному контейнері на роботу інших.
- Масштабованість: Docker полегшує масштабування програм, дозволяючи розподіляти навантаження між різними контейнерами. Можна створити кластер контейнерів з високою доступністю та автоматичним розподілом навантаження за допомогою таких інструментів оркестрації, як Kubernetes або Docker Swarm.
- Швидкість і ефективність: завдяки використанню контейнерних образів Docker створювати, запускати та зупиняти контейнери можна швидко. Вони потребують меншої кількості апаратних ресурсів, ніж віртуальні машини.

- Стандартизація та спільне використання: контейнери Docker легко транспортувати та використовувати. Розробники можуть легко переносити та ділитися своїми програмами в різні середовища, а також використовувати контейнери, створені іншими розробниками, що сприяє спільному використанню та швидкому розгортанню програм.

Через ці переваги Docker є потужним інструментом для управління контейнерами в програмній розробці. Можливості Docker для розширення, ізоляції та масштабування прискорюють процеси розгортання та забезпечують підтримку стійкої роботи програмного забезпечення.

2.2 Висновки до розділу 2

Використання мови програмування Go та інтегрованого середовища розробки GoLand для проєкту, заснованого на інтеграційному тестуванні оркестратора контейнерів, в даному випадку Kubernetes, видалося логічним і продуктивним. Мова програмування Go відома своєю швидкістю, ефективністю та простим синтаксисом, що дозволяє ефективно реалізовувати концепції та створювати надійні програмні системи.

Для написання, тестування та налагодження програм на Go інтегроване середовище розробки GoLand пропонує потужні інструменти. Налаштування, профілювання, автоматичне завершення коду та багато інших функцій роблять програмування простішим та ефективнішим. Це середовище розробки спростило створення програмного забезпечення та гарантувало якість готового продукту.

Загалом, проєкт тестування інтеграції оркестратора контейнерів був ефективно реалізований завдяки використанню мови програмування Go та інтегрованого середовища розробки GoLand. Ці інструменти забезпечили ефективну розробку та надійну роботу програмного забезпечення. На успіх розробки проєкту значною мірою вплинув вибір таких потужних і корисних інструментів.

Docker є потужним інструментарієм для управління контейнерами, який надає розробникам та адміністраторам систем зручний та ефективний спосіб розгортання, управління та масштабування програмного забезпечення. Використання Docker допомагає прискорити процес розробки, полегшити розгортання додатків та забезпечити більшу гнучкість та ефективність управління інфраструктурою.

РОЗДІЛ 3

ПРОЄКТУВАННЯ ТА КОНСТРУЮВАННЯ ТЕСТІВ

3.1 Розробка тестів для інтеграційного тестування Kubernetes

Інтеграція з Kubernetes була протестована за допомогою тестів, перелічених нижче:

- Тест на створення та видалення Pod
- Тест на створення та видалення сервісу
- Тест для масштабування Pod
- Тест на оновлення Pod
- Тест на автоматичне аварійне відновлення
- Тест на балансування навантаження

Цей процес розробки тестів інтеграційного тестування Kubernetes дозволив тестувати програму в різних умовах, визначити потенційні проблеми та переконатися, що він відповідає кластерним вимогам і вимогам безпеки, а також підвищити стабільність і надійність роботи програми на платформі Kubernetes.

3.2 Процес проведення інтеграційного тестування Kubernetes

У процедуру інтеграційного тестування Kubernetes можуть бути включені наступні етапи:

1. Створення тестового середовища: Запустіть тест на тестовому кластері Kubernetes або в існуючому тестовому середовищі.
2. Визначте тестові сценарії: Створіть тестові кейси для різноманітних завдань з ресурсами Kubernetes, включаючи управління ресурсами, мережею, масштабування, оновлення, видалення контейнерів та розгортання. Вимоги проєкту та різноманітні варіанти використання повинні бути включені в тестові сценарії.

3. Автоматизація тестування - це створення автоматизованих тестів, які виконуються без участі людини. Для цього можна використовувати Kubernetes Test Grid, наскрізні тести Kubernetes E2E (End-to-End), а також ваші власні внутрішні інструменти.

4. Виконання автоматизованих тестів у тестовому середовищі для запуску тестових сценаріїв називається тестуванням. Щоб підтвердити продуктивність і стабільність системи за різних обставин, тестові сценарії слід запускати в різних конфігураціях.

5. Оцінка результатів тестування, включаючи реакцію системи на різні тестові сценарії, аналіз результатів і виявлення проблем, недоліків і помилок. Результати тестування можуть бути переглянуті за допомогою відповідного програмного забезпечення для аналізу даних.

6. Документація: Написання результатів тестування, аналізів, висновків та рекомендацій. е може бути використано для створення звіту про тестування, який може бути використаний для додаткового аналізу, розробки та вдосконалення системи.

7. Виправлення помилок: Якщо під час тестування виявлено будь-які недоліки або помилки, їх слід виправити і проаналізувати результати модифікацій.

8. Повторне тестування: Після усунення дефектів і внесення змін до системи можна провести повторне тестування, щоб перевірити успішність модифікацій і стабільність системи.

9. Аналіз продуктивного використання: Щоб підтвердити фактичну продуктивність і стабільність системи під час реального використання, наступний етап тестування може бути проведений на виробничому кластері, де Kubernetes вже використовується.

10. Звітність та вдосконалення системи: Коли тестування завершено і результати проаналізовано, можна створити звіт про тестування, який містить висновки, коментарі та рекомендації щодо подальшого вдосконалення

системи. Це дослідження показує, що система може бути змінена і вдосконалена.

11. Процес тестування інтеграції Kubernetes може здійснюватися циклічно, коли тестування проводиться знову, щоб підтвердити ефективність і стабільність змін і поліпшень, внесених в систему. Цей цикл може тривати доти, доки не будуть досягнуті заздалегідь визначені стандарти якості та надійності системи.

Це повне пояснення процесу інтеграційного тестування. Виконання кожного кроку буде залежати від цілей проєкту, використовуваних інструментів, розміру та складності програми та багатьох інших змінних. Щоб гарантувати якість, надійність і безпеку програми в середовищі Kubernetes, важливо враховувати найкращі практики та стандарти.

3.3 Аналіз результатів інтеграційного тестування Kubernetes

Щоб оцінити продуктивність, швидкодію, безпеку і надійність програми в середовищі Кубер, аналіз результатів інтеграційної перевірки Кубера включає в себе ретельну оцінку інформації, зібраної в ході випробування. Нижче наведені основні процедури, які можуть бути використані при аналізі результатів випробувань:

1. Оцінка стабільності та продуктивності: ви можете перевірити стабільність та продуктивність програми в середовищі Кубер, переглянувши результати випробування. В ході цього процесу можуть бути оцінені такі параметри продуктивності, як час відгуку, використання ресурсів (процесор, пам'ять), навантаження на мережу і так далі. Виявлення потенційних збоїв та проблем зі стабільністю програми в середовищі Кубер може бути частиною процесу оцінки стабільності.

2. Аналіз безпеки: у середовищі Кубер підтримка безпеки програми має важливе значення. Відповідність програми прийнятим стандартам безпеки, можливість вразливостей, точність налаштувань безпеки кластера,

відповідність засобам контролю доступу та інші фактори безпеки - все це може бути перевірено в рамках аналізу результатів випробування.

3. Оцінка масштабованості: Кубер забезпечує як горизонтальне, так і вертикальне масштабування програми. Оцінка масштабованості програми, виявлення потенційних обмежень масштабованості та розробка пропозицій щодо поліпшення масштабування програми в середовищі Кубер - все це може бути включено до аналізу результатів випробування.

4. Виявлення проблем та формулювання рішень: у контексті Кубер перегляд результатів випробування може допомогти виявити потенційні проблеми, збої або слабкі місця в додатку. Це може включати виявлення завантажувальних ліній додатків, неправильних налаштувань, недостатнього охоплення моніторингом, помилок у налаштуванні ресурсів Kubernetes та інших елементів, які можуть вплинути на продуктивність і надійність програми. Ви можете надати пропозиції щодо усунення несправностей та покращення продуктивності програми в середовищі Кубер на основі результатів випробування.

5. Перевірка функціональності: у середовищі Кубера аналіз результатів випробування дозволяє перевірити відповідність функціональним вимогам програми. Це може включати виконання тестових сценаріїв, порівняння результатів з очікуваними вимогами та перевірку доступності та точності реалізації функцій, серед інших функціональних можливостей.

6. Облік бізнес-вимог: при аналізі результатів випробування слід враховувати бізнес-вимоги до додатка. Це може включати вимоги до відмовостійкості програми, доступності, продуктивності та інших важливих функцій для бізнес-операцій, які вона підтримує.

7. Створення плану вдосконалення: у середовищі Кубер ви можете створити план вдосконалення програми на основі аналізу результатів випробування. Для поліпшення продуктивності програми в середовищі Kubernetes наведені рекомендації щодо усунення виявлених проблем, оптимізації налаштувань, впровадження кращих практик на практиці та роботі

з Кубер, поліпшення моніторингу та ведення журналу, створення автоматизованих тестів і скриптів для пошуку і усунення проблем, впровадження резервного копіювання і відновлення додатків та інші дії можуть бути включені.

8. Перевірка поліпшень: після реалізації плану поліпшення ви повинні відстежувати результати і ефективність внесених коригувань. Це може спричинити відстеження продуктивності програми, вивчення журналів та показників, повторну перевірку для пошуку вдосконалень та виявлення нових проблем та, якщо потрібно, зміну стратегії вдосконалення.

9. Звітність: аналіз результатів перевірки і поліпшення середовища Кубер повинні бути зафіксовані і представлені в звіті. Це може містити пояснення виявлених проблем, пропозиції щодо їх вирішення, результати впровадження плану вдосконалення в дію, спосіб відстеження прогресу та іншу відповідну інформацію, яка може бути корисною розробникам, інженерам DevOps та іншим зацікавленим сторонам.

10. Впровадження змін на практиці: ви можете вносити зміни до програми в середовищі Кубер на основі аналізу результатів перевірки та створеного плану вдосконалення. Щоб внести зміни до програми в середовищі Кубер, це може спричинити зміну файлів конфігурації, налаштування ресурсів Кубера, розгортання нової версії програми, покращення моніторингу та ведення журналу, виконання автоматизованих перевірок та інші дії.

11. Виконання резервного копіювання та відновлення: після внесення змін до програми слід також перевірити процеси резервного копіювання та відновлення, щоб переконатися, що вони працюють належним чином у середовищі. Створення резервних копій програми та її даних, визначення можливості відновлення програми з резервної копії, перевірка методів відновлення за різних обставин та розробка автоматизованих сценаріїв для резервного копіювання та відновлення програми - це кілька прикладів того, що це може включати.

12. Постійна оптимізація: важливо продовжувати відстежувати, оцінювати та підвищувати продуктивність програми в середовищі Кубер після внесення змін та вдосконалення. Це може спричинити налаштування моніторингу, виявлення та усунення проблем, оптимізацію ресурсів та використання таких функцій, як автоматичне масштабування, балансування навантаження та інші, щоб забезпечити ефективну та надійну роботу програми в середовищі.

13. Постійне навчання та розвиток: оскільки Кубер - це технологія, що швидко розвивається, важливо бути в курсі всіх аспектів використання цього інструменту. Щоб вдосконалити свої навички та знання, вам слід стежити за випусками Кубера, вивчати нові методи та найкращі практики використання системи, а також брати участь у спільнотах, форумах, конференціях та інших заходах, пов'язаних з ним.

14. Постійне вдосконалення процесу: важливо продовжувати вдосконалювати процедури налаштування та запуску середовища Кубер. Це може спричинити оцінку процесу впровадження змін на ефективність, пошук областей для вдосконалення та адаптацію до змінних потреб бізнесу та внутрішніх процедур у фірмі. Щоб підвищити ефективність та швидкість розгортання додатків, ви можете використовувати автоматизацію та організацію процесів.

15. Підтримка безпеки: дуже важливо підтримувати безпеку середовища Кубер. Поточні засоби контролю безпеки, такі як налаштування мережі, управління ролями та дозволами, налаштування TLS, виявлення вразливостей та виправлення помилок, моніторинг та реагування на події безпеки та навчання команди безпеки, повинні регулярно перевірятися. В системі Кубер підтримка безпеки - це безперервний процес, що вимагає постійних оновлень і вдосконалень.

3.4 Порівняння результатів із результатами інших досліджень

Оцінити відмінність, схожість або розбіжність результатів можна шляхом порівняння результатів одного дослідження з результатами інших досліджень. Це може допомогти визначити потенційні межі дослідження, з'ясувати, чи підтверджуються результати іншими дослідженнями, або виявити нові елементи, які можуть бути важливими для майбутніх досліджень.

Для порівняння отриманих результатів з результатами інших досліджень використовуйте наступні методи:

1. Огляд літератури: аналіз опублікованих результатів досліджень про Kubernetes або споріднені технології в науковій літературі. Порівняння ваших результатів з результатами інших досліджень може допомогти вам виявити обмеження або нові ракурси дослідження, а також паралелі або варіації між дослідженнями.

2. Аналіз методології: порівняння підходу, застосованого в цьому дослідженні, з підходом, застосованим в інших дослідженнях. Висновки щодо відмінностей у отриманих результатах можна зробити, проаналізувавши відмінності в методології дослідження.

3. Результати та висновки: Оцінка отриманих результатів і порівняння з результатами інших досліджень. Порівняння або протиставлення висновків може допомогти в оцінці впливу результатів на наукову спільноту та подальші пов'язані з ними дослідження.

4. Обмеження дослідження: Порівняння обмежень, зазначених у цьому дослідженні, з обмеженнями, зазначеними в інших дослідженнях. Це може допомогти виявити будь-які збіги або розбіжності в обмеженнях, які можуть вплинути на достовірність і застосовність результатів дослідження.

5. Порівняння результатів дослідження в різних умовах, беручи до уваги такі змінні, як географічне розташування, розмір організації, використання ресурсів та інші елементи, які можуть вплинути на результати.

Розуміння того, наскільки узагальнюючими та релевантними є результати в різних сценаріях, може допомогти врахування різних обставин.

6. Критичний аналіз: Оцінка результатів дослідження під критичним кутом зору, враховуючи будь-які упередження, помилки та інші змінні, які можуть вплинути на результати. Це може полегшити об'єктивну оцінку результатів порівнянь.

Загалом, порівняння результатів дослідження з результатами інших досліджень може допомогти зрозуміти контекст і релевантність результатів, виявити зв'язки або відмінності між дослідженнями та сприяти розвитку науки.

3.5 Основні результати дослідження

Нижче наведено основні результати проведеного дослідження інтеграційного тестування Kubernetes:

1. Успішна інтеграція: Для побудови функціонального кластеру, дослідження успішно інтегрувало різноманітні компоненти Kubernetes, включаючи контейнерні двигуни, управління ресурсами, мережеві рішення та інші. Було перевірено здатність цих компонентів взаємодіяти один з одним та запускати розподілені програми в контейнерах на декількох вузлах кластера.

2. Виявлення та вирішення проблем: В процесі тестування було виявлено багато проблем, включаючи помилки конфігурації, мережеві проблеми, обмеження ресурсів та інші. Ці труднощі були проаналізовані, а також створені відповідні плани та рішення для їх усунення та гарантування стабільного функціонування кластера.

3. Запобігання помилкам: Були створені кроки та процеси для виявлення та виправлення помилок налаштування, резервного копіювання даних, стеження за ресурсами та вирішення інших завдань, пов'язаних з кластером. Це дозволило підвищити стабільність роботи кластера та зменшити кількість помилок.

4. **Продуктивність та масштабованість:** Було проведено дослідження продуктивності та масштабованості кластера, яке включало оцінку ефективності контейнерних додатків, обмежень ресурсів, оптимізації мережі та інших факторів. Результати показують, що Кубер має хорошу продуктивність і масштабованість при правильній конфігурації та оптимізації.

5. **Безпека:** Дослідження включає оцінку безпеки кластера, виявлення будь-яких слабких місць та створення відповідних контрзаходів безпеки. Були вжиті запобіжні заходи для захисту мережевої безпеки, доступу до ресурсів кластера та безпеки контейнерних додатків.

6. **Просте розгортання та управління кластером:** Було оцінено всі аспекти розгортання та управління кластером, включаючи конфігурацію, моніторинг, масштабування, модернізацію та відновлення. Для спрощення розгортання та обслуговування кластерів були створені кращі практики та пропозиції.

7. **Порівняння результатів дослідження з результатами попередніх досліджень та застосуванням Kubernetes у реальних проєктах.** В результаті стало можливим порівняти ефективність, продуктивність та безпеку кластера з альтернативними системами, а також визначити характеристики та переваги використання Kubernetes в конкретних умовах дослідження.

Отже, ефективна інтеграція Kubernetes, виявлення та вирішення проблем, запобігання помилкам, продуктивність, масштабованість, безпека, простота розгортання та обслуговування, а також порівняння з іншими дослідженнями є основними висновками дослідження.

3.6 Висновки до розділу 3

Дослідження показало, наскільки корисним є Kubernetes як платформа для оркестрування контейнерів, яка надає потужні інструменти для створення та адміністрування розподілених додатків. Щоб подолати проблеми з інтеграцією та роботою Kubernetes, можна налаштувати налаштування, покращити використання ресурсів і застосувати правильні практики безпеки.

Результати дослідження мають реальний вплив і можуть бути використані в поточних проєктах і дослідженнях, які стосуються використання Kubernetes в різних додатках. Таким чином, компанії та розробники зможуть краще використовувати потужні можливості платформи та створити надійну, масштабовану основу для своїх додатків.

Загалом, дослідження інтеграційного тестування оркестратора контейнерів на основі Kubernetes показує його практичну значущість і цінність. Це значний крок до покращення розуміння Kubernetes, що допоможе його майбутньому розвитку та використанню в сучасній індустрії розробки програмного забезпечення.

ВИСНОВКИ

Результати дослідження підтверджують ідею про те, що оркестратор контейнерів і технології контейнеризації Kubernetes є важливими інструментами для створення, підтримки та тестування розподілених додатків. Інтеграційне тестування оркестратора контейнерів, зокрема Kubernetes, має вирішальне значення для забезпечення безпеки, надійності та ефективності програмного забезпечення.

Проект інтеграційного тестування оркестру контейнерів, який використовує мову програмування Go та інтегроване середовище розробки GoLand, має бути простим і ефективним. Щоб створювати надійні програмні системи, мова програмування Go пропонує швидкість, ефективність і простоту синтаксису. Унікальні інструменти, надані інтегрованим середовищем розробки GoLand, дозволяють створювати, тестувати та налагоджувати додатки Go.

Таким чином, результати дослідження підкреслюють перевагу технології Kubernetes, перевагу тестування інтеграції оркестратора контейнерів і переваги реалізації проєктів з використанням мови програмування Go та інтегрованого середовища розробки GoLand. Ці висновки повинні бути враховані розробниками, компаніями та іншими зацікавленими сторонами, які планують перейти на Kubernetes і провести інтеграційне тестування своїх розподілених додатків.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

- 1 Kubernetes. [Електронний ресурс]. URL: <https://kubernetes.io/>.
- 2 Docker. [Електронний ресурс]. URL: <https://www.docker.com/>.
- 3 Salimchev, A. Automated Testing of Kubernetes Applications. [Електронний ресурс]. URL: <https://www.altoros.com/blog/automated-testing-of-kubernetes-applications/>.
- 4 Sharma, P. Integration testing with Kubernetes [Електронний ресурс]. URL: <https://www.magalix.com/blog/integration-testing-with-kubernetes>.
- 5 KubeCon + CloudNativeCon Europe 2021. [Електронний ресурс]. URL: <https://events.linuxfoundation.org/kubecon-cloudnativecon-europe/>.
- 6 Kops. [Електронний ресурс]. URL: <https://github.com/kubernetes/kops>.
- 7 Acceptance Testing. [Електронний ресурс]. URL: <https://origin.geeksforgeeks.org/acceptance-testing-software-testing/>.
- 8 Helm. [Електронний ресурс]. URL: <https://helm.sh/>.
- 9 Gubernikoff, D. Testing Kubernetes: A Complete Guide. [Електронний ресурс]. URL: <https://container-solutions.com/testing-kubernetes-a-complete-guide/>.
- 10 Ng, B. Testing Kubernetes Admission Controllers. [Електронний ресурс]. URL: <https://itnext.io/testing-kubernetes-admission-controllers-9d94f507a840>.
- 11 Jenkins X. [Електронний ресурс]. URL: <https://jenkins-x.io/>.
- 12 Kubernetes Documentation. (2021). Kubernetes. URL: <https://kubernetes.io/docs/home/>.
- 13 Burns, B., & Vohra, S. (2016). Kubernetes: Up and Running: Dive into the Future of Infrastructure. O'Reilly Media, Inc.
- 14 Bhatia, M. (2020). Kubernetes Application Development: Build, deploy, and manage scalable microservices with Kubernetes. Packt Publishing Ltd.
- 15 Bormann, M. (2020). Integration testing with Kubernetes. URL: <https://medium.com/@martin.bormann/integration-testing-with-kubernetes-dbd753c26068>.

- 16 Klenk, M. (2018). Kubernetes Cookbook: Building Cloud-Native Applications. Packt Publishing Ltd.
- 17 Naik, V. (2020). Kubernetes Best Practices: Blueprints for Building Successful Applications on Kubernetes. Apress.
- 18 Hightower, K., Burns, B., & Beda, J. (2017). Kubernetes: Up and Running: Dive into the Future of Infrastructure. O'Reilly Media, Inc.
- 19 Luksa, M. (2019). Kubernetes in Action. Manning Publications.
- 20 Singh, J. (2020). Hands-On Kubernetes on Azure: Build, deploy, and manage scalable container applications with Kubernetes on Azure. Packt Publishing Ltd.
- 21 The Kubernetes Authors. (2021). Kubernetes Documentation: Testing. URL: <https://kubernetes.io/docs/concepts/testing/>.
- 22 Microservices & Kubernetes. [Электронный ресурс]. URL: <https://slides.com/volkansengul/microservices-kubernetes/fullscreen>.
- 23 Integration testing [Электронный ресурс] URL: <https://coderlessons.com/tutorials/kachestvo-programmnogo-obespecheniia/ruchnoe-testirovanie/integratsionnoe-testirovanie-2>.

Полтава 2023

ЗМІСТ

1. ОБ'ЄКТ ВИПРОБУВАНЬ	50
2. МЕТА ТЕСТУВАННЯ	50
3. МЕТОДИ ТЕСТУВАННЯ	50

1. ОБ'ЄКТ ВИПРОБУВАНЬ

Розроблені тести повинні надійно працювати на macos та ubuntu.

Для вимірювання якості розробленого програмного продукту виконано тестування на операційних системах.

Програмний продукт повинен:

1. Надійно працювати на macos та ubuntu.

2. МЕТА ТЕСТУВАННЯ

Метою тестування є процес технічного дослідження про якість продукту щодо контексту, в якому він повинен використовуватися. До цього процесу входить виконання програми з метою виявлення помилок.

3. МЕТОДИ ТЕСТУВАННЯ

Для тестування якості написаних інтеграційних тестів, була використана програма-Лінтер golangci-lint.

golangci-lint (Go Linter) - популярний і потужний Лінтер для Go code. Він призначений для аналізу вихідного коду Go і виявлення різних проблем, порушень і потенційних багів. Це допомагає гарантувати, що Go code дотримується найкращих практик, дотримується стандартів кодування та підтримує високу якість коду.

Деякі з критеріїв, які перевіряє golangci-lint, включають:

1. Проблеми з форматуванням: програма перевіряє наявність помилок форматування коду та невідповідностей, таких як неправильний відступ, непотрібні пробіли та порушення довжини рядка.
2. Стиль коду та Конвенції: це забезпечує дотримання загальноприйнятих конвенцій про кодування Go та найкращих практик. Це включає рекомендації щодо конвенцій про іменування, затінення змінних та організації пакетів.

3. Обробка помилок: він виявляє потенційні проблеми з обробкою помилок, такі як ігнорування помилок, неповернення або записування помилок, а також неправильне використання змінних помилок.

4. Складність коду: він вимірює складність функцій Go та визначає складний код, який може бути важко зрозуміти або супроводжувати. Це допомагає визначити області, де потрібен рефакторинг або спрощення коду.

5. Невикористаний код: він ідентифікує невикористані змінні, функції та імпортні дані, допомагаючи видалити непотрібний код і зменшити його захащеність.

6. Уразливості безпеки: він вказує на потенційні проблеми безпеки, такі як використання небезпечних криптографічних алгоритмів або небезпечних методів, які можуть призвести до вразливостей.

Щоб встановити та запустити клієнт `golang`, виконайте такі дії

1. Встановіть клієнт `golang`, виконавши таку команду:

`go install github.com/golangci/golangci-lint/cmd/`

2. Переконайтеся, що клієнт `golang` успішно встановлений, запустивши:

`golangci-lint –version`

3. Перейдіть до кореневого каталогу вашого проєкту Go.

4. Запустіть `golangci-lint` у своєму проєкті, виконавши таку команду:

`golangci-lint run`

5. Це дозволить проаналізувати код Go вашого проєкту та відобразити всі виявлені проблеми чи порушення.

Ви також можете налаштувати поведінку `golangci-lint`, налаштувавши його за допомогою файлу `.golangci.yml` у кореневому каталозі вашого проєкту. Цей файл дозволяє вмикати/ вимикати певні засоби компонування, налаштовувати параметри засобу компонування та вказувати каталоги / файли, які потрібно включити або виключити з аналізу.

Використання golangci-lint як частини робочого процесу розробки може допомогти виявити потенційні проблеми на ранній стадії та покращити загальну якість вашого коду Go.

Результат тестування виглядає наступним чином (Рис. А.1).

```

+ tests golangci-lint run --timeout 10m -v
INFO [config_reader] Config search paths: [./ /Users/alexnelepa/go/src/github.com/nagapw09/tests /Users/alexnelepa/go/src/github.com/nagapw09 /Users/alexnelepa/go/src/github.com /Users/alexnelepa/go/src /Users/alexnelepa/go /Users/alexnelepa /Users /]
INFO [lintersdb] Active 7 linters: [errcheck gosimple govet ineffassign staticcheck typecheck unused]
INFO [loader] Go packages loading at mode 575 (types_sizes|compiled_files|deps|exports_file|imports|name|files) took 176.47ms
INFO [runner/filename_unadjuster] Pre-built 0 adjustments in 1.568417ms
INFO [linters context/goanalysis] analyzers took 0s with no stages
INFO [runner] processing took 7.041µs with stages: max_same_issues: 2.25µs, nolint: 584ns, cgo: 500ns, max_from_linter: 500ns, skip_dirs: 500ns, identifier_marker: 250ns, uniq_by_line: 250ns, autogenerated_exclude: 250ns, sort_results: 209ns, diff: 208ns, exclude: 208ns, severity-rules: 208ns, path_prettifier: 167ns, source_code: 166ns, filename_unadjuster: 166ns, skip_files: 125ns, exclude-rules: 125ns, path_prefixer: 125ns, path_shortener: 125ns, max_per_file_from_linter: 125ns
INFO [runner] linters took 27.07875ms with stages: goanalysis_metadata_linter: 26.994917ms
INFO File cache stats: 0 entries of total size 0B
INFO Memory: 4 samples, avg is 33.4MB, max is 36.6MB
INFO Execution took 225.7205ms
+ tests

```

Рисунок А.1 Результат тестування

Тести перевірялися на таких операційних системах:

- macOS Ventura 13.3.1;
- ubuntu 22.04.2 LTS;
- ubuntu 23.04;
- ubuntu 22.10;

Тести відміно працювали. Критичних помилок не виявлено.

ДОДАТОК Б

Міністерство освіти і науки України	
Державний заклад “Луганський національний університет імені Тараса Шевченка”	
Факультет (інститут)	Навчально-науковий інститут фізики, математики та інформаційних технологій
	<hr/> (повна назва)
Кафедра	Інформаційних технологій та систем
	<hr/> (повна назва)

КЕРІВНИЦТВО КОРИСТУВАЧА

на виконання програмної розробки (ПР):

**“ІНТЕГРАЦІЙНЕ ТЕСТУВАННЯ ОРКЕСТРАТОРА
КОНТЕЙНЕРІВ НА ПРИКЛАДІ KUBERNETES”**

Полтава 2023

ЗМІСТ

1.1. Підготовка до роботи з додатком	55
1.2. Робота з додатком	56

1.1. Підготовка до роботи з додатком

Система mac os або ubuntu

Для тестування Kubernetes, ви можете використовувати як систему Mac OS, так і Ubuntu. Обидві системи підтримують Kubernetes і забезпечують можливість налаштування тестового середовища.

1. Mac OS: Якщо ви працюєте на комп'ютері Mac, ви можете встановити Kubernetes використовуючи Docker для Mac або встановити minikube. Docker для Mac містить вбудований Kubernetes, який дозволяє швидко налаштувати локальний кластер для тестування. Minikube також підтримує Mac OS і надає можливість запускати локальний однокластерний кластер Kubernetes.
2. Ubuntu: Ubuntu є популярною операційною системою для розробки і тестування Kubernetes. Ви можете встановити Kubernetes на Ubuntu, використовуючи інструменти, такі як kubectl, kubeadm та kubelet. Це дозволить вам налаштувати багатокomпонентний кластер Kubernetes на своєму локальному середовищі.

Встановлення Kubernetes на Mac OS або Ubuntu

Mac OS:

- Встановіть Docker Desktop: Це інтегроване середовище, яке містить Kubernetes. Відвідайте офіційний веб-сайт Docker і завантажте Docker Desktop для Mac. Встановіть його на свою систему, виконавши встановлювач.
- Увімкніть Kubernetes: Після встановлення Docker Desktop перейдіть до його налаштувань і активуйте підтримку Kubernetes. Це дозволить вам використовувати локальний кластер Kubernetes.

Ubuntu:

- Встановіть kubeadm, kubectl і kubelet: Відкрийте термінал на Ubuntu і виконайте наступні команди для встановлення kubeadm, kubectl і kubelet:

sudo apt-get update

sudo apt-get install -y kubelet kubeadm kubectl

sudo apt-mark hold kubelet kubeadm kubectl

- Ініціалізуйте кластер: Після встановлення інструментів Kubernetes виконайте команду для ініціалізації кластера:

sudo kubeadm init

Це створить новий кластер Kubernetes на вашому Ubuntu-сервері.

Ці кроки є загальними і можуть змінюватись залежно від версій Kubernetes та операційних систем. Рекомендується ознайомитись з офіційною документацією Kubernetes та документацією вашої конкретної операційної системи для отримання докладніших інструкцій та оновленої інформації.

Встановіть інструменти для тестування Kubernetes: Для запуску тестів Kubernetes вам знадобляться один з таких інструментів:

- kubectl: Це інтерфейс командного рядка для керування кластером Kubernetes. Він входить до складу Docker Desktop і можна використовувати безпосередньо з терміналу.
- Minikube: Якщо ви прагнете працювати з локальним кластером Kubernetes на Mac, ви можете встановити Minikube. Він дозволяє створювати та управляти локальними кластерами Kubernetes для тестування.
- kubetest: Kubetest є інструментом, розробленим спільнотою Kubernetes, який допомагає виконувати інтеграційні тести на кластері Kubernetes.

1.2. Робота з додатком

Для запуску теста потрібно виконати наступні кроки:

1. Для початку треба поставити галочку в конфігурації тесту щоб дати доступ sudo (рис. Б.1)

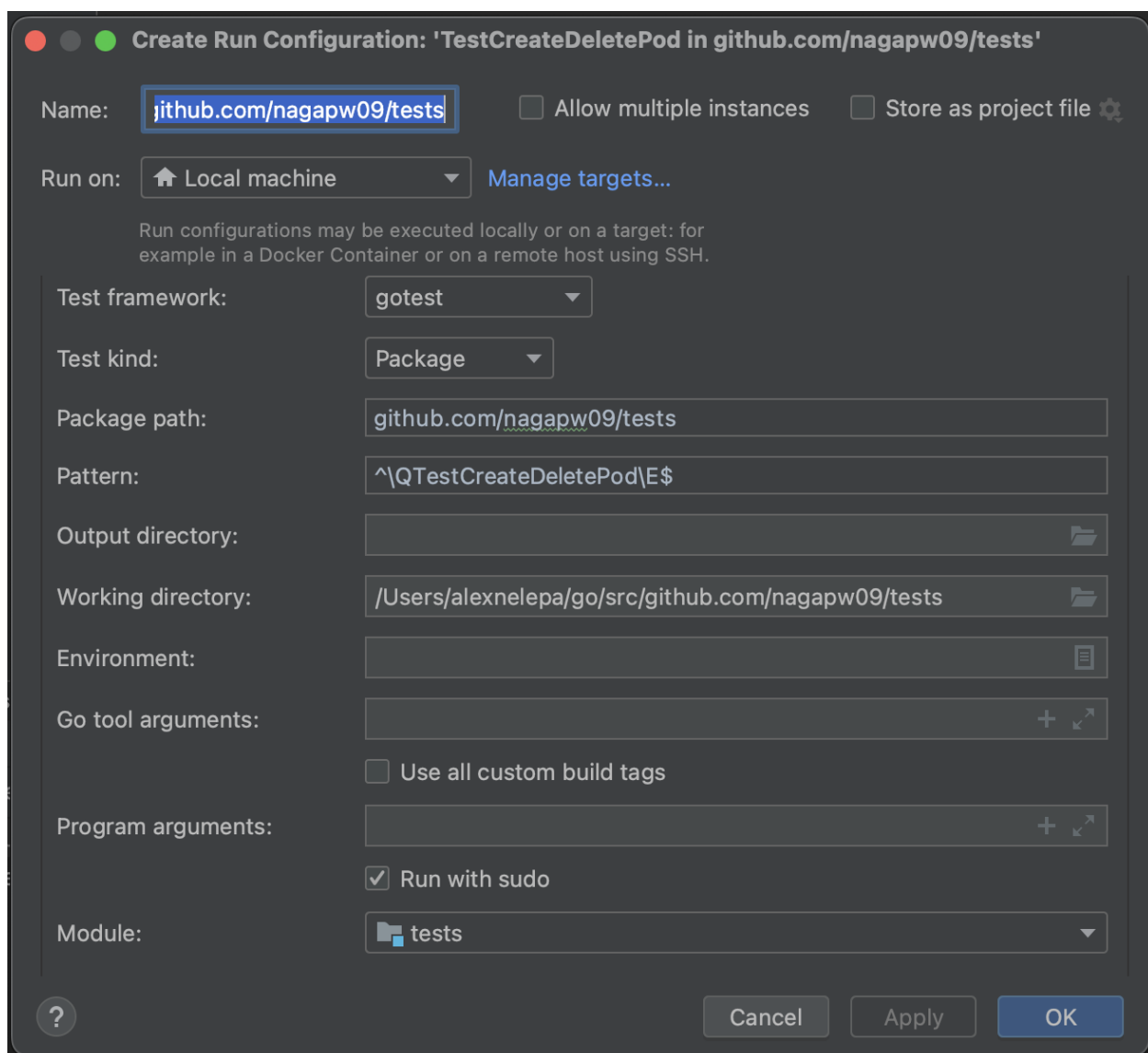


Рисунок Б.1 Конфігурація теста

2. Запустити тест (Рис. Б.2)

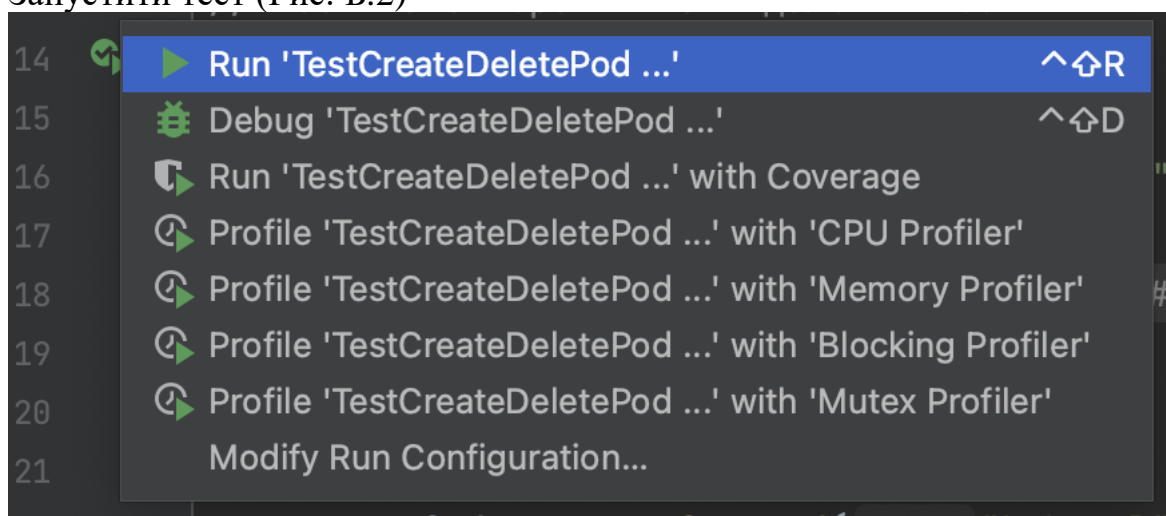


Рисунок Б.2 Меню теста

3. Чекати результатів (Рис. Б.3)

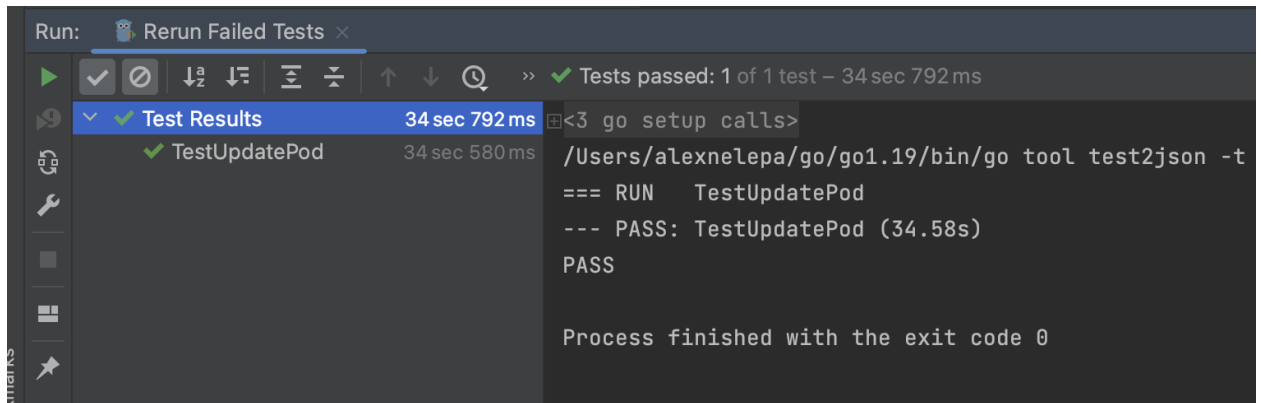


Рисунок Б.3 Результати виконання

для виконання цього тесту знадобилося 35 секунд і також він виконався правильно.

ДОДАТОК В

main_test.go

```
package work

import (
    "os"
    "testing"
)

func TestMain(m *testing.M) {
    os.Exit(m.Run())
}
```

automatic_recovery_test.go

```
package work

import (
    "fmt"
    "os/exec"
    "testing"
    "time"
)

// Тест на автоматичне аварійне відновлення
func TestAutomaticRecovery(t *testing.T) {
    // Старт Minikube
    startCmd := exec.Command("minikube", "start", "--vm-driver=docker", "--force")
    if err := startCmd.Run(); err != nil {
        t.Errorf("Failed to start Minikube: %v", err)
    }

    // Створення Pod
    createCmd := exec.Command("kubectl", "create", "-f", "pod.yaml")
    if err := createCmd.Run(); err != nil {
        t.Errorf("Failed to create Pod: %v", err)
    }

    if err := waitForPodRunning(); err != nil {
        t.Fatalf("Pod did not start successfully: %v", err)
    }

    deleteCmd := exec.Command("kubectl", "delete", "pod", "my-pod")
    if err := deleteCmd.Run(); err != nil {
        t.Fatalf("Failed to delete Pod: %v", err)
    }

    time.Sleep(10 * time.Second)

    if err := waitForPodRunning(); err != nil {
        t.Fatalf("Pod did not recover successfully: %v", err)
    }

    deleteCmd = exec.Command("kubectl", "delete", "pod", "my-pod")
    if err := deleteCmd.Run(); err != nil {
        t.Errorf("Failed to delete Pod: %v", err)
    }

    // Зупинка Minikube
}
```

```

stopCmd := exec.Command("minikube", "stop")
if err := stopCmd.Run(); err != nil {
    t.Errorf("Failed to stop Minikube: %v", err)
}
}

func waitForPodRunning() error {
    checkCmd := exec.Command("kubectl", "get", "pod", "my-pod", "-o",
"jsonpath={.status.phase}")
    for {
        output, err := checkCmd.Output()
        fmt.Println(string(output))
        if string(output) == "Running" {
            return nil
        }
        if err != nil {
            return err
        }
    }
}

```

create_belete_pod_test.go

```

package work

import (
    "os/exec"
    "testing"
)

// Тест на створення та видалення Pod
func TestCreateDeletePod(t *testing.T) {
    // Старт Minikube
    startCmd := exec.Command("minikube", "start", "--vm-driver=docker", "--
force")
    if err := startCmd.Run(); err != nil {
        t.Errorf("Failed to start Minikube: %v", err)
    }

    // Створення Pod
    createCmd := exec.Command("kubectl", "create", "-f", "pod.yaml")
    if err := createCmd.Run(); err != nil {
        t.Errorf("Failed to create Pod: %v", err)
    }

    // Перевірка наявності Pod
    describeCmd := exec.Command("kubectl", "describe", "pod", "my-pod")
    if err := describeCmd.Run(); err != nil {
        t.Errorf("Failed to describe Pod: %v", err)
    }

    // Видалення Pod
    deleteCmd := exec.Command("kubectl", "delete", "pod", "my-pod")
    if err := deleteCmd.Run(); err != nil {
        t.Errorf("Failed to delete Pod: %v", err)
    }

    // Зупинка Minikube
    stopCmd := exec.Command("minikube", "stop")
    if err := stopCmd.Run(); err != nil {
        t.Errorf("Failed to stop Minikube: %v", err)
    }
}

```

```
}
}
```

create_belete_service_test.go

```
package work

import (
    "os/exec"
    "testing"
)

// Тест на створення та видалення сервісу
func TestCreateDeleteService(t *testing.T) {
    // Старт Minikube
    startCmd := exec.Command("minikube", "start", "--vm-driver=docker", "--force")
    if err := startCmd.Run(); err != nil {
        t.Errorf("Failed to start Minikube: %v", err)
    }

    // Створення сервісу
    createCmd := exec.Command("kubectl", "create", "-f", "service.yaml")
    if err := createCmd.Run(); err != nil {
        t.Errorf("Failed to create Service: %v", err)
    }

    // Перевірка наявності сервісу
    describeCmd := exec.Command("kubectl", "describe", "service", "my-service")
    if err := describeCmd.Run(); err != nil {
        t.Errorf("Failed to describe Service: %v", err)
    }

    // Видалення сервісу
    deleteCmd := exec.Command("kubectl", "delete", "service", "my-service")
    if err := deleteCmd.Run(); err != nil {
        t.Errorf("Failed to delete Service: %v", err)
    }

    // Зупинка Minikube
    stopCmd := exec.Command("minikube", "stop")
    if err := stopCmd.Run(); err != nil {
        t.Errorf("Failed to stop Minikube: %v", err)
    }
}
```

load_balancing_test.go

```
package work

import (
    "os/exec"
    "testing"
)

// Тест на балансування навантаження
func TestLoadBalancing(t *testing.T) {
    // Старт Minikube
    startCmd := exec.Command("minikube", "start", "--vm-driver=docker", "--force")
    if err := startCmd.Run(); err != nil {
        t.Errorf("Failed to start Minikube: %v", err)
    }
}
```

```

}

// Створення сервісу
createCmd := exec.Command("kubectl", "create", "-f", "service.yaml")
if err := createCmd.Run(); err != nil {
    t.Errorf("Failed to create Service: %v", err)
}

// Виконання запиту до сервісу
curlCmd := exec.Command("kubectl", "run", "my-service", "--image=radial/busyboxplus:curl")
if err := curlCmd.Run(); err != nil {
    t.Errorf("Failed to perform request: %v", err)
}

// Видалення сервісу
deleteCmd := exec.Command("kubectl", "delete", "service", "my-service")
if err := deleteCmd.Run(); err != nil {
    t.Errorf("Failed to delete Service: %v", err)
}

// Зупинка Minikube
stopCmd := exec.Command("minikube", "stop")
if err := stopCmd.Run(); err != nil {
    t.Errorf("Failed to stop Minikube: %v", err)
}
}

```

scale_pod_test.go

```

package work

import (
    "os/exec"
    "testing"
)

// Тест для масштабування Pod
func TestScalePod(t *testing.T) {
    // Старт Minikube
    startCmd := exec.Command("minikube", "start", "--vm-driver=docker", "--force")
    if err := startCmd.Run(); err != nil {
        t.Errorf("Failed to start Minikube: %v", err)
    }

    createCmd := exec.Command("kubectl", "create", "deployment", "my-deployment", "--image=my-image:latest")
    if err := createCmd.Run(); err != nil {
        t.Errorf("Failed to create Service: %v", err)
    }

    // Масштабування Pod
    scaleCmd := exec.Command("kubectl", "scale", "deployment", "my-deployment", "--replicas=3")
    if err := scaleCmd.Run(); err != nil {
        t.Errorf("Failed to scale Pod: %v", err)
    }

    // Видалення сервісу
    deleteCmd := exec.Command("kubectl", "delete", "deployment", "my-deployment")
}

```

```

if err := deleteCmd.Run(); err != nil {
    t.Errorf("Failed to delete Service: %v", err)
}

// Зупинка Minikube
stopCmd := exec.Command("minikube", "stop")
if err := stopCmd.Run(); err != nil {
    t.Errorf("Failed to stop Minikube: %v", err)
}
}

```

update_pod_test.go

```

package work

import (
    "os/exec"
    "testing"
)

// Тест на оновлення Pod
func TestUpdatePod(t *testing.T) {
    // Старт Minikube
    startCmd := exec.Command("minikube", "start", "--vm-driver=docker", "--force")
    if err := startCmd.Run(); err != nil {
        t.Errorf("Failed to start Minikube: %v", err)
    }

    // Оновлення Pod
    updateCmd := exec.Command("kubectl", "apply", "-f", "updated-pod.yaml")
    if err := updateCmd.Run(); err != nil {
        t.Errorf("Failed to update Pod: %v", err)
    }

    // Перевірка оновленого Pod
    describeCmd := exec.Command("kubectl", "describe", "pod", "my-pod")
    if err := describeCmd.Run(); err != nil {
        t.Errorf("Failed to describe Pod: %v", err)
    }

    // Зупинка Minikube
    stopCmd := exec.Command("minikube", "stop")
    if err := stopCmd.Run(); err != nil {
        t.Errorf("Failed to stop Minikube: %v", err)
    }
}

```

pod.yaml

```

apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: nginx:latest
      ports:
        - containerPort: 80

```

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

updated-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: my-updated-image:latest
      ports:
        - containerPort: 80
```