Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра комп’ютерних наук

Пояснювальна записка

до магістерської кваліфікаційної роботи
(освітньо-кваліфікаційний рівень)
на тему:

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ ПОБУДОВИ ОБЧИСЛЮВАЛЬНОЇ ПЛАТФОРМИ ДЛЯ РОЗМІЩЕННЯ І ПІДТРИМКИ СЕРВЕРНИХ ДОДАТКІВ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

Виконав: студент 2 курсу, гр. 1КН-19м
спеціальності
122 «Комп'ютерні науки»
Ткачук А.С.
(прізвище та ініціали)

Керівник д.т.н., проф. Яровий А.А.
(прізвище та ініціали)
Рецензент к.т.н., доц. Черноволик Г.О.
(прізвище та ініціали)

Вінниця - 2020 року

ЗАТВЕРДЖУЮ
_______ ФОП Ткачук А.С.
«__» ___________ 2020 р.

ЗАТВЕРДЖУЮ
Зав. каф. КН, д.т.н., проф.
_______ А.А. Яровий
«__» ___________ 2020 р.

ЗАВДАННЯ

на магістерську кваліфікаційну роботу на здобуття кваліфікації магістра зі спеціальності: 122 - «Комп'ютерні науки»

08-22.МКР.014.19.000.ПЗ

Магістранта групи 1КН-19м Ткачука Анатолія Сергійовича

Тема магістерської кваліфікаційної роботи: "Інформаційна технологія побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою"

Вхідні дані: Кількість майстер-серверів - не менше 1; кількість демон-серверів - не менше 2; наявність серверу для статичних файлів; використання ОС Ubuntu 18.04 (дистрибутив Linux); наявність підсистеми логування та збору і аналізу статистичних даних; наявність моніторингу; наявність підсистеми автоматичного розгортування клієнтських додатків.

Короткий зміст частин магістерської кваліфікаційної роботи:

1. Графічна: UML-діаграма прецедентів обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою, UML-діаграма компонентів, загальна структурна схема платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою, схема алгоритму роботи плейбуку для встановлення кластера Kubernetes, схема алгоритму роботи плейбуку для створення середовища для нового додатку на платформі, схема алгоритму розгортування додатку з Helm-чарту за допомогою Jenkins, схема алгоритму контролю реплік додатків Kubernetes-ом.
2. Текстова (пояснювальна записка): вступ, обґрунтування доцільності розробки інформаційної технології побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою, аналіз параметрів та програмного інструментарію розробки підсистем платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою, проектування та програмна реалізація інформаційної системи побудови платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою, економічна частина, висновки, перелік використаних джерел, додатки.

КАЛЕНДАРНИЙ ПЛАН

№ етапу	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Обґрунтування доцільності розробки інформаційної технології побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою	12.10.20	14.10.20	Аналітичний огляд літературних джерел, задачі досліджень, розділ 1 ПЗ
2	Аналіз параметрів платформ та ПЗ яке використовується для реалізації підсистем платформи	15.10.20	18.10.20	розділ 2
3	Програмна реалізація розробленої інформаційної технології, тестування та оцінка параметрів	19.10.20	22.10.20	розділ 3
4	Підготовка економічної частини	23.10.20	26.10.20	розділ 4
5	Апробація та/або впровадження результатів дослідження	27.10.20	31.10.20	тези доповідей, акт впровадження
6	Оформлення пояснювальної записки, графічного матеріалу та презентації	01.11.20	05.11.20	Пояснювальна записка, графічний матеріал, презентація

Консультанти з окремих розділів магістерської кваліфікаційної роботи

1. Науковий керівник _____________ д.т.н., професор
(підпис) наук. ступінь, вчене звання (посада)
“____” _________2020 р. А. А. Яровий
ініціали та прізвище

2. Економічна частина ______________ к.е.н, доцент кафедри ЕПВМ
(підпис) наук. ступінь, вчене звання (посада)
“____” _________2020 р. Бальзан М.В
ініціали та прізвище

Дата попереднього захисту роботи “_06___” ____11_____2020 р.

Рецензент ______________ к.т.н, доц. кафедри ПЗ
(підпис) наук. ступінь, вчене звання (посада)
Г. О. Черноволик
ініціали та прізвище

Завдання видав
науковий керівник ______________ д.т.н., професор
(підпис) наук. ступінь, вчене звання (посада)
А. А. Яровий
ініціали та прізвище

“_08__” ____09_____2020 р.

Завдання отримав магістрант __________ А. С. Ткачук
(підпис) ініціали та прізвище
“__08__” ____09_____2020 р

АНОТАЦІЯ

У даній магістерській кваліфікаційній роботі реалізується інформаційна технологія побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою. Її призначенням є зменшення часу, який витрачається на розгортування додатку на платформі.

В ході роботи проведено аналіз предметної області платформ для розміщення і підтримки серверних додатків та систем-аналогів. Розглянуто існуючі методи, що можуть бути використані у створенні технології побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою. Спроектовано підсистеми інформаційної технології побудови платформи для розміщення і підтримки серверних додатків.

Здійснено програмну реалізацію технології побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою на базі системи Kubernetes з використанням контейнеризації на базі Docker-контейнерів, а також із застосуванням таких програмних засобів як Ansible, Jenkins, Helm, Prometheus, Grafana, Elasticsearch, Logstash, Kibana.

ABSTRACT

In this master's qualification work the information technology of construction of a computer platform for placement and support of server applications with microservice architecture was realized. Its purpose is to reduce the time spent deploying the application on the platform.

During the work, the analysis of the subject area of platforms for placement and support of server applications and systems-analogues were analyzed. Existing methods that can be used to create a technology for building a computing platform for hosting and supporting server applications with microservice architecture were considered. Subsystems of information technology for building a platform for hosting and supporting server applications were designed.

Software implementation of computer platform construction technology for hosting and support of server applications with microservice architecture based on Kubernetes system using containerization based on Docker containers, as well as using software such as Ansible, Jenkins, Helm, Prometheus, Grafana, Logstash, Kibana.

ЗМІСТ

1 ОБҐРУНТУВАННЯ ДОЦІЛЬНОСТІ РОЗРОБКИ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ ПОБУДОВИ ОБЧИСЛЮВАЛЬНОЇ ПЛАТФОРМИ ДЛЯ РОЗМІЩЕННЯ І ПІДТРИМКИ СЕРВЕРНИХ ДОДАТКІВ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ 11
1.1 Аналіз архітектури серверних додатків та платформ для розміщення і підтримки серверних додатків з мікросервісною архітектурою 11
1.1.1 Аналіз архітектури серверних додатків 11
1.1.1.1 Монолітна архітектура серверних додатків 11
1.1.1.2 Мікросервісна архітектура серверних додатків 12
1.1.1.3 Порівняння монолітної та мікросервісної архітектури серверних додатків 14
1.1.2 Аналіз платформ для розміщення і підтримки розподілених серверних додатків з мікросервісною архітектурою. 19
1.1.2.1 Апаратний сервер 19
1.1.2.2 Віртуальний сервер 21
1.1.2.3 Віртуальний контейнер 23
1.2 Аналіз об'єкту проектування 25
1.3 Характеристика і аналіз аналогів 28
1.4 Висновок 31
2 АНАЛІЗ ОСОБЛИВОСТЕЙ ТА ПРОГРАМНОГО ІНСТРУМЕНТАРІЮ РОЗРОБКИ ПІДСИСТЕМ ПЛАТФОРМИ ДЛЯ РОЗМІЩЕННЯ І ПІДТРИМКИ СЕРВЕРНИХ ДОДАТКІВ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ 32
2.1 Аналіз особливостей платформ для розміщення і підтримки серверних додатків з мікросервісною архітектурою 32
2.2 Аналіз програмного інструментарію, що використовується для реалізації підсистем платформи 33
2.2.1 Аналіз програмного інструментарію для контейнерування 33
2.2.2 Аналіз програмного інструментарію для реалізації підсистеми управління контейнерами 35
2.2.3 Аналіз програмного інструментарію для реалізації підсистеми автоматизування встановлення платформи 37
2.2.4 Аналіз програмного інструментарію для реалізації підсистеми автоматизованого розгортування клієнтських додатків 41
2.2.5 Аналіз програмного інструментарію для реалізації підсистеми логування та збору і аналізу статистичних даних 43
2.2.6 Аналіз програмного інструментарію для реалізації підсистеми моніторингу 45
2.2.7 Аналіз сервісів для реалізації підсистеми збереження конфігураційних файлів 50

2.2.8 Аналіз програмного інструментарію необхідного для функціонування серверу для статичних файлів 53
2.3 Специфічні аспекти побудови платформ на базі оркестратору контейнерів Kubernetes 54
2.3.1 Структура та основні елементи Kubernetes 54
2.3.2 Основні абстракції Kubernetes 55
2.3.3 Безпека в Kubernetes 59
2.3.4 Управління ресурсами в Kubernetes 60
2.4 Висновок 61
3 ПРОЕКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ ПОБУДОВИ ПЛАТФОРМИ ДЛЯ РОЗМІЩЕННЯ І ПІДТРИМКИ СЕРВЕРНИХ ДОДАТКІВ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ 62
3.1 Проектування платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою 62
3.2 Розробка структури платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою 65
3.3 Розробка схеми алгоритму роботи платформи 67
3.4 Тестовий приклад роботи платформи і аналіз результатів 70
3.4.1 Тестування автоматичного встановлення платформи 70
3.4.2 Тестування автоматизованого розгортування додатків на платформі 75
3.4.3 Тестування автоматизованої підтримки роботи додатків на платформі 81
3.4.4 Аналіз результатів 82
3.5 Висновок 84
4 ЕКОНОМІЧНА ЧАСТИНА 85
4.1 Оцінювання комерційного потенціалу розробки 85
4.2 Прогнозування витрат на виконання науково-дослідної роботи та конструкторсько–технологічної роботи 87
4.3 Прогнозування комерційних ефектів від реалізації результатів розробки 91
4.4 Розрахунок ефективності вкладених інвестицій та період їх окупності .. 94
4.5 Висновок 98
ВИСНОВКИ 99
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 101
Додаток А Акт впровадження 107
Додаток Б Інструкція користувача 108
Додаток В Лістинг програми 111
Додаток Г Графічна частина 125

ВСТУП

Актуальність. Із активним розвитком комп'ютерних технологій не менш активно розвивається і Інтернет. В наш час Інтернет є невід'ємною частиною багатьох сервісів, таких як файлообмінні мережі, мережі хмарних обчислень, електронні платіжні системи, системи обміну повідомленнями, електронна пошта тощо. Все більше додатків створюється на основі поєднання двох архітектур. А саме, на основі клієнт-серверної архітектури, для зменшення навантаження на пристрої користувачів [1], та мікросервісної архітектури для спрощення розробки великих додатків [2].

При використанні клієнт-серверної архітектури безперечно виникає потреба у наявності серверу, на якому буде розміщено серверну частину додатку. В такому разі сервера частина повинна приймати запити від клієнтів, обробляти їх певним чином і надсилати клієнтові результат виконання запитів. Однак, з використанням мікросервісної архітектури сервісний додаток може бути розділений на декілька логічних частин, кожна з яких зазвичай розміщується на різних серверах або і взагалі на групі серверів [3]. Проте, частина розробників може просто не мати власного серверу. І навіть просто наявності серверу чи групи серверів недостатньо. Самі сервери потрібно вміти налаштовувати, потрібно з’єднати їх в мережу, налаштувати процеси автоматизації роботи з серверами і так далі.

Саме для випадків, коли у команди розробників програмного забезпечення недостатньо знань та вмінь для самостійного проведення таких робіт були створені обчислювальні платформи для розміщення і підтримки серверних додатків. Вибір подібної платформи дуже важливий, так як він буде впливати на ціну, якість та надійність роботи, швидкість оновлення та зручність розробки додатку на протязі значної частини життєвого циклу програмного забезпечення. Обчислювальні платформи для розміщення і підтримки серверних додатків дозволяють значно спростити процес розгортування, оновлення та моніторингу додатку на сервері для розробників.

Зв’язок роботи з науковими програмами, планами, темами.

Магістерська робота виконана відповідно до напрямку наукових досліджень кафедри комп’ютерних наук Вінницького національного технічного університету 22 К1 «Розробка спеціалізованих засобів штучного інтелекту на основі інтелектуального аналізу даних та машинного навчання».

Мета і завдання досліджень. Метою магістерської кваліфікаційної роботи є зменшення показника часу, який витрачається на розгортування нових додатків або їх підсистем та версій на платформі.

Для досягнення мети розробки необхідно виконати наступні задачі:

- здійснити обґрунтування доцільності розробки інформаційної технології побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою;
- здійснити проектування обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою;
- обґрунтувати вибір програмного інструментарію для реалізації інформаційної технології обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою;
- здійснити програмну реалізацію інформаційної технології обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою;
- здійснити тестування інформаційної технології обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою.

Об’єктом дослідження є процес підтримки роботи серверних додатків з мікросервісною архітектурою.

Предметом дослідження є програмні засоби побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою.

Методи дослідження. Для досягнення мети дослідження застосовувалися методи моделювання UML, модель представлення хмарних обчислень PaaS, концепція керування інфраструктурою IaaC, підхід до адміністрування DevOps, розглядались патерни проектування мікросервісів, такі як Aggregator, Proxy, Chained, Shared Data.

Наукова новизна одержаних результатів.

1. Удосконалено інформаційну модель обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою, яка на відміну від існуючих містить сервер зі статичними даними, сервіси збереження конфігураційних файлів та логування, збору і аналізу статистичних даних, що забезпечило розширення функціональних можливостей та підвищило зручність керування програмними додатками в межах платформи;
2. Розроблено інформаційну технологію обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою, яка містить удосконалену інформаційну модель, що забезпечило зменшення витрат часу на розгортування програмного додатку на платформі.

Практичне значення одержаних результатів полягає у розробленому на основі проведених досліджень програмному забезпеченні, що реалізує інформаційну технологію обчислювальної платформи для розміщення і підтримки серверних додатків. Розроблена інформаційна технологія підвищує швидкість розгортування нових додатків або їх підсистем та версій на платформі завдяки наступним аспектам:

- розроблено алгоритм розгортування додатку з Helm-чарту за допомогою Jenkins;
- розроблено алгоритм роботи плейбуку створення середовища для нового додатку на платформі;
- програмно реалізовано платформу для розміщення і підтримки серверних додатків з мікросервісною архітектурою.

Достовірність теоретичних положень магістерської кваліфікаційної роботи підтверджується коректністю постановки завдання, коректністю використання підходів до реалізації платформи, експериментальними дослідженнями тестування програмної реалізації інформаційної технології обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою.

Особистий внесок здобувача. Результати даної магістерської кваліфікаційної роботи отримані самостійно. В публікації у співавторстві здобувачу належить аналіз наявних технологій для побудови систем для підтримки роботи розподілених додатків [4].

Апробація результатів роботи. Результати досліджень було апробовано на XLIX науково-технічній конференції професорсько-викладацького складу, співробітників та студентів Вінницького національного технічного університету у 2020р.

Публікації. За основними результатами досліджень опубліковано одну публікацію [4], а також подано до розгляду свідоцтво про реєстрацію авторського права на твір (комп’ютерну програму).

1 ОБҐРУНТУВАННЯ ДОЦІЛЬНОСТІ РОЗРОБКИ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ ПОБУДОВИ ОБЧИСЛЮВАЛЬНОЇ ПЛАТФОРМИ ДЛЯ РОЗМІЩЕННЯ І ПІДТРИМКИ СЕРВЕРНИХ ДОДАТКІВ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

1.1 Аналіз архітектури серверних додатків та платформ для розміщення і підтримки серверних додатків з мікросервісною архітектурою

1.1.1 Аналіз архітектури серверних додатків

1.1.1.1 Монолітна архітектура серверних додатків

Монолітний додаток повністю замкнутий в контексті поведінки. Під час роботи він може взаємодіяти з іншими службами або сховищами даних, проте основа його поведінки реалізується у власному процесі, а весь додаток зазвичай розгортується як один елемент. Для горизонтального масштабування такий додаток зазвичай цілком дублюється на декількох серверах або віртуальних машинах [5]. Типова структура монолітного додатку наведена на рисунку 1.1.

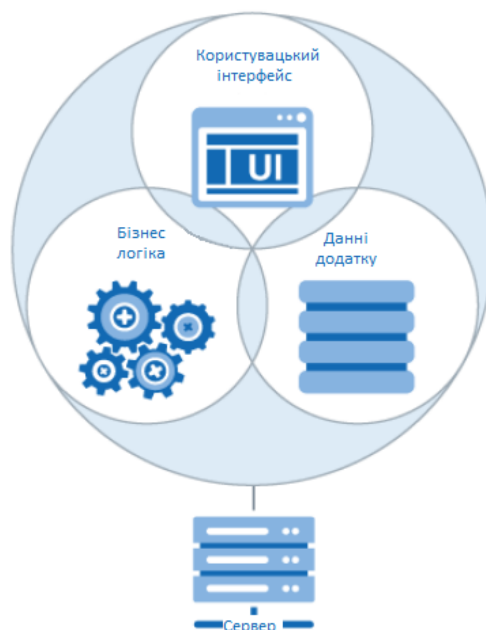

Рисунок 1.1 – Структура монолітного серверного додатку

Всі функції монолітного додатку або основна їх частина зосереджені в одному процесі або контейнері, який розбивається на внутрішні шари або бібліотеки. Недолік цього підходу стає очевидним, коли додаток розростається і його необхідно масштабувати. Якщо додаток масштабується як єдине ціле, все вийде. Але в більшості випадків необхідно масштабувати лише деякі частини програми, поки інші компоненти працюють нормально.

Для прикладу, в програмі для онлайн-магазину, найімовірніше, буде потрібно масштабування компонента з відомостями про товари. Клієнти частіше переглядають товари, ніж купують їх. Клієнти частіше складають товари в кошик, ніж оплачують їх. Не так багато клієнтів пишуть коментарі або переглядають історію покупок.

При масштабуванні монолітних рішень весь код розгортується багаторазово. Крім того, що необхідно масштабувати всі компоненти, зміни в одному компоненті вимагають повного повторного тестування всієї програми і повного повторного розгортування всіх її копій.

Монолітний підхід знайшов широке поширення і використовується багатьма організаціями при розробці архітектури. У багатьох випадках це дозволяє домогтися бажаних результатів, але іноді організація стикається з обмеженнями.

У багатьох організаціях додатки будувалися за такою моделлю, так як кілька років тому за допомогою існуючих інструментів та інфраструктури надто складно було створювати розподілені мікросервісні архітектури, і проблем не виникало, поки програма не починала розростатися [6].

1.1.1.2 Мікросервісна архітектура серверних додатків

Сьогодні все більше фахівців сходяться на тому, що при побудові великого, складного додатка слід задуматися про використання мікросервісів.

Узагальнене визначення мікросервісної архітектури (або мікросервісів) звучить так: це стиль проектування, який розбиває додаток на окремі

сервіси з різними функціями [7]. Приклад взаємодії мікросервісного додатку з клієнтом наведено на рисунку 1.2

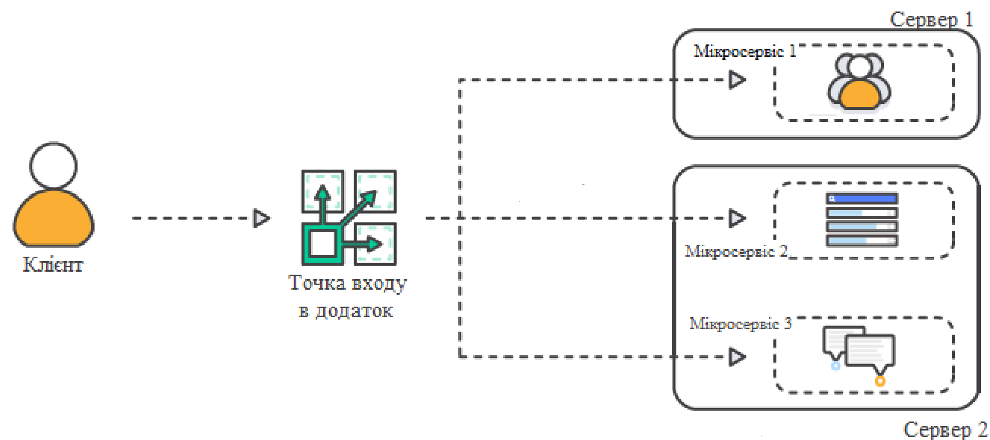

Рисунок 1.2 – Приклад взаємодії мікросервісного додатку з клієнтом

Сервіс - це міні-додаток, що реалізує вузькоспеціалізовані функції, наприклад, в контексті ресторану, такі як управління замовленнями, управління клієнтами і так далі.

Ключовою особливістю мікросервісної архітектури є те, що сервіси слабо пов'язані між собою і взаємодіють тільки через API. Слабкої пов'язаності можна досягти за рахунок виділення кожному сервісу окремої бази даних [7].

Також, самі мікросервіси можна розбити на дві основні групи. А саме:

– Stateless (сервіси без збереження стану);
– Persistence (сервіси, які зберігають свій стан).

Stateless це такі мікросервіси, які не мають ніякої залежності від storage (cache і т.д.) і визначають дуже прості дії. Приклад - конвертація з однієї валюти в іншу. Дуже швидка дія, немає залежності від інших сервісів і від дискового простору. Їх легко замінити і легко масштабувати.

Persistence – це інший тип мікросервісів які зберігають свій стан. До них відносяться сервіси, які зберігають залежність від storage, у них є операції запису та/або читання, їх точно так само легко масштабувати, навіть якщо є залежність від диску [8].

1.1.1.3 Порівняння монолітної та мікросервісної архітектури серверних додатків

У кожної з названих вище архітектур звісно є свої переваги та недоліки. Говорячи про монолітну архітектуру можна виокремити такі переваги:

- Простота розробки - IDE і інші інструменти розробки зосереджені на побудові єдиного додатку;
- Легкість внесення радикальних змін - ви можете поміняти код і структуру бази даних, а потім зібрати і розгорнути отриманий результат;
- Простота тестування;
- Простота розгортування - розробнику досить скопіювати файли додатку на сервер із встановленим ПЗ;
- Відносна легкість масштабування – можна просто запустити кілька копій додатку, розміщених за балансувальником навантаження [7].

Однак, виникає і ряд недоліків, таких як:

- Збільшення об’єму кодової бази з ростом додатку;
- Великий додаток часто складніше підтримувати і повністю розуміти новим розробникам;
- При розробці функціоналу одного додатку декількома командами можуть виникати конфлікти в кодовій базі;
- Компіляція коду в одну програму займає багато часу;
- Довгий процес тестування програми при внесені змін через необхідність виконувати всі тести відразу;
- Неможливість використання безперервної доставки і розгортування додатку;
- Неоднакові вимоги до апаратного забезпечення зі сторони різних модулів додатку [7].

Організацію процесу роботи розробників над додатком з монолітною архітектурою наведено на рисунку 1.3.

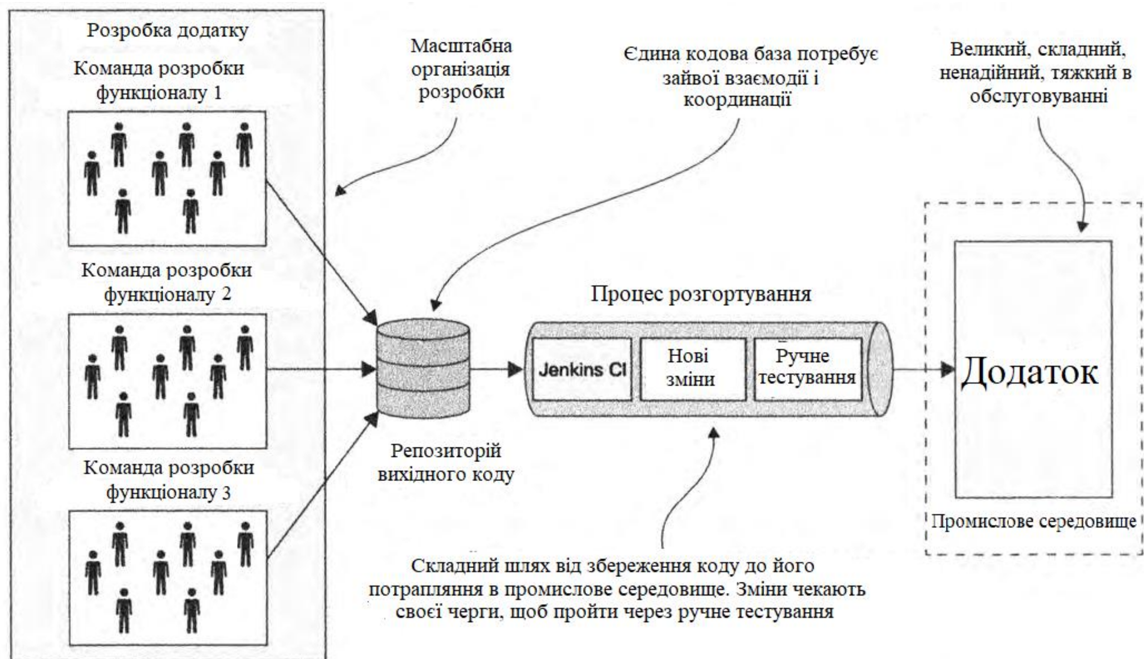

Рисунок 1.3 – Організація процесу роботи розробників над додатком з монолітною архітектурою

Коли ж мова йде про мікросервісну архітектуру, то можна перелічити такі переваги:

- Розбиття на сервіси (модулі) дає можливість змінювати частини додатку незалежно одна від іншої. Також це дає можливість розробляти різні модулі додатку на різних мовах програмування без виникнення додаткових складнощів з інтегруванням коду;
- Кожний сервіс легко ізолювати від інших, що дозволяє значно збільшити безпеку всього додатку в цілому;
- Вихід з ладу частини сервісів не впливатиме на загальну роботу додатку. Додаток продовжить функціонувати за винятком функцій, за які відповідають сервіси що вийшли з ладу;
- В кожного сервісу своя база даних, тому обмін інформацією з БД одного сервісу не змушує чекати своєї черги іншого сервісу, так як БД в кожного сервісу своя;

– Сервіси масштабуються незалежно один від одного;
– Мікросервісна архітектура забезпечує більшу автономність команд розробників;
– Стає можливим безперервна доставка і розгортування великих та складних додатків [7].

Організація роботи розробників над додатком з мікросервісною архітектурою зображена на рисунку 1.4.

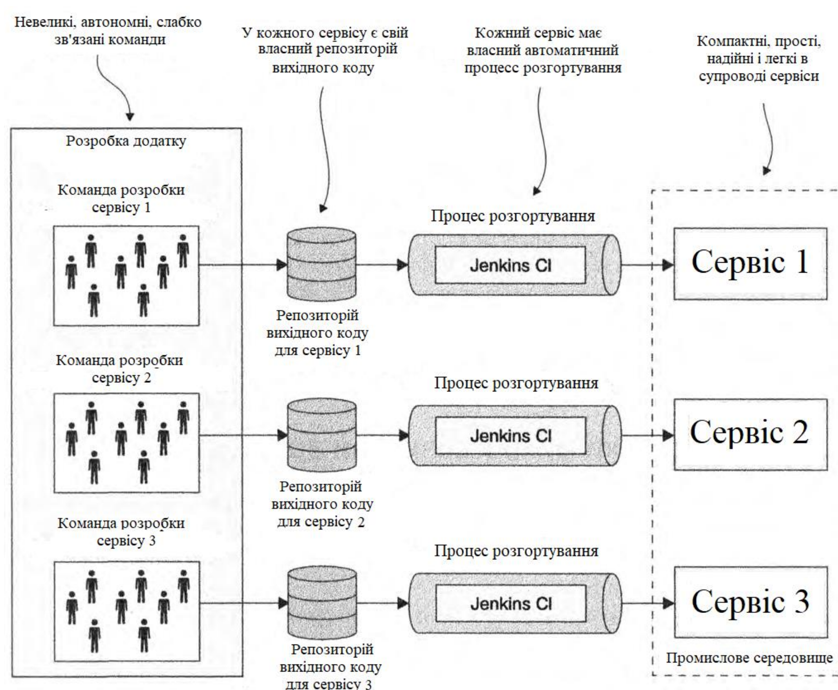

Рисунок 1.4 – Організація роботи розробників над додатком з мікросервісною архітектурою

Але, використання мікросервісної архітектури має і декілька недоліків:
– Складно підібрати підходящий набір сервісів під кожен додаток;
– Складність розподілених систем ускладнює процес розробки, тестування і розгортування;
– Розгортування нових функцій, які охоплюють декілька сервісів, вимагають чіткої координації [7];
– Розподілені мікросервісні додатки залежні від якості мережі між ними.

Підсумовуючи всі переваги та недоліки обох архітектур було зроблену порівняльну таблицю, яка представлена у вигляді Таблиці 1.1.

Таблиця 1.1 – порівняння монолітної та мікросервісної архітектур

	Монолітна архітектура	Мікросервісна архітектура
Складність розробки	Низька	Висока
Складність підтримки	Висока	Низька
Легкість та гнучкість масштабування	Середня	Висока
Необхідність чіткої взаємодії між командами розробників	Висока	Низька
Залежність розгортування коду однієї команди від іншої	Є	Немає
Залежність від якості мережі	Низька	Висока

Виходячи з таблиці порівняння монолітної та мікросервісної архітектур, а також з усіх наведених переваг та недоліків кожної з архітектур, можна дійти висновку, що монолітна архітектура краще підходить для:

- невеликих додатків;
- додатків, які не будуть активно змінюватись з часом;
- додатків, розробкою яких займається невелика група людей.

Натомість, мікросервісну архітектуру краще застосовувати для:

- додатків середніх та великих розмірів;
- додатків, над якими працюють багато команд;
- додатків, які планують активно розвивати;

- додатків, для яких необхідне використання декількох мов програмування.

Також, мікросервісну архітектуру застосовують такі компанії, як Uber, Netflix, Amazon, eBay, SoundCloud [9], Tinder, Airbnb, Spotify [10] та багато інших.

Так як більшість додатків активно розвивається з плином часу, а різні мови програмування можуть підходити для різних задач всередині додатку, то кращим вибором буде мікросервісна архітектура.

Мікросервісна архітектура має декілька патернів проектування, серед яких [11]:

- Aggregator - у найпростішій формі агрегатор є звичайною веб-сторінкою, що викликається набором сервісів для реалізації, необхідного в додатку. Всі послуги онлайн (послуга А, послуга В і послуга С), за допомогою легкого REST-механізму веб-сторінка може отримати дані і обробити / відобразити їх як потрібно. Якщо потрібна будь-яка обробка, наприклад, застосувати бізнес-логіку до даних, отриманих від окремих сервісів, то для цього може бути створений CDI-компонент, що перетворює дані таким чином, щоб їх можна було вивести на веб-сторінці;
- Proxy - патерн «посередник» при роботі з мікросервісами - це варіант агрегатора. В такому випадку агрегація повинна відбуватися на стороні клієнта, але в залежності від бізнес-вимог при цьому може викликатися додатковий мікросервіс;
- Chained - мікросервісний патерн проектування «Ланцюжок» видає єдину консолідовану відповідь на запит. Проте, не можна робити ланцюжок занадто довгим. Це критично, оскільки ланцюжок синхронний за своєю природою, і чим він довша, тим довше доведеться чекати клієнтові, особливо якщо відгук полягає у виведенні веб-сторінки на екран;

- Shared Data - При цьому патерні кілька мікросервісів можуть працювати в ланцюжку і спільно використовувати сховища кеша і бази даних. Його використання доцільно лише в разі, якщо між двома сервісами існує сильний зв'язок.

1.1.2 Аналіз платформ для розміщення і підтримки розподілених серверних додатків з мікросервісною архітектурою.

Розподілені серверні додатки можуть бути розміщені на різних платформах. До таких відносять:

- Апаратний (виділений) сервер;
- Віртуальний сервер;
- Віртуальний контейнер [12].

Кожна з них має свої переваги та недоліки.

1.1.2.1 Апаратний сервер

Апаратний сервер – це окремий, фізичний комп’ютер, розміщений в дата-центрі [13].

При реалізації розподілених систем за допомогою виділених серверів елементи системи та програмного додатку розміщуються безпосередньо на окремих серверах [4].

До його переваг можна віднести:

- Висока продуктивність - користувач використовує сервер безпосередньо, минаючи рівні абстракції і віртуалізацію;
- Висока надійність - невелика кількість елементів, схильних до поломок і неполадок;
- Раціональне використання ресурсів - при використанні виділених серверів процеси не борються з іншими процесами або

віртуальними машинами за обчислювальні потужності процесора, пам'ять і мережеві ресурси [14].

До недоліків відносять:

- Складність управління - клонувати виділений сервер не так легко - для нього не можна створити образ, який потім просто буде працювати в іншому місці;
- Ціна - в більшості випадків ви платите за апаратну частину, а також за її розміщення, або просто орендуєте її у хостинг-провайдера. У будь-якому випадку, у вас не вийде просто відключити сервер, якщо він раптом став вам не потрібен - доводиться ретельно планувати витрати;
- Всі процеси і програми, запущені на виділеному сервері, працюють на одній операційній системі. Щоб домогтися гарної масштабованості, зазвичай потрібно виконувати одну задачу на одному сервері - наприклад, розділити веб-сервер і сервер баз даних. Складно оптимізувати роботу операційної системи, коли вона повинна справлятися з безліччю завдань відразу [14];
- Відносно складний процес оновлення системи та складових апаратного забезпечення [4].

При розгляді розгортування мікросервісних додатків на базі апаратних серверів, як правило, використовують вже зібрані пакети, як наприклад, для мови Java – JAR. Тож в даному аспекті також виникає ряд переваг та недоліків [7].

Переваги:

- Швидке розгортування копії;
- Ефективне використання ресурсів [7].

Недоліки:

- Відсутність інкапсуляції стеку технологій;
- Неможливість обмеження ресурсів, які буде використовувати кожен екземпляр сервісу;

– Недостатня ізоляція при запуску декількох екземплярів сервісу на одному комп’ютері;
– Складнощі з автоматичним визначенням місця розміщення екземплярів сервісу [7].

Розгортування мікросервісних додатків на базі апаратного серверу показано на рисунку 1.5.

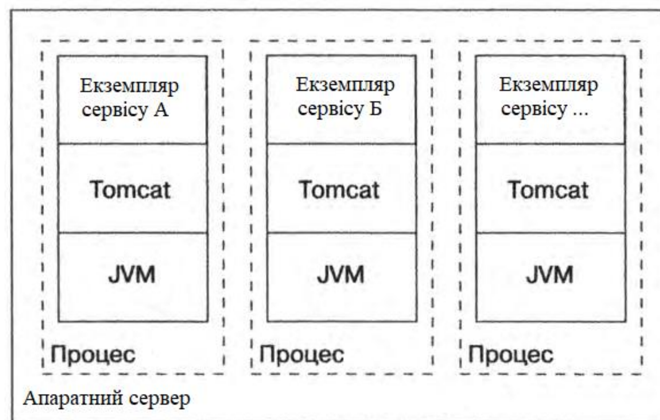

Рисунок 1.5 – Розгортування мікросервісних додатків на базі апаратного серверу

1.1.2.2 Віртуальний сервер

Віртуалізація дозволяє розділити виділений сервер на маленькі віртуальні сервери, які мають доступ лише до частини всіх ресурсів фізичного сервера.

Для прикладу, можна взяти фізичний сервер з двома чотирьох ядерними процесорами і 16 Гб оперативної пам'яті і перетворити його в 8 віртуальних машин, де на кожну буде виділено одне ядро і 2 Гб оперативної пам'яті [14].

Прикладами технологій віртуалізації можуть служити Xen [15], KVM [16], VMware [17], Hyper-V [18] і ще багато інших.

Загальна структура будь яких віртуальних серверів і додатків в них зображена на рисунку 1.6.

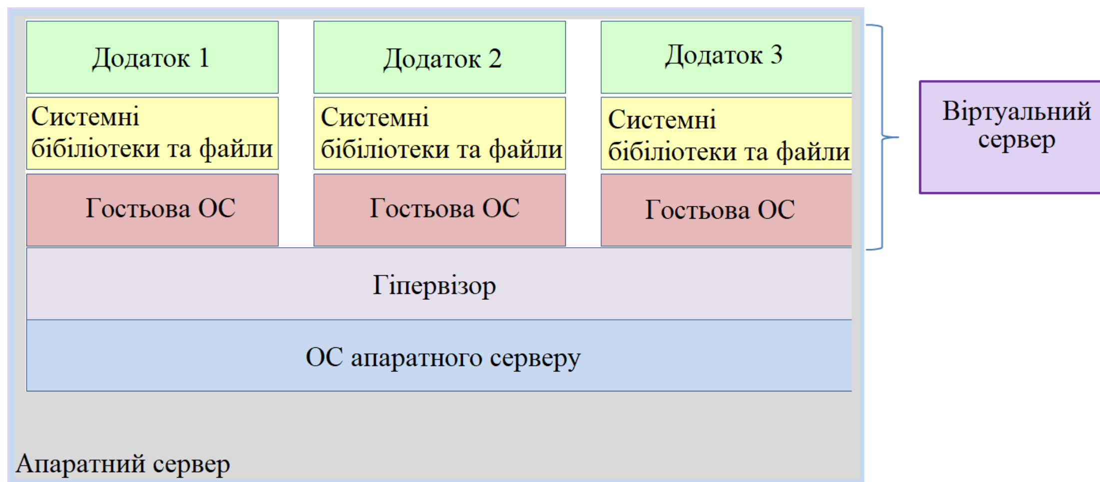

Рисунок 1.6 – Загальна структура віртуальних серверів з додатками

З переваг віртуальних серверів, як платформи для розміщення і підтримки серверних додатків, можна назвати:

– Можливість клонування віртуальної машини, якщо потрібна більша їх кількість при зростанні навантаження;
– Можна зберігати образи віртуальних машин, щоб можна було легко відновити їх [14];
– Можливість виділення конкретних ресурсів кожному вузлу системи [4].

Проте, є і ряд недоліків:

– Ресурси сервера можуть використовуватися не в повному обсязі. Наприклад, якщо виділити на сервері місце під зберігання образу віртуальної машини, то цей обсяг пам'яті вже не можна буде використовувати для інших цілей, навіть якщо віртуальна машина, пов'язана з диском, не використовує весь виділений обсяг.
– За допомогою віртуальних машин не можна працювати з фізичним обладнанням безпосередньо. Наприклад, якщо буде необхідність, щоб замість процесору віртуальної машини обчислювальні операції виконувалися за допомогою GPU, то

нічого не вийде - принаймні, простими способами - так як віртуальна машина працює не в середовищі сервера.

- Як правило, віртуальні машини працюють не так швидко, як фізичні сервери, оскільки їх ресурси витрачаються на емуляцію апаратних засобів для віртуального сервера [19].

Якщо ж говорити про розгортування мікросервісних додатків всередині віртуальних серверів, то є такі переваги:

- Образ віртуального серверу інкапсулює стек технологій;
- Екземпляри сервісів ізольовані [7].

Проте, є і недоліки, які частково є наслідками недоліків самих віртуальних серверів:

- Менш ефективно використовуються ресурси;
- Розгортування протікає доволі повільно
- Потрібні додаткові витрати на системне адміністрування [7].

1.1.2.3 Віртуальний контейнер

Контейнери - це більш легкий сучасний механізм розгортування. Вони використовують механізм віртуалізації на рівні операційної системи.

Контейнер зазвичай складається з одного процесу (хоча їх може бути кілька), запущеного в середовищі, ізольованому від інших контейнерів. Процес, запущений всередині контейнера, поводиться так, ніби йому відведено цілий окремий сервер.

Для ізоляції контейнерів один від одного їх середовище виконання використовує механізми операційної системи. При створенні контейнера можна вказати його процесор, обсяг пам'яті і, в залежності від реалізації, ресурси введення / виведення [7].

Загальна структура додатку, який працює на базі контейнерів зображено на рисунку 1.7.

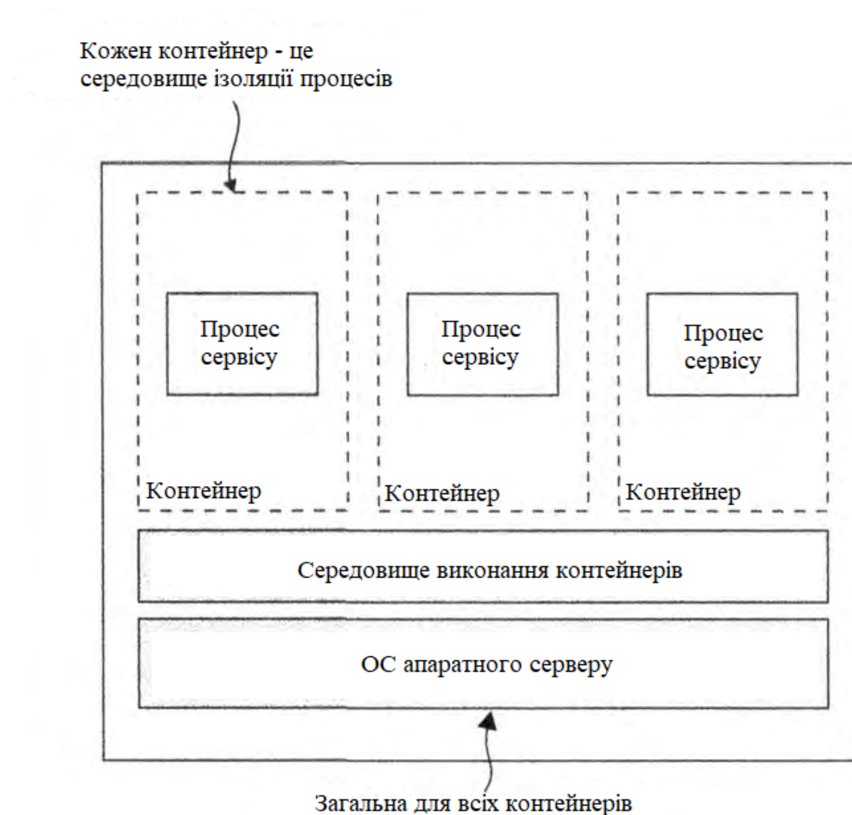

Рисунок 1.7 – Загальна структура додатку який працює на базі контейнерів

Використання контейнерів має свої переваги і дозволяє:

- Отримати доступ до апаратних засобів, не вдаючись до методів транзитної передачі. Додаток працює на тій же операційній системі, що і сервер;
- Ефективно використовувати ресурси системи;
- Отримати продуктивність виділеного сервера для додатків через відсутність рівня апаратної емуляції, яка відділяє їх від хост-сервера;
- Отримати можливість запускати додатки в середовищі, де можна швидко переносити з одного хост-сервера на інший;
- Ізолювати додатки. Функціонал дозволяє адміністраторам запобігати взаємодія додатків і обмежувати права і використання ресурсів для кожного контейнера [19];
- Незалежність додатків в контейнері від ОС серверу [20];

Однак, є і деякі недоліки:

- Потрібно займатись створенням і оновленням образів додатків;
- Зростає складність адміністрування. Замість одного або декількох серверів з декількома сервісами кожен з контейнерів представляє собою незалежну систему, кожен зі своїм сервісом [7];
- Виникає проблема забезпечення якості при значному збільшенні числа контейнерів;
- Проблема безпеки. Необхідно ретельно налаштовувати безпеку кожного контейнеру та системи в цілому [20].

1.2 Аналіз об’єкту проектування

Так як є декілька розповсюджених платформ, що застосовуються для розміщення серверних додатків з мікросервісною архітектурою, не зовсім очевидний вибір підходящої. Тому було прийнято рішення створити порівняльну табличку, результати якої наведені у Таблиці 1.2.

Зважаючи на всі переваги та недоліки всіх платформ можна зробити висновок, що віртуальні контейнери є найкращим вибором для розміщення і підтримки розподілених серверних додатків з мікросервісною архітектурою.

Також, не найменш важливим фактором є те, що контейнери використовують такі відомі компанії як BBC News, eBay, PayPal, Yandex, Uber[21], Netflix[22].

Однак, все ще не зникають недоліки контейнерів, серед яких зниження якості та складності управління при рості системи.

Тому, виникає необхідність в системі керування контейнерами. Такі системи називають оркестраторами контейнерів.

Таблиця 1.2 – порівняння платформ для розміщення і підтримки розподілених серверних додатків з мікросервісною архітектурою

	Апаратний сервер	Віртуальний сервер	Віртуальний контейнер
Ізоляція додатків	-	+	+
Можливість керування ресурсами для сервісів	-	+	+
Витрати ресурсів на додаткове ПЗ	-	+	-
Ціна реалізації	Висока	Середня	Низька
Складність масштабування	Висока	Середня	Низька
Складність управління	Низька	Середня	Висока
Складність оновлення одного сервісу	Висока	Висока	Низька
Швидкість запуску одного сервісу	Низька	Низька	Висока

Оркестрація - це координація взаємодії декількох контейнерів. Оркестрація дозволяє створювати інформаційні системи з безлічі контейнерів, кожен з яких відповідає тільки за одну певну задачу, а спілкування здійснюється через мережеві порти і загальні каталоги. При необхідності кожен такий контейнер можна замінити іншим, що дозволяє, наприклад, швидко перейти на іншу версію бази даних при необхідності. Засоби оркестрації дозволяють

реалізувати зручні та ефективні засоби розгортування контейнерних систем, побудови єдиної централізованої інфраструктури [23].

Проте, оркестратори не дають ніяких можливостей окрім власне управління контейнерами. А для підтримки додатків також необхідні моніторнг, система збору логів, інструменти які дозволять автоматизувати встановлення всієї системи, а також інструменти, які дозволять запускати додатки та оновлювати все існуючі додатки та їх окремі сервіси.

Отже, потрібно використовувати цілісну систему, яка задовольнить ці вимоги. Подібні системи носять назву PaaS. PaaS (Platform as a Service/Платформа як послуга) - це повноцінне середовище розробки і розгортування з ресурсами, які дозволяють розгортувати будь-які додатки, від простих до просунутих хмарних додатків промислового класу. Користувач купує необхідні ресурси у постачальника платформи, оплачує в міру використання і підключаєтеся до них за допомогою інтернет-підключення. PaaS являє собою одну з моделей представлення хмарних обчислень. Також можливі випадки, коли купується ліцензія на використання або ж просто вже готову PaaS систему і користувач використовує і обслуговує її самостійно. Деякі системи користувач зможе сам змінити під свої задачі. PaaS включає інфраструктуру (сервери, сховище та мережеве обладнання), а також ПО проміжного шару, засоби бізнес-аналітики (BI), служби системи управління базами даних і інше.

PaaS призначена для підтримки повного життєвого циклу веб-додатків: розробки, тестування, розгортування, управління та оновлення. PaaS також дозволяє уникнути витрат і труднощів, пов'язаних з створенням і встановленням базової інфраструктури додатків, ПЗ проміжного рівня, оркестратору контейнерів, та інших ресурсів, а також управлінням ними. Користувач керує додатками і службами, які розробляє, а постачальник платформи зазвичай керує всім іншим [24]. Якщо ж система була куплена для власного, незалежного використання, то підтримкою системи займаються системні адміністратори користувача.

Зважаючи на розповсюдженість мікросервісної архітектури, та переваги і недоліки всіх підходів до її розміщення на серверах, найкращим варіантом буде використання PaaS.

1.3 Характеристика і аналіз аналогів

Одними з найпопулярніших PaaS систем, які наразі представлені на ринку є OpenShift та Heroku.

Red Hat OpenShift - це гібридна хмарна платформа для корпоративних програм на основі Kubernetes. Red Hat OpenShift включає в себе оптимізовані робочі процеси, щоб допомогти командам швидше досягти випуску продукту, включаючи вбудовані конвеєри Jenkins [25]. Головне меню Red Hat OpenShift зображено на рисунку 1.8.

Однак, найбільшим недоліком цієї системи є її ціна. Цінова політика зображена на рисунку 1.9.

Навіть при використанні власних хмарних потужностей, ціна використання системи починається від 16000 доларів в рік, що становить близько 1300 доларів на місяць. Для маленьких та середніх компаній така ціна може бути зависокою. А при розміщенні на серверах Red Hat, чи при наявності потреби використовувати більш стабільнішу систему с декількома Master-серверами ціна зростає в декілька разів.

Також, як видно з рисунку 1.8, на даний момент підтримується близько 34 різних вбудованих сервісів, необхідних для побудови системи, програмних засобів та мов програмування.

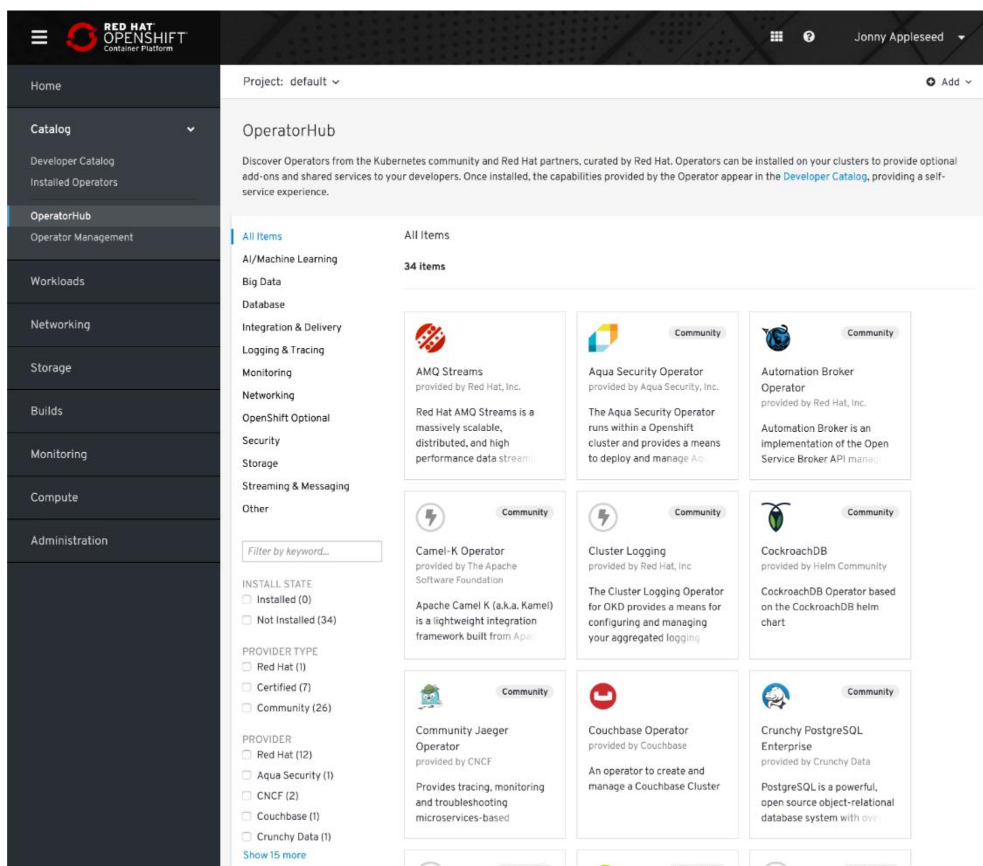

Рисунок 1.8 – Головне меню Red Hat OpenShift

	Standard Deploy in cloud provider accounts owned by Red Hat	Bring your own cloud Leverage your existing cloud provider discounts and settings
Single availability-zone cluster	Starts at $36,000/yr	Starts at $16,000/yr
Multiple availability-zone cluster	Starts at $81,000/yr	Starts at $36,000/yr

Рисунок 1.9 – Цінова політика Red Hat OpenShift

Проте, можливі ситуації, коли потрібної технології не буде доступно, або користувач не матиме доступу до налаштувань, які потрібно виконати саме для додатку користувача.

Heroku - це хмарна платформа, яка дозволяє компаніям створювати, доставляти, контролювати та масштабувати програми. Heroku робить процеси

розгортування, налаштування, масштабування, налаштування та управління програмами максимально простими та зрозумілими, щоб розробники могли зосередитися на написанні нового коду [26]. Головне вікно Heroku зображено на рисунку 1.10.

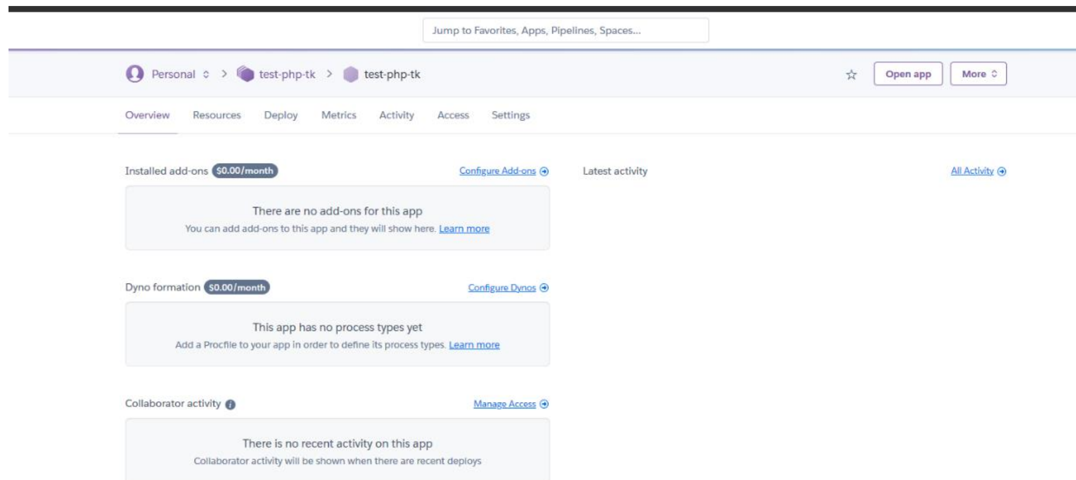

Рисунок 1.10 – Головне вікно Heroku

Ціни на різні тарифні плани значно гнучкіші, доступних тарифних планів більше, також можливо написати в підтримку та сформувати власний тарифний план. Також є безкоштовний тарифний план для ознайомлення з системою. Цінова політика відображена на рисунку 1.11.

Рисунок 1.11 – Цінова політика Heroku

Однак, у сервісі Heroku додатки, бази даних, та брокери повідомлень чітко розмежовані, тому об'єднання їх в одну систему може викликати деякі складнощі. Також, не всі тарифи в лінійці мають чітку логічну структуру. Наприклад, дорожчий тариф може не мати деяких функцій з попереднього по вартості тарифу, тому підбір оптимального тарифу по ціні та функціях може викликати деякі складнощі та зайняти деякий час.

1.4 Висновок

Під час роботи над розділом обґрунтування доцільності розробки інформаційної технології побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою проаналізовано предметну область архітектури серверних додатків та предметну область платформ для розміщення і підтримки серверних додатків з мікросервісною архітектурою. Було описано основні архітектури побудови серверних додатків. Визначено сутність мікросервісних додатків та основних підходів до їх розміщення. Для розміщення серверних додатків з мікросервісною архітектурою пропонується використовувати Docker-контейнери.

Виконано порівняння існуючих методів розміщення. Вибрано оптимальний метод для розміщення серверних додатків з мікросервісною архітектурою. А саме, побудову платформи за принципом PaaS.

В ході аналізу об'єкту проектування було визначено вимоги до інформаційної технології.

В результаті аналізу систем-аналогів, таких як Heroku та Red Hat OpenShift, було розглянуто вказані системи та визначено їх недоліки.

2 АНАЛІЗ ОСОБЛИВОСТЕЙ ТА ПРОГРАМНОГО ІНСТРУМЕНТАРІЮ РОЗРОБКИ ПІДСИСТЕМ ПЛАТФОРМИ ДЛЯ РОЗМІЩЕННЯ І ПІДТРИМКИ СЕРВЕРНИХ ДОДАТКІВ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

2.1 Аналіз особливостей платформ для розміщення і підтримки серверних додатків з мікросервісною архітектурою

При виборі платформи у вигляді PaaS-системи, для розміщення і підтримки серверних додатків зазвичай звертають увагу на такі параметри [27]:

- Можливість реплікації БД;
- Інструменти адміністрування;
- Готові інструменти для переносу додатків на платформу;
- Наявність інструментарію для бізнес-аналітики;
- Час, необхідний на розгортування додатку на платформі [24];
- Можливість зміни цілої платформи або її складових під потреби користувача.

Можливість реплікації БД дозволяє отримати відмовостійкий кластер для БД, що у випадку залежності від будь-якої СУБД є дуже критичним.

Інструменти адміністрування дозволяють керувати платформою та додатками і сервісами на платформі.

Готові інструменти для переносу додатків на платформу допомагають розробникам швидко та легко запустити копію додатку, яка розроблялась локально, а не на платформі.

Наявність інструментарію для бізнес-аналітики дають можливість експертам з аналітики оцінити параметри додатку, які впливають на його ціну, актуальність, зацікавленість аудиторії в тому чи іншому функціоналі та інші подібні параметри.

Час виходу на ринок, або Time-to-market (TTM) – це час від початку розробки продукту до його кінцевого виходу на ринок і подальшого

використання. Час виходу на ринок є невід’ємною частиною всього життєвого циклу продукту. Так як, в області створення додатків, в нього входять такі етапи як дослідження ринку, розробка стратегії, планування, написання коду, збірки, тестування, випуску версії, розгортування, експлуатація та моніторинг, то час, необхідний на розгортування додатку буде напряму впливати на час виходу на ринок, як для нових продуктів, так і для нових функцій вже існуючих продуктів [28]. Адже, чим більший показник TTM, тим довше розробники не отримують прибуток від продукту, що в свою чергу означає ріст вартості прибутку. Також, чим довше новий продукт не буде потрапляти на ринок тим більша ймовірність того, що хтось інший представить такий самий продукт на ринку [29].

Тому, зменшення часу, необхідного на розгортування додатку на платформі є одним з найважливіших показників.

Можливість зміни платформи дозволяє гнучко налаштовувати систему в процесі її використання, адже з часом можуть з’являтись нові, кращі технології, вимоги зі сторони користувача змінюватись, а також можуть бути виявлені вразливості будь-якої частини системи.

2.2 Аналіз програмного інструментарію, що використовується для реалізації підсистем платформи

2.2.1 Аналіз програмного інструментарію для контейнерування

Контейнеризація - це легка заміна віртуалізації і ізоляція ресурсів на рівні операційної системи, яка дозволяє запускати додаток і необхідний йому мінімум системних бібліотек в повністю стандартизованому контейнері. Контейнер не залежить від ресурсів або архітектури хосту, на якому він працює [30].

Найвідомішим засобом для створення віртуальних контейнерів є Docker [31]. Станом на 2020 рік в загальний репозиторій Docker було виконується близько 8 мільярдів завантажень образів контейнерів на місяць, що на 2.5 мільярдів більше ніж в 2019 році, кількість загальних користувачів досягла 5

мільйонів, а кількість встановлення лише на персональні комп'ютери становила близько 2.4 мільярдів [32].

Найбільш відомим конкурентом Docker наразі є Podman [33]. Однак, нажаль, недоліки Podman-а не дозволяють його використовувати, а саме:

- Основа Podman – ядро Docker-у. Тому не зовсім зрозуміла необхідність використання іншого інструменту;
- Також, для розгортування використовуються образи від Docker;
- Наявна проблема з роботою з диском. Через особливості архітектури та застосованих драйверів у системах з використанням Podman часто наявні проблеми з навантаженням на диск [34].
- Станом на 2020 рік Podman все ще знаходиться в стані активної розробки і не може гарантувати високої стабільності [35].

При цьому, Docker працює і стабільно розвивається з 2013 року [36], а багато світових компаній, таких як Adobe, AT&T, PayPal, використовують Docker для своїх повсякденних задач [22].

Тому, як засіб для контейнерування був вибраний саме Docker. На рисунку 2.1 зображено загальну структуру Docker-системи.

За допомогою клієнта, який представляє собою Docker CLI, користувач передає відповідні команди до Docker deamon-у, який виконує управління контейнерами. При цьому Docker deamon може знаходитись як на локальній, так і на віддаленій системі. При необхідності запуску нового контейнеру, з образу якого ще немає на локальній системі, Docker deamon виконує запит в регістр (Registry), звідки скачує необхідний образ на сервер, де встановлений сам Docker deamon і після цього виконує розгортування контейнеру зі скачаного образу.

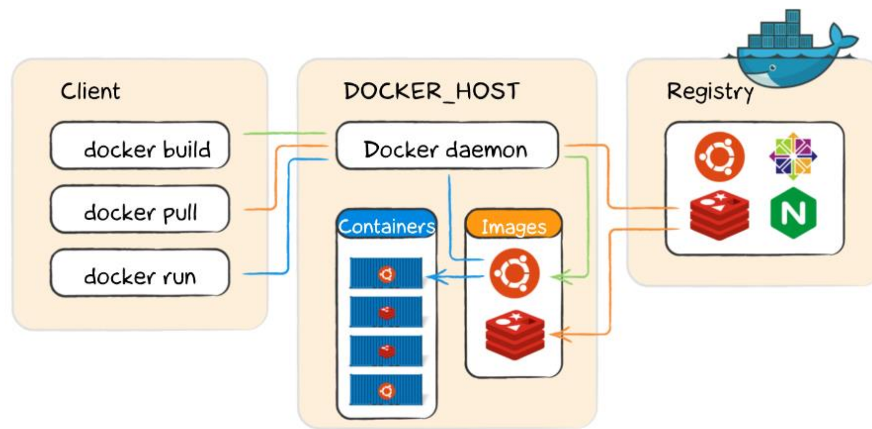

Рисунок 2.1 – Загальна структура Docker-системи

2.2.2 Аналіз програмного інструментарію для реалізації підсистеми управління контейнерами

Серед популярних програмних засобів для реалізації підсистеми управління (або оркестрації) контейнерами можна назвати такі готові програмні засоби:

- Docker Swarm
- Mesosphere
- Kubernetes

Docker випустив власний продукт для оркестрації - Docker Swarm. Основна перевага кластера Docker Swarm полягає в тому, що він входить до складу установки Docker і використовує стандартні API Docker. Таким чином, навчання не буде таким складним і почати освоювати ці продукти легше, ніж інші. Проте Docker Swarm відстає від Kubernetes та інших подібних систем з точки зору можливостей і готовності. Крім того, Docker Swarm не відрізняється бездоганною репутацією і в тому випадку, коли мова йде про якісні технології і безпеку. Організації і розробники, зацікавлені в стабільності своїх систем, можуть відмовитися від Docker Swarm. Розробники Docker усвідомлюють проблему і роблять кроки для її усунення. Але недавно і розробники Docker визнали високе місце Kubernetes як платформи для оркестрації контейнерів.

Mesosphere - компанія, що стоїть за відкритим вихідним кодом Apache Mesos, а продукт DC/OS - це платформа, що відповідає за управління контейнерами і великими даними в хмарі. Технологія стабільна, і Mesosphere розвиває її, але у них немає ресурсів і імпульсу, якими володіє Kubernetes. Крім того, Mesosphere також визнала, що вони не можуть перемогти Kubernetes і вирішили приєднатися до неї. В DC/OS 1.11 ви отримуєте Kubernetes як сервіс.

Kubernetes - це система з відкритим вихідним кодом, яка автоматизує розгортування і масштабування упакованих в контейнер додатків, а також управління ними. Якщо ви запускаєте багато контейнерів або хочете керувати ними автоматично, ця система вам знадобиться. Автор Kubernetes - компанія Google. Користувачі стікаються до Kubernetes в міру того, як зростає поінформованість про неї, ЗМІ визнають, що вона займає лідируючу позицію, екосистема розвивається, великі корпорації і компанії слідом за Google активно підтримують її, а багато хто оцінює і запускають у виробництво. Користувачі стікаються до Kubernetes в міру того, як зростає поінформованість про неї, ЗМІ визнають, що вона займає лідируючу позицію, екосистема розвивається, великі корпорації і компанії слідом за Google активно підтримують її, а багато хто оцінює і запускають у виробництво. Всі основні постачальники хмарних послуг безпосередньо підтримують Kubernetes. Kubernetes активно розвивається, її спільнота збільшується [37]. На рисунку 2.2 зображено кількість запитів по тегу Kubernetes на популярному сайті для розробників та системних адміністраторів, Stack Overflow.

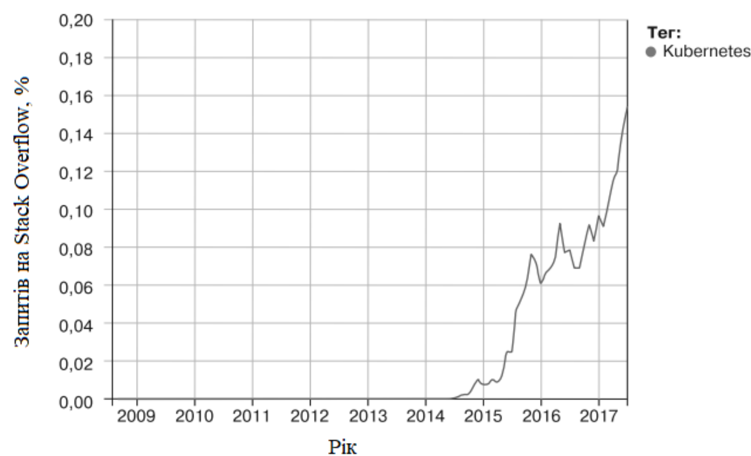

Рисунок 2.2 – Кількість запитів по тегу Kubernetes на сайті Stack Overflow

Kubernetes з самого першого випуску домоглася великих успіхів в області модулювання. Офіційна документація Kubernetes стає все краще і краще. Онлайн підручники відмінно підходять для початку роботи з цією платформою. Збільшується кількість книг, присвячених різним аспектам роботи з Kubernetes. Компанії, які розробляють програмне забезпечення та доповнення для контейнерної оркестрації, зосереджені на Kubernetes, а не на створенні продуктів, що підтримують кілька рішень оркестрації. Kubernetes оновлюється досить регулярно. На рисунку 2.3 зображено статистику по використанню систем управління контейнерами.

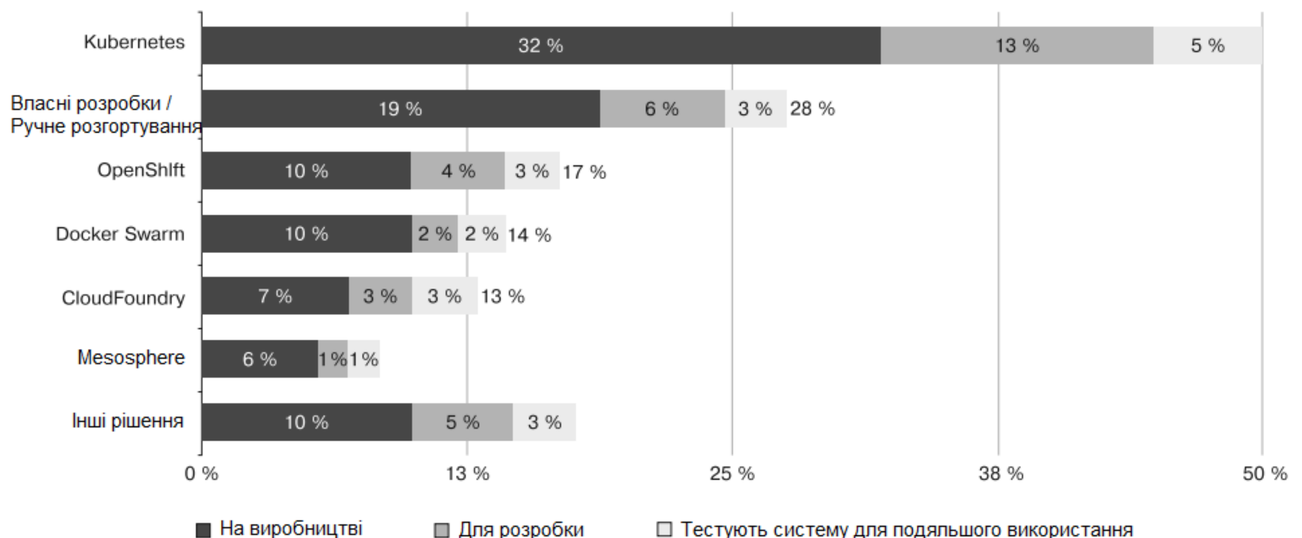

Рисунок 2.3 – Статистика по використанню систем управління контейнерами

2.2.3 Аналіз програмного інструментарію для реалізації підсистеми автоматизування встановлення платформи

При розміщенні будь-яких додатків на серверах кожен з цих серверів вимагає виконання початкових налаштувань, регулярного оновлення та встановлення нового ПЗ. Звичайно, для вирішення цих завдань можна скористатися найпростішим способом - підключитися до кожного серверу по протоколу ssh і виконати необхідні зміни. При всій своїй простоті цей спосіб пов'язаний з деякими труднощами: він надзвичайно трудомісткий, а на

виконання одноманітних операцій йде дуже багато часу, особливо, коли серверів більше десятка.

Щоб спростити процеси налаштування і конфігурації серверів, можна також писати shell-скрипти. Але і цей спосіб навряд чи можна назвати досконалим. Скрипти потрібно постійно змінювати, підлаштовуючи їх під кожну нову задачу. При їх написанні необхідно враховувати відмінність операційних систем і версій. Не будемо забувати і про те, що налагодження скриптів забирає багато зусиль і забирає чимало часу.

Оптимальним варіантом вирішення описаних проблем є впровадження системи віддаленого управління конфігурацією. У таких системах досить лише описати потрібний стан керованого вузла. Система повинна сама визначити, що потрібно зробити для досягнення цього стану, і здійснить усі необхідні дії [38]. Самі ж такі системи дозволяють застосувати принцип IaaS – інфраструктура як код. При застосуванні принципу IaaS адміністратору необхідно лише написати код, який виконає всі необхідні дії автоматично.

Популярними серед таких систем є Puppet [39] та Ansible [40]. Обидві системи є відкритими та безкоштовної, в кожної з них є платні версії, які гарантують підтримку від розробників. Однак, в той же час Ansible значно відрізняється від Puppet в кращу сторону.

Puppet використовує в своїй роботі агенти і базується на клієнт-серверній архітектурі. Тобто, перед його використанням потрібно на цільовий сервер виконати встановлення Puppet-агенту, який буде виконувати звернення до Puppet-серверу для перевірки поточного потрібного стану системи. Керування конфігураціями в такому випадку виглядає так [41]:

1) Адміністратор вносить зміни в сценарій управління конфігураціями;
2) Адміністратор передає зміни на Puppet-сервер;
3) Puppet-агент періодично вмикається за таймером;
4) Puppet-агент виконує підключення до Puppet-серверу;
5) Puppet-агент читає нові сценарії управління конфігураціями;

6) Puppet-агент виконує отримані сценарії локально, оновлюючи стан серверу.

На відміну від Ansible, який працює по протоколу SSH з використанням Python, який по замовчуванню встановлений на всіх Linux-системах. В такому випадку, процес керування конфігураціями виглядає так [41]:

1) Адміністратор вносить зміни в сценарій;
2) Адміністратор запускає новий сценарій;
3) Ansible підключається до потрібних серверів і запускає відповідні модулі, оновлюючи стан системи серверів.

Схематичне зображення процесу роботи Ansible наведено на рисунку 2.4.

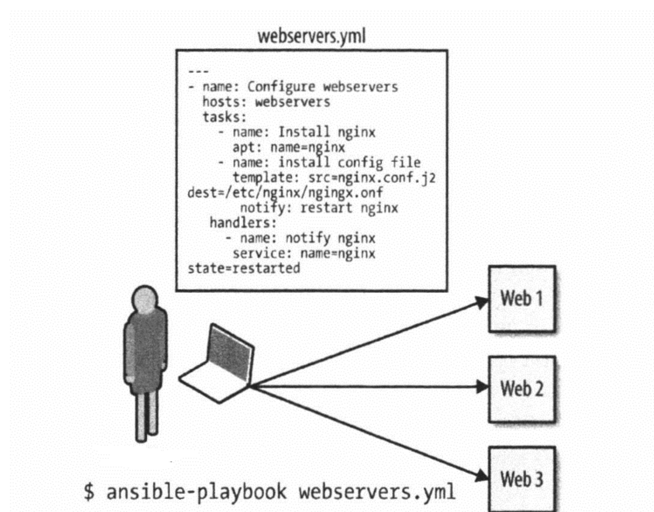

Рисунок 2.4 – Схематичне зображення процесу роботи Ansible

Також, однією з переваг Ansible є простота синтаксису. В Ansible всі сценарії пишуться з використанням YAML, тоді як у Puppet синтаксис маніфестів дуже схожий на синтаксис С-подібних мов програмування, так як Puppet написано на мові Ruby і тому синтаксис прив’язано саме до неї. Приклад обох конфігураційних сценаріїв наведено на рисунку 2.5.

```
class mysql::client (
    $bindings_enable = $mysql::params::bindings_enable,
    $install_options = undef,
    $package_ensure = $mysql::params::client_package_ensure,
    $package_manage = $mysql::params::client_package_manage,
    $package_name = $mysql::params::client_package_name,
)  inherits mysql::params {

    include '::mysql::client::install'

    if $bindings_enable {
        class { 'mysql::bindings':
            java_enable => true,
            perl_enable => true,
            php_enable => true,
            python_enable => true,
            ruby_enable => true,
        }
    }

# Anchor pattern workaround to avoid resources of mysql::client::install to
# "float off" outside mysql::client
anchor { 'mysql::client::start': }
-> Class['mysql::client::install']
-> anchor { 'mysql::client::end': }
}
```

Puppet

```
- name: Add Kube GPG apt Key
  apt_key:
    url: https://packages.cloud.google.com/apt/doc/apt-key.gpg
    state: present

- name: Add Kube repo
  apt_repository:
    repo: deb https://apt.kubernetes.io/ kubernetes-xenial main
    state: present

- name: Install Kube tools
  apt:
    update_cache: yes
    pkg:
    - kubelet
    - kubeadm
    - kubectl

- name: Mark Kube tolls hold (kubelet)
  dpkg_selections:
    name: kubelet
    selection: hold

- name: Mark Kube tolls hold (kubeadm)
  dpkg_selections:
    name: kubeadm
    selection: hold

- name: Mark Kube tolls hold (kubectl)
  dpkg_selections:
    name: kubectl
    selection: hold
```

Ansible

Рисунок 2.5 – Приклад конфігураційних сценаріїв для Puppet та Ansible

Також, Ansible, на відміну від Puppet має велику кількість вбудованих модулів та наявна можливість завжди створювати свої власні модулі [41].

Крім того, за статистикою, яка представлена на рисунку 2.6, Ansible більш розповсюджений, ніж Puppet.

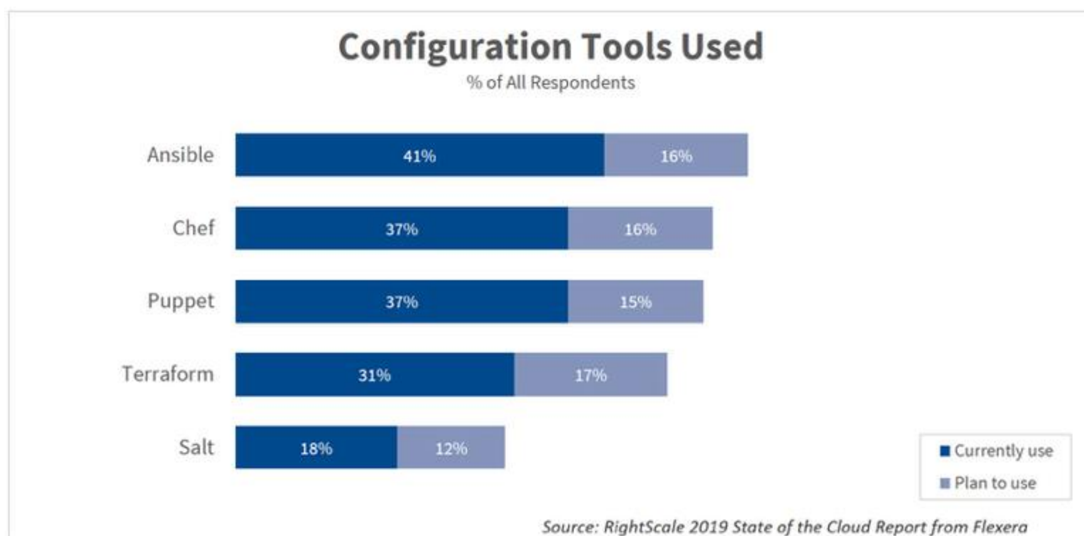

Рисунок 2.6 – Розповсюдженість різних систем віддаленого управління конфігурацією

2.2.4 Аналіз програмного інструментарію для реалізації підсистеми автоматизованого розгортування клієнтських додатків

Автоматизоване розгортування додатків допомагає спростити процес завантаження, запуску, тестування та оновлення додатків на сервері.

При автоматизації подібних задач використовують принцип CI/CD (Continuous Integration / Continuous Delivery) – безперервної інтеграції та безперервної доставки.

Визначення CI досить просте. Це практика розробки, яка вимагає від розробника інтеграції коду в центральне спільне сховище. Кожного разу коли розробник завантажує нову версію коду, він інтегрується з іншим кодом і перевіряється під час виконання тестів. CI запускається кожного разу, коли ми передаємо код до централізованого сховища. Це означає що кожного разу, коли ми щось змінюємо, наприклад, мітку на HTML-сторінці, або змінну в коді, ми перевіряємо весь додаток, оскільки тестуємо додаток при кожному окремому завантаженні коду. Розробники та адміністратори можуть знаходити помилки швидше та легше у збірці та виправляти їх. Якщо ми розглянемо типовий проект, то побачимо, що різні люди беруть участь у різних сферах. Розробник створює функцію і при необхідності змінює інші скрипти, відповідно до створеної функції. Адміністратор бази даних (DBA) встановлює скрипти і консультує розробника, коли база даних вже використовується. Починаючи з цього, розробник продовжує розробку ПЗ. В кінці розробки програмне забезпечення інтегрується та тестується все разом [42].

CD - це практика програмного забезпечення, яка використовується для випуску програмного забезпечення протягом короткого циклу. Це означає, що з кожним завантаженням нового коду ми створюємо нову збірку всього програмного забезпечення [42].

Серед популярних інструментів для використання підходу CI/CD в системі можна назвати TeamCity та Jenkins.

TeamCity - це інструмент CI/CD на основі Java, створений JetBrains, виробником інших корисних інструментів, таких як PyCharm, IntelliJ Idea, RubyMine, ReShaper та інших. Невеликі команди можуть безкоштовно користуватися TeamCity. Його можна встановити на сервери Windows і Linux. Він підтримує покрокову збірку при виконанні потрібних умов, може запустити агенти збірки у кластері Kubernetes. Також наявна можливість інтегруватися з такими популярними інструментами управління проектами, як Azure DevOps та Jira [43].

Jenkins, мабуть, найвідоміше програмне забезпечення, пов'язане з безперервною інтеграцією та доставкою (CI/CD). Jenkins - це програмне забезпечення з відкритим кодом, написане на Java, щоб допомогти автоматизувати всі "комп’ютерні" процеси, що відбуваються під час розробки програмного забезпечення. Jenkins працює майже з будь-якими типами сховищ, такими як CVS, Git, Subversion та іншими. Можна дуже автоматизувати всі процеси, пов'язані з CI/CD легко. Найважливіша перевага в Jenkins - це плагіни. Можна знайти плагін майже для будь-якої необхідності. За допомогою Jenkins можна використовувати тригер для запуску збірки після завантаження коду в репозиторії, можемо автоматизувати будь-яке окреме завдання, пов'язане з цим процесом. Це означає що кожного разу, завантажується нова версія коду, автоматично виконуються всі завдання, необхідні для збірки програмного забезпечення. Jenkins пропонує дуже хорошу підтримку для створення конвеєрів CI та CD [42].

Для вибору одного з них необхідно виконати порівняння частини характеристик.

І TeamCity, і Jenkins мають підтримку спільноти, доступну на їх веб-сайті, що дозволяє користувачам їх програмного забезпечення обмінюватися інформацією та колективно вирішувати проблеми, а також надає базу знань, яку можна переглядати при пошуку конкретних відповідей.
В даному аспекті, Jenkins має невелику перевагу тому, що вони існують довше і

як проект з відкритим кодом, користувачі Jenkins більше покладаються на спільноту для усунення несправностей.

TeamCity пропонує дві версії свого продукту CI. TeamCity Professional безкоштовно для 100 конфігурацій збірки та трьох агентів збірки. Додаткові ліцензії агенту збірки можна придбати за $299 і вона включає одного додаткового агенту збірки та десять додаткових конфігурацій збірки. TeamCity Enterprise дозволяє використовувати необмежену кількість конфігурацій збірки та ліцензій від $ 1999 для 3-х агентів збірки, масштабуючи до 100 агентів за $ 21,999. Однак, Jenkins є системою з відкритим кодом, під ліцензією MIT, і як може бути завантажений та використаний безкоштовно.

І TeamCity, і Jenkins пропонують RESTful API для розробників. API Jenkins поставляється в трьох варіантах: XML, JSON та Python. Jenkins має більшу розширюваність, оскільки майже кожна ланка системи може бути налаштована за потребою. TeamCity має набагато менше документації по API на своєму веб-сайті.

Плагіни - це велика перевага Jenkins, який пропонує сотні безкоштовних плагінів, від управління вихідним кодом, створення інструментів до спеціальних інструментів розробки. Ця, вже існуюча база даних корисних плагінів значно полегшує введення Jenkins у робоче середовище, позбавляючи потреби у дорогих власних або сторонніх продуктах. TeamCity також має понад 380 безкоштовних, розроблених натовпом плагінів, які вони пропонують як є, за межами підтримуваного комерційного продукту, але їх загальна кількість так функціональність замала. Крім того, архітектура Jenkins дозволяє створювати власні плагіни [44].

2.2.5 Аналіз програмного інструментарію для реалізації підсистеми логування та збору і аналізу статистичних даних

За останні роки Інтернет, мобільні додатки, соціальні мережі генерують величезну кількість даних. Ці нові джерела інформації неможливо обробити за

допомогою традиційних технологій зберігання даних, стандартних реляційних баз даних. Для розробників додатків або бізнес-аналітиків важливо зробити так, щоб всі вимоги з пошуку інформації та аналізу додатку були задоволені.

За останні кілька років з'явилися різні системи для зберігання і обробки великих масивів даних. Серед них можна виділити аналітичні системи на зразок Elastic Stack або Graylog.

Elastic Stack - велика екосистема компонентів, які служать для пошуку і обробки даних. Основні компоненти Elastic Stack - це Kibana, Logstash, Beats, X-Pack і Elasticsearch. Ядром Elastic Stack виступає пошукова система Elasticsearch, яка надає можливості для зберігання, пошуку та обробки даних. Утиліта Kibana є відмінним засобом візуалізації і призначеним для користувача інтерфейсом для Elastic Stack. Компоненти Logstash і Beats дозволяють передавати дані в Elastic Stack. X-Pack надає потужний функціонал: можна налаштовувати моніторинг, додавати різні повідомлення, встановлювати параметри безпеки для підготовки системи до експлуатації [45]. Загальна схема роботи Elastic Stack зображена на рисунку 2.7.

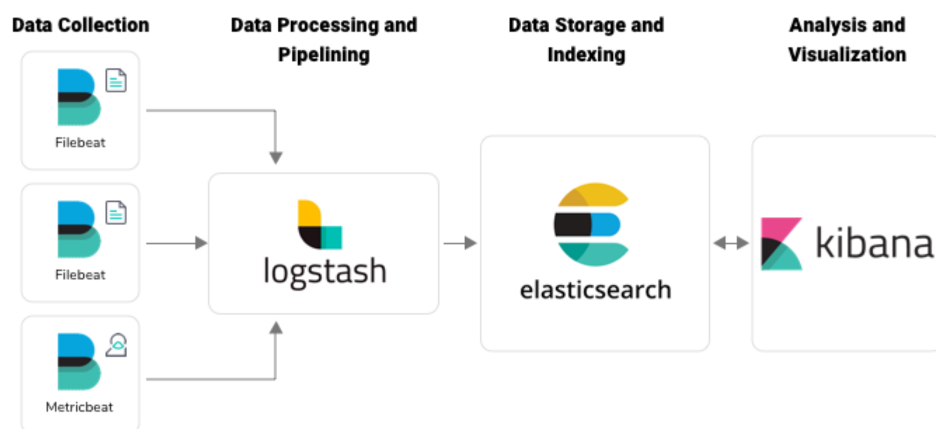

Рисунок 2.7 – Загальна схема роботи Elastic Stack

Graylog – це ще один потужний інструмент для управління журналами, який надає безліч варіантів аналізу вхідних журналів з різних серверів. Принцип роботи Graylog дуже схожий на ELK. На додаток до самого сервера Graylog, який

складається з додатка та сервера веб-інтерфейсу, також потрібно мати MongoDB та Elasticsearch, щоб зробити весь стек повністю працездатним [46].

Однак, її використання пов’язане з деякими проблемами [46]:

- Graylog не може читати з файлів syslog, тому вам потрібно надсилати свої повідомлення безпосередньо в Graylog, а частина сервісів та ПЗ може не мати такої функції;
- З точки зору управління, інтерфейс не дуже доброзичливий до нових користувачів;
- Функціональність звітування досить погана;
- Через використання MongoDB система може працювати повільно при надвеликих масивах інформації;
- Досить мало плагінів, які дозволяють розширити функціональність [47];
- Відсутність стандартної функції по створенню кластеру для забезпечення відмовостійкості системи.

2.2.6 Аналіз програмного інструментарію для реалізації підсистеми моніторингу

Сьогодні моніторинг значно відрізняється, від того, яким він був лише кілька років тому, і навіть більше, ніж був 10 років тому. З широкою популярністю хмарної інфраструктури з’явилися нові проблеми моніторингу, а також створення нових способів вирішення старих проблем.

Зростання популярності мікросервісів також позначився на тому, які підходи так інструменти використовуються при моніторингу. Оскільки вже не існує тільки монолітного сервера додатків, постає проблема в моніторингу взаємодії десятків, а то й сотень невеликих сервісів різних додатків, які постійно спілкуються. Типовою логічною одиницею в архітектурі мікросервісів є сервіс ,

що може існувати лише години або навіть хвилини, що спричинило хаос у віковічній тактиці та інструментах моніторингу, на які колись покладались.

Однак, деякі частини всього процесу залишаються незмінними. Все ще необхідно турбуватись про продуктивність веб-серверу. Все ще занепокоєння викликає питання вільного місця на серверах. Продуктивність сервера баз даних все ще грає важливу роль в роботі всіх систем, що його використовують. Хоча деякі проблеми, які актуальні сьогодні, схожі на (або такі ж, як і) проблеми, які були у 10 років тому, а доступні інструменти та методології значно вдосконалені.

Моніторинг - це спостереження та перевірка поведінки та результатів роботи системи та її компонентів у часі. Це визначення є широким, але актуальним – моніторинг включає в себе багато елементів, це: метрики, журналювання, оповіщення, управління інцидентами, статистика та багато, багато іншого [48].

Серед популярних систем моніторингу можна виділити Munin, Zabbix та Prometheus.

Munin - засіб моніторингу що слідкує за серверами в мережі та записує зібрані данні в БД. Він представляє всю інформацію на графіках через веб-інтерфейс [49]. Інтерфейс Munin зображено на рисунку 2.8.

Його акцент робиться на можливостях plug and play. Після завершення встановлення велика кількість плагінів для моніторингу буде працювати без жодних зусиль.

Використовуючи Munin, ви можете легко відстежувати продуктивність своїх серверів, мереж, мереж SAN, додатків та майже будь-яких метрик на сервері [49].

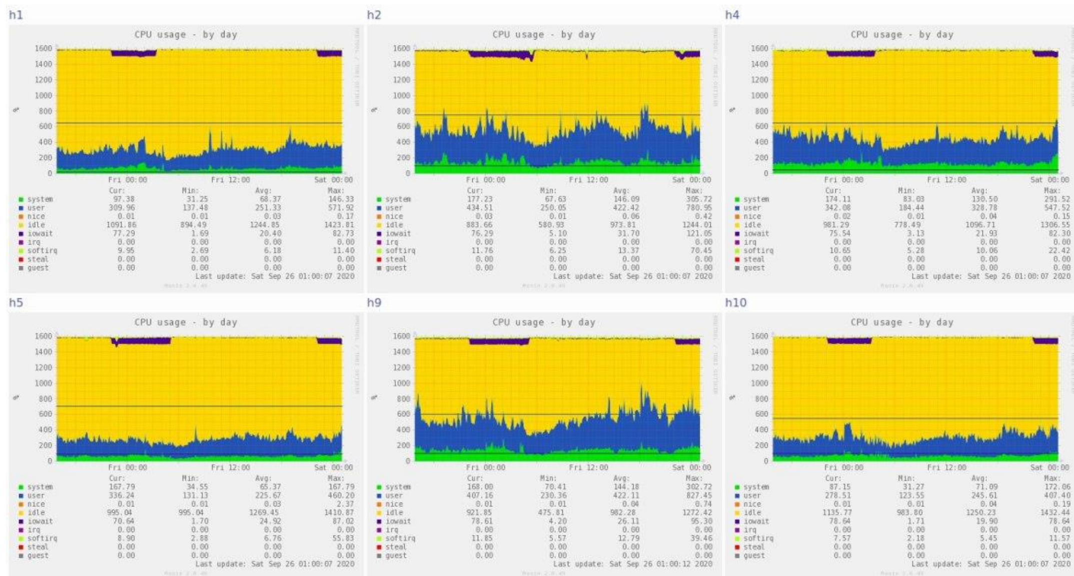

Рисунок 2.8 – Інтерфейс системи моніторингу Munin

Однак, інтерфейс Munin виглядає застаріло. А також, відсутність інтерактивності ускладнює роботу з ним, особливо коли відслідковується одна метрика великої кількості вузлів. Для прикладу, на рисунку 2.9 зображено графік навантаження на диск виділеного серверу, яке створювали віртуальні сервери на ньому.

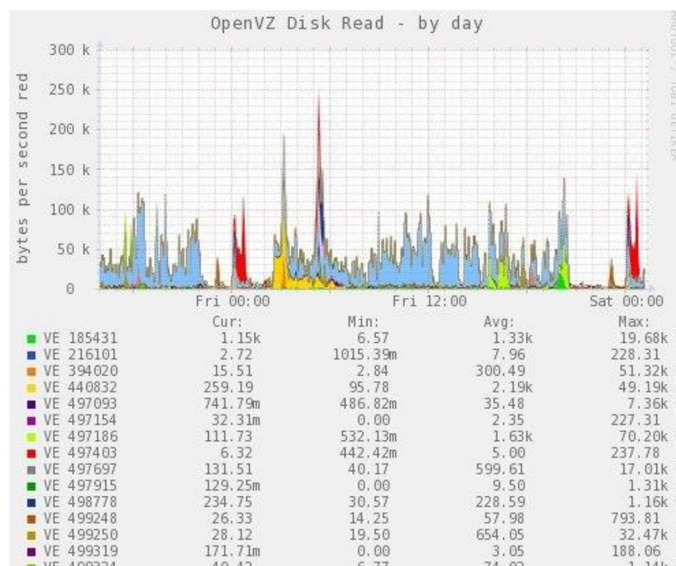

Рисунок 2.9 – Приклад складного для розуміння графіку Munin

Саме через відсутність інтерактивності та велику кількість кольорів і великий масштаб, який неможливо змінити, дуже складно зрозуміти більшу частину графіку.

Zabbix - це універсальний інструмент моніторингу, здатний відслідковувати динаміку роботи серверів та мережевого обладнання, дозволяє швидко реагувати на позаштатні ситуації та попереджати можливі проблеми при навантаженні. Система моніторингу Zabbix може збирати статистичні дані у вказаному робочому середовищі [50]. Головний екран Zabbix зображено на рисунку 2.10.

Рисунок 2.10 – Головний екран Zabbix

Однак, нажаль Zabbix також не може надати максимальної інтерактивності графіків. Їх масштабування стало значно простішим, зручнішим і кориснішим, у порівнянні з Munin, однак цього може бути недостатньо.

Prometheus - це система моніторингу на основі метрик із відкритим кодом. Вона має просту, але потужну модель даних і мову запитів, що дозволяє аналізувати, як програми та інфраструктури працюють. Вона не намагається вирішити проблеми поза простору метрик, залишаючи їх іншим більш відповідним інструментам.

Prometheus в основному написаний на Go та ліцензований за ліцензією Apache 2.0. Станом на 2018 рік десятки тисяч організацій використовують Prometheus у виробництві.

Для створення власного коду та модулів наявні клієнтські бібліотеки у всіх популярних мовах, включаючи Go, Java, C#/.Net, Python, Ruby, Node.js, Erlang,

Rust. Таке програмне забезпечення, як Kubernetes та Docker, постачається з вже готовими клієнтськими бібліотеками Prometheus. Для стороннього програмного забезпечення, яке відображає показники у інших форматах, які не підтримує Prometheus , доступні сотні інтеграційних інструментів. Простий текстовий формат полегшує імпорт метрик в Prometheus. Інший системи моніторингу, як з відкритим кодом, так і комерційні, додали підтримку цього формату.

Модель даних ідентифікує кожен часовий ряд не просто з іменем, а й з невпорядкованим набором пар ключ-значення, що називається мітками. Вони можуть бути візуалізовані за допомогою спеціальних систем, наприклад Grafana.

Один сервер Prometheus може приймати мільйони метрик в секунду. Всі компоненти Prometheus можна запускати в контейнерах, і це дозволяє уникати зайвих дій, що ускладнювали б їх встановлення. Prometheus розроблений для того, щоб бути інтегрованим до вашої вже створеної інфраструктури та а не бути самою платформою управління [51]. Типова структура моніторингу на основі Prometheus та Grafana зображено на рисунку 2.11.

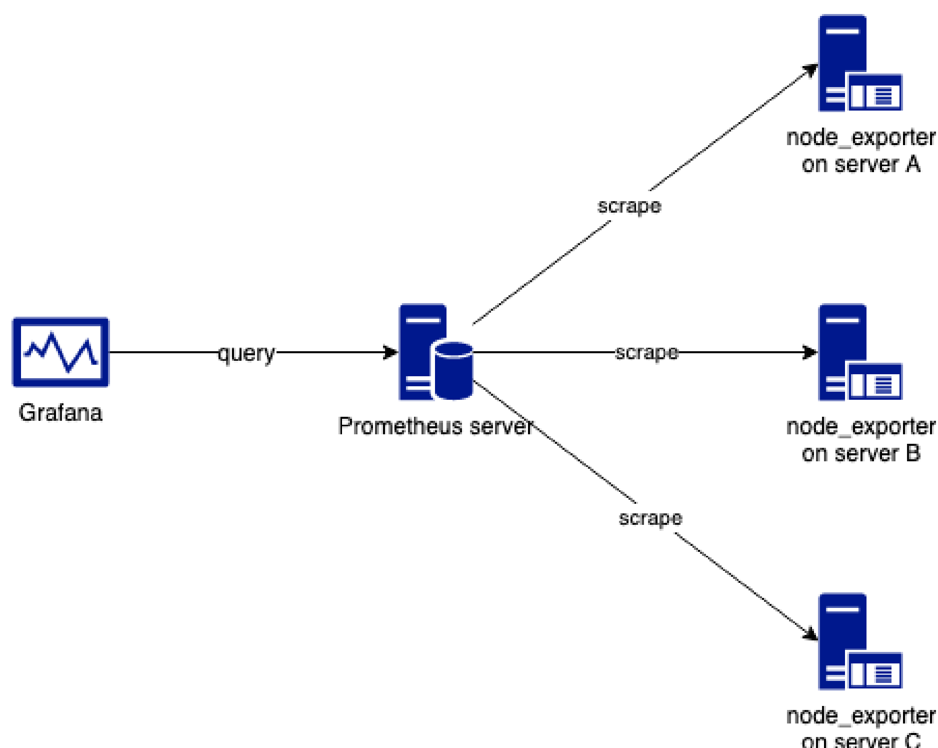

Рисунок 2.11 – Типова структура моніторингу на основі Prometheus та Grafana

2.2.7 Аналіз сервісів для реалізації підсистеми збереження конфігураційних файлів

Для зберігання загальних конфігураційних файлів, шаблонів конфігураційних файлів та конфігураційних сценаріїв необхідно мати сховище. Простого файл-сховища буде недостатньо, так як необхідно мати декілька версій конфігураційних файлів, щоб в разі потреби мати змогу виконати відновлення старішої версії файлів. А при використанні звичайного файл сховища для реалізації подібного функціоналу потрібно створювати якісь додаткові програмні засоби.

Тому було прийнято рішення, використовувати вже готові сервіси, які дозволяють реалізувати подібні функції. Такі сервіси базуються на системах контролю версій.

Система управління версіями, або контролю версій (VCS), записує історію зміни файлу або набору файлів, щоб в майбутньому була можливість повернутися до конкретної версії. Вона дозволяє повертати в попередній стан як окремих файлів, так і цілого проекту, порівнювати зроблені зміни, дивитися, хто і коли останнім редагував файл, який є джерелом проблеми, і багато іншого. Наявність VCS в загальному випадку означає, що при пошкодженні, збої в системі або втраті файлів ви можете легко повернутися в робочий стан. На даний момент широко використовуються розподілені системи контролю версій. У розподілених системах контролю версій (наприклад, Git, Mercurial, Bazaar або Darcs) клієнти не просто вивантажують останні знімки файлів, вони створюють повну дзеркальну копію сховища. Відповідно в разі виходу з ладу одного з серверів його працездатність можна відновити, скопіювавши один з клієнтських репозиторіїв [52]. Загальна структура такої системи зображена на рисунку 2.12.

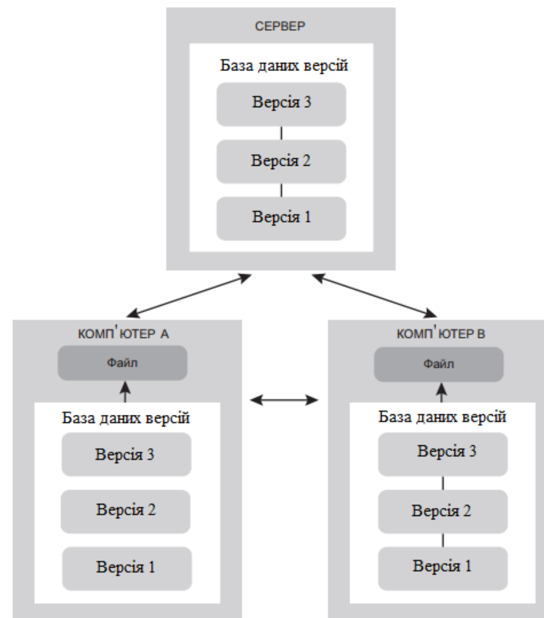

Рисунок 2.12 – Загальна структура розподіленої системи контролю версій

Кожне таке вивантаження супроводжується повним резервним копіюванням всіх даних. Більш того, багато які з цих систем мають кілька віддалених репозиторіїв, що дозволяє різним робочим групам співпрацювати в рамках одного проекту. Допустима настройка різних типів робочих процесів.

Найпопулярнішою основою для таких систем на даний час є Git. Головною відмінністю Git від будь-якої іншої системи контролю версій (в тому числі Subversion і їй подібних) є сприйняття даних. Більшість систем зберігає інформацію у вигляді списку змін, пов'язаних з файлами. Ці системи (CVS, Subversion, Perforce, Bazaar та інші) розглядають збережені дані як набір файлів і змін, які вносилися в ці файли протягом їх життя. Такий підхід є застарілим [52]. Схема збереження даних в таких системах наведено на рисунку 2.13.

Рисунок 2.13 – Схема збереження в застарілих системах контролю версій

Система Git не сприймає і не зберігає файли подібним чином. В її сприйнятті дані представляють собою набір знімків стану мініатюрної файлової системи. Кожен раз, коли ви створюєте нову версію або зберігаєте стан проекту в Git, по суті, робиться знімок всіх файлів в конкретний момент часу і зберігається посилання на цей знімок. Для підвищення продуктивності замість файлів, які не зазнали змін, зберігається лише посилання на їх раніше збережені версії, що зображено на рисунку 2.14. Git сприймає дані швидше як потік знімків стану (stream of snapshots) [52].

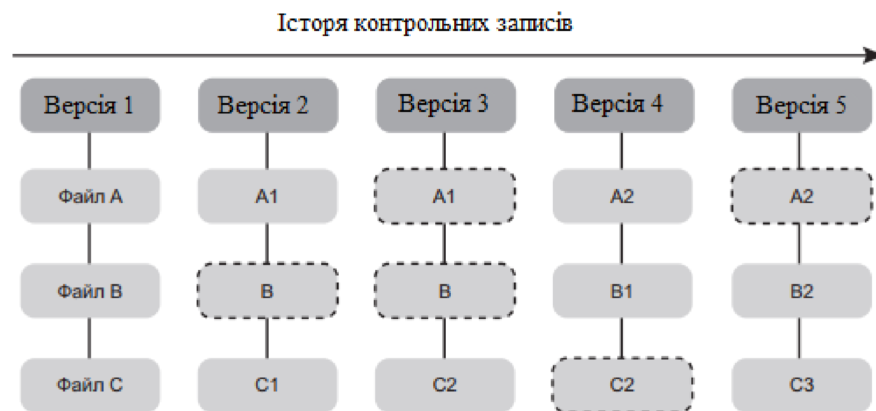

Рисунок 2.14 – Схема збереження в Git

Серед найвідоміших систем контролю версій можна виокремити GitHub та Bitbucket. Проте, Bitbucket краще ніж GitHub по таким причинам [53]:

1) Bitbucket є більш гнучким, ніж GitHub - хоча GitHub має безліч функцій і дозволяє створювати власні робочі процеси, BitBucket має вбудовану більшу гнучкість. BitBucket також може імпортувати з Git, CodePlex, Google Code, HG, SourceForge та SVN. А GitHub обмежений Git, SVN, HG і TFS.;
2) Bitbucket надає необмежену кількість приватних репозиторіїв;
3) Bitbucket має кращі ціни на приватні роботи - цінова політика обох сервісів наведена на рисунку 2.15
4) Безперервна інтеграція / доставка вбудована за замовчуванням;

5) Розумніший семантичний пошук;

6) Bitbucket має потужну інтеграцію з усіма процесами Atlassian - серед них такі інструменти та сервіси як Jira, Trello, Confluence та багато інших.

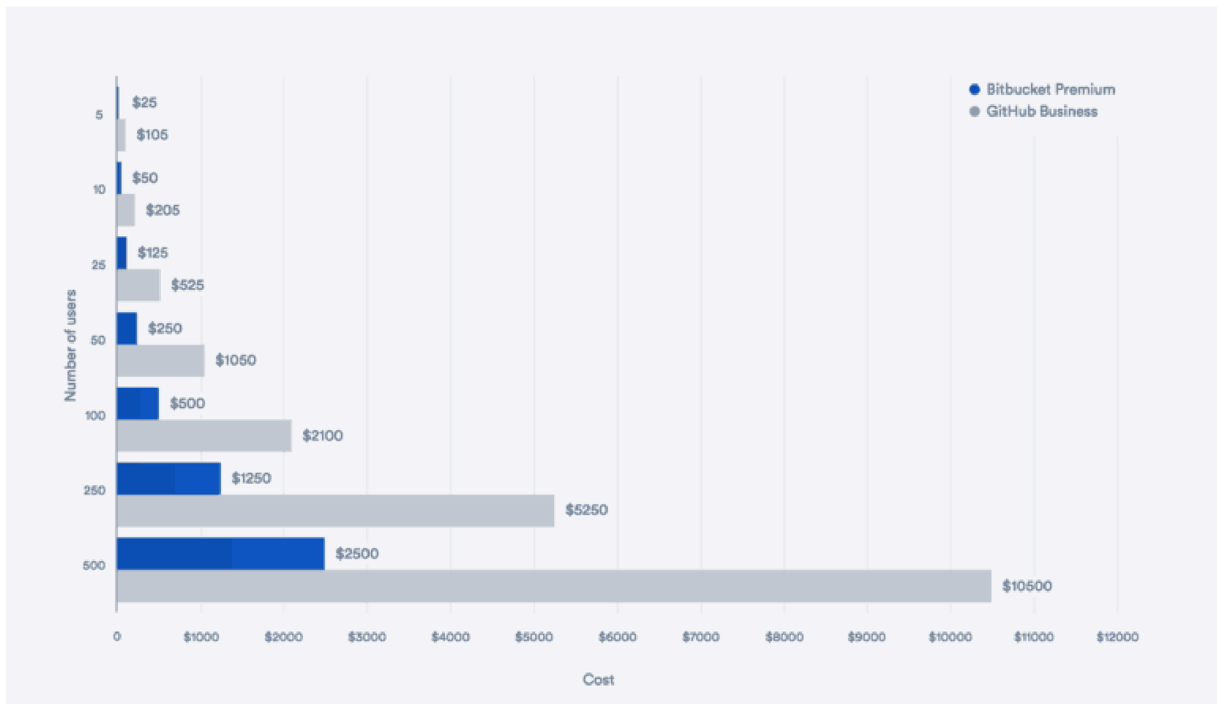

Рисунок 2.15 – Порівняння цінових політик BitBucket та GitHub

2.2.8 Аналіз програмного інструментарію необхідного для функціонування серверу для статичних файлів

Частина додатків може використовувати статичні файли, такі як картинки, відео, анімації та багато інших. Вони напряму не впливають на роботу додатку. А включення цих файлі в образ додатку або сервісу може значно збільшити об’єм вже зібраного образу та збільшити час, який буде затрачено на збірку самого образу додатку.

Тому, необхідно розмежовувати статичні файли та код. Для статичних файлів необхідно мати окремий сервер, де вони будуть зберігатись і будуть доступними для додатків.

Вимоги до такого серверу наступні:

– Доступ до статичних файлів по протоколу HTTP;

- Розмежування HTTP-доступу до статичних файлів по різним доменам для різних додатків;
- Розмежування користувачів;
- Наявність FTP-доступу для завантаження файлів;
- Розмежування FTP-доступу для файлів різних додатків.

Для надання доступу до статичних файлів по протоколу HTTP найкраще підходить веб-сервер Nginx. Він був створений саме для швидкої обробки та видачі статичного контенту. За результатами досліджень Nginx найкраще справляється з роботою з статичними файлами, так як був створений саме для цього [54], та є одним з найлегших веб-серверів [55], що дозволить мінімізувати навантаження на систему.

Для організації FTP-доступу до статичних файлів найкращим рішенням є ProFTPd, так як він є найбезпечнішим, найгнучкішим в налаштуваннях та найстабільнішим серед усіх інших FTP-серверів [56].

Розмежування користувачів легко реалізувати за допомогою вбудованих інструментів Linux для створення користувачів, таких як useradd [57].

2.3 Специфічні аспекти побудови платформ на базі оркестратору контейнерів Kubernetes

2.3.1 Структура та основні елементи Kubernetes

Загальна структура кластеру Kubernetes показана на рисунку 2.16.

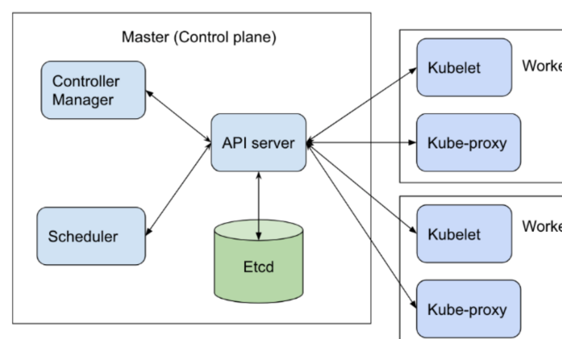

Рисунок 2.16 – Загальна структура кластеру Kubernetes

До неї входять etcd, API server, scheduler, kubelet, kube-proxy, controller manager. Kubelet та kube-proxy розміщаються на Worker-серверах, а etcd, API server, scheduler та controller manager – на Master-сервері.

API server – це сервер, з яким взаємодіє адміністратор і абсолютно всі компоненти системи для виконання будь-яких задач всередині системи [58].

Scheduler, який розподіляє поди (Pods) по вузлам системи [58]. Виконується розподілення на основі багатьох різних параметрів, таких як вимоги поду до ресурсів та наявність їх на Worker-серверах, спеціальних обмежень та допусків, місцезнаходження даних або розділів для даних і так далі [37].

Controller manager, що виконує функції кластерного рівня, такі як реплікація компонентів, відстеження робочих вузлів, обробка аварійних збоїв вузлів і так далі.

Etcd - надійне розподілене сховище даних, яке безперервно зберігає конфігурацію кластера.

Kubelet це агент, який обмінюється інформацією з API-сервером і виконує управління контейнерами на своєму вузлі системи.

Kube-proxy це службовий proxy-сервер, який виконує балансування навантаження мережевого трафіку між компонентами додатку та серверами.

2.3.2 Основні абстракції Kubernetes

В Kubernetes існують базові елементи, на яких базується більшість додатків. Так як вони не являють собою “фізичні” одиниці, то їх Називають абстракціями.

Основними абстракціями для побудови платформ на основі Kubernetes є pod, replicaset, deployment, stateful set, service, ingress, horizontal pod autoscaler, configmap, secret, persistence volume, persistence volume claim.

Pod (под) - це об'єкт Kubernetes, який представляє групу з одного або декількох контейнерів [59]. Всі контейнери всередині пода мають одні і ті ж IP-

адресу і простір портів, вони можуть спілкуватися між собою через локальний сервер або за допомогою взаємодії між процесами. Крім того, всі контейнери мають доступ до загального локального сховищу даних вузла, на якому знаходиться под [37].

ReplicaSet відповідає за групу ідентичних подів (реплік). Якщо виявиться, що подів занадто мало (або багато) в порівнянні зі специфікацією, контролер ReplicaSet запустить нові (або зупинить існуючі) копії, щоб виправити ситуацію [59].

Deployment керують об'єктами ReplicaSet і контролюють поведінку реплік в момент їх оновлення - наприклад, при запуску нової версії додатку. Коли Deployment оновлюється, для керування подами створюється новий об'єкт ReplicaSet. По завершені оновлення стара копія ReplicaSet знищується разом зі своїми подами [59]. Взаємозв'язок між подами, Replicaset та Deployment показано на рисунку 2.17.

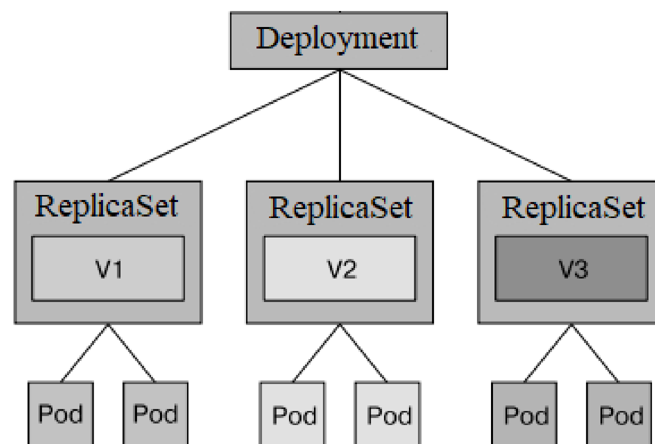

Рисунок 2.17 – Взаємозв'язок між подами, Replicaset та Deployment

StatefulSet – ще одна абстракція. Вона схожа на Deployment, проте працює безпосередньо з подами, без ReplicaSet між ними. Кожен под, створений через StatefulSet має порядковий номер (а не випадково згенерований, як це буде в випадку використання Deployment), стабільне мережеве ім'я, яке доступне по DNS та стабільне сховище, прив'язане до порядкового номеру. Це необхідно при створенні таких кластерів, як наприклад кластер СУБД MySQL.

Service надає одну незмінну IP-адресу або таке ж доменне ім'я, які автоматично перенаправляють на будь-який відповідний под. Принцип роботи сервісу можна побачити на рисунку 2.18.

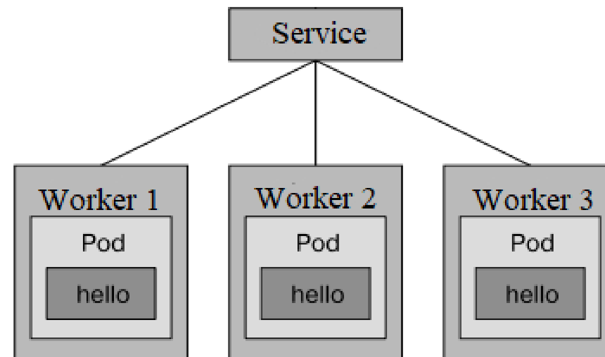

Рисунок 2.18 – Принцип роботи сервісу

Сервіс можна вважати веб-проксі або балансувальником навантаження, який направляє запити до групи внутрішніх подів. Але він не обмежений веб-портами і може направити трафік на будь-який інший порт відповідно з розділом ports в специфікації [59].

Ingress це об'єкт конфігурації Kubernetes, який дозволяє вивести сервіс «назовні» і налаштовувати безліч дрібних деталей. він володіє наступними можливостями [37]:

- Надає сервісу зовнішню URL-адресу;
- Балансує трафік;
- Надає можливість створення SSL-з’єднання.

Робота Ingress продемонстрована на рисунку 2.19.

Як ви можете бачити, контролер Ingress не перенаправив запит в сервіс. Він використовував її тільки для вибору поду [58].

Horizontal pod autoscaler (HPA) спостерігає за вказаними абстракціями, відстежуючи певні показники і відповідно збільшує або зменшує кількість реплік [59]. Даний контролер періодично перевіряє метрики поду, обчислює кількість реплік, необхідне для відповідності цільового значення метрики,

сконфігурованій в ресурсі HorizontalPodAutoscaler, і налаштовує поле replicas на цільовому ресурсі (Deployment, ReplicaSet, StatefulSet) [58]. Одним з найпоширеніших показників автомасштабування є завантаженість процесора.

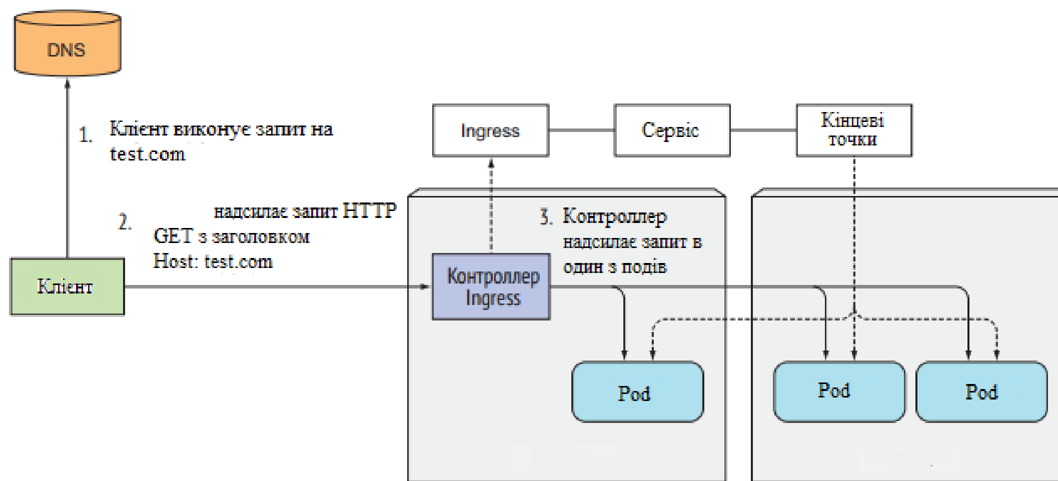

Рисунок 2.19 – Робота Ingress

Можна автомасштабувати розгортування, спираючись на це значення: наприклад, створити контролер HPA, який націлений на 80% -ву завантаженість процесора. Наприклад, коли всі поди в середньому будуть займати 70% від виділеного для поду обсягу, HPA почне скорочувати кількість реплік адже якщо поди не працюють на повну, їх стільки не потрібно. А коли середня завантаженість перевищить вибране значення і досягне 90%, потрібно буде знизити навантаження, додавши більше реплік, HPA модифікує відповідно абстракцію і збільшить кількість подів. Кожен раз, коли контролер HPA бачить необхідність у масштабуванні, він регулює кількість реплік, виходячи зі співвідношення показників поточної і бажаної завантаженості. Якщо абстракція знаходиться дуже близько до бажаної завантаженості, HPA додасть або видалить лише невелику кількість реплік, якщо ж різниця значна, кількість теж буде відкоректована істотно [59].

ConfigMap - це основний об'єкт для зберігання конфігураційних даних в Kubernetes. Його можна представити у вигляді іменованого набору пар «ключ-значення », в якому зберігається конфігурація. За допомогою ConfigMap ви

можете надавати ці дані додатку, впроваджуючи їх в оточення поду або створюючи в ній відповідний файл [59].

Secret - об'єкт спеціального типу, призначений для зберігання конфіденційних даних. Як і ConfigMap, об'єкт Secret можна зробити доступним в контейнері у вигляді змінних середовища або файлу на його диску. Об'єкти Secret в Kubernetes мають тип Opaque: це означає, що їх вміст не показується у виведенні результатів команди kubectl describe, журнальних записах і терміналі, завдяки чому неможливо випадково розкрити конфіденційну інформацію. Оскільки конфіденційна інформація може бути недоступною для виведення (як, наприклад, у випадку з ключем шифрування TLS), об'єкти Secret завжди зберігаються в форматі base64 [59].

PersistentVolume (PV) - це частина сховища в кластері, яка була надана адміністратором або динамічно надана за допомогою класів зберігання (StorageClass). Цей об'єкт API фіксує деталі реалізації сховища, будь то NFS, iSCSI або система зберігання даних для хмарного постачальника [60].

Один з найкращих способів використання постійних томів у Kubernetes полягає у створенні об'єкта PersistentVolumeClaim. Це запит виділення тому PersistentVolume певного типу та розміру: наприклад, тому об'ємом 10 ГБ з високошвидкісним сховищем, доступним для читання та запису. Под може додати цей запит PersistentVolumeClaim в якості розділу диску, і його контейнери зможуть підключити та використовувати це сховище [59].

2.3.3 Безпека в Kubernetes

Корисним механізмом контролю за споживанням ресурсів у вашому кластері є використання просторів імен. Простір імен в Kubernetes надає спосіб поділу кластера на окремі частини в тих чи інших цілях. Наприклад, у можна створити простір імен prod для промислових додатків і test - для експериментів. Як припускає термін «простір імен», імена одного простору не помітні в іншому. Це означає, що в просторах імен prod і test у вас може бути два різних сервісу з

ім'ям demo, які не будуть між собою конфліктувати [59]. Простір імен дозволяє розділити додатки, щоб вони не існували по відношенню один до одного, тому, якщо один додаток буде, наприклад, зламано зловмисниками, вони не отримають контроль над всією системою.

2.3.4 Управління ресурсами в Kubernetes

Контрольні групи Linux (cgroups), використовуються в Kubernetes для обмеження обсягу ресурсів, які може споживати процес. Процес не може використовувати більше, ніж налаштований обсяг ЦП, оперативної пам'яті, пропускної здатності мережі і так далі. Завдяки цьому процеси не можуть перехоплювати ресурси, зарезервовані для інших процесів, що аналогічно тому, коли кожен процес виконується на окремій машині [58].

В Kubernetes ресурси можна обмежити для кожного контейнеру, як це показано на рисунку 2.20.

```
containers:
- name: nginx
  image: nginx:stable-alpine
  resources:
    requests:
      cpu: 50m
      memory: 100Mi
    limits:
      cpu: 100m
      memory: 120Mi
```

Рисунок 2.20 – Налаштування обмеження ресурсів для контейнеру в Kubernetes

Також, можна налаштовувати стандартні ліміти за замовчуванням для кожного контейнеру в кожному namespace і задавати максимально можливі ліміти для namespace.

2.4 Висновок

В даному розділі магістерської кваліфікаційної роботи було виконано аналіз особливостей платформ для розміщення і підтримки серверних додатків з мікросервісною архітектурою.

Описано та проаналізовано існуючі програмні засоби для побудови таких підсистем платформи, як: підсистема управління контейнерами, підсистема автоматизування встановлення платформи, підсистема автоматизованого розгортування клієнтських додатків, підсистема логування та збору і аналізу статистичних даних, підсистема моніторингу, підсистема збереження конфігураційних файлів. Також були розглянуті програмні засоби контейнерування та програмні засоби, необхідні для побудови серверу для статичних файлів.

Описано переваги та недоліки популярних програмних засобів у своїх категоріях. Обрано оптимальні програмні засоби що будуть використані при проектуванні та програмній реалізації інформаційної технології побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою. Для підсистеми управління контейнерами вибрано оркестратор контейнерів Kubernetes. Для реалізації підсистеми автоматизування встановлення платформи обрано Ansible. Для підсистеми автоматизованого розгортування клієнтських додатків обрано Jenkins. Для підсистеми логування та збору і аналізу статистичних даних вибрано Elastic Stack. Для підсистеми моніторингу обрано Prometheus та Grafana. Для підсистеми збереження конфігураційних файлів обрано сервіс BitBucket. В якості програмного засобу контейнерування обрано Docker. Для побудови серверу для статичних файлів вибрано такі програмні засоби як Nginx та ProFTPd.

Також, було розглянуто особливі аспекти побудови платформ на базі оркестратору Kubernetes, розглянуто його загальну структуру.

3 ПРОЕКТУВАННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ ПОБУДОВИ ПЛАТФОРМИ ДЛЯ РОЗМІЩЕННЯ І ПІДТРИМКИ СЕРВЕРНИХ ДОДАТКІВ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

3.1 Проектування платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою

Під проектуванням технічних систем розуміють комплекс робіт з дослідження, розрахунків та конструювання, які мають за мету отримання технічної документації, необхідної для створення нових пристроїв або реалізації нових процесів, що задовольняють задані вимоги [61].

З огляду на масштабність та спеціалізованість подібного проекту потрібно знайти таке рішення, яке б максимально можливо задовольнило користувача в рамках функціональності платформи, а також усунуло недоліки вже існуючих подібних систем. Тому потрібно чітко зрозуміти, якими перевагами повинна володіти така система. Виходячи з аналізу предметної області та систем-аналогів можна сформувати вимоги до нової розробки:

- Можливість інтеграції інтерфейсу;
- Проста але водночас повна функціональність;
- Розмежування адміністративного і користувацького рівнів;
- Відкритість системи для можливості внесення необхідних змін та удосконалення системи
- Зменшення часу розгортування додатку у порівнянні з аналогами;
- Наявність наступних підсистем: підсистеми управління контейнерами, підсистеми автоматизування встановлення платформи, підсистеми автоматизованого розгортування клієнтських додатків, підсистеми логування та збору і аналізу статистичних даних, підсистеми моніторингу, підсистеми збереження конфігураційних файлів.

Процес проектування ІС - це процес побудови такого опису цієї системи, який дає можливість її вивчення та створення. Для проектування інформаційних систем широко використовується мова UML - уніфікована мова візуального моделювання. Це мова для візуалізації, специфікації, конструювання і документування програмних систем. Мова UML передбачає використання сукупності діаграм [62].

Основні можливості обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою можна представити за допомогою UML-діаграми прецедентів, яка представлена на рисунку 3.1.

Для представлення структури системи в цілому доцільно застосувати діаграму компонентів. Діаграма компонентів показує взаємозв'язки між підсистемами (логічними або фізичними), з яких складається система що моделюється [63]. Діаграма компонентів наведена на рисунку 3.2.

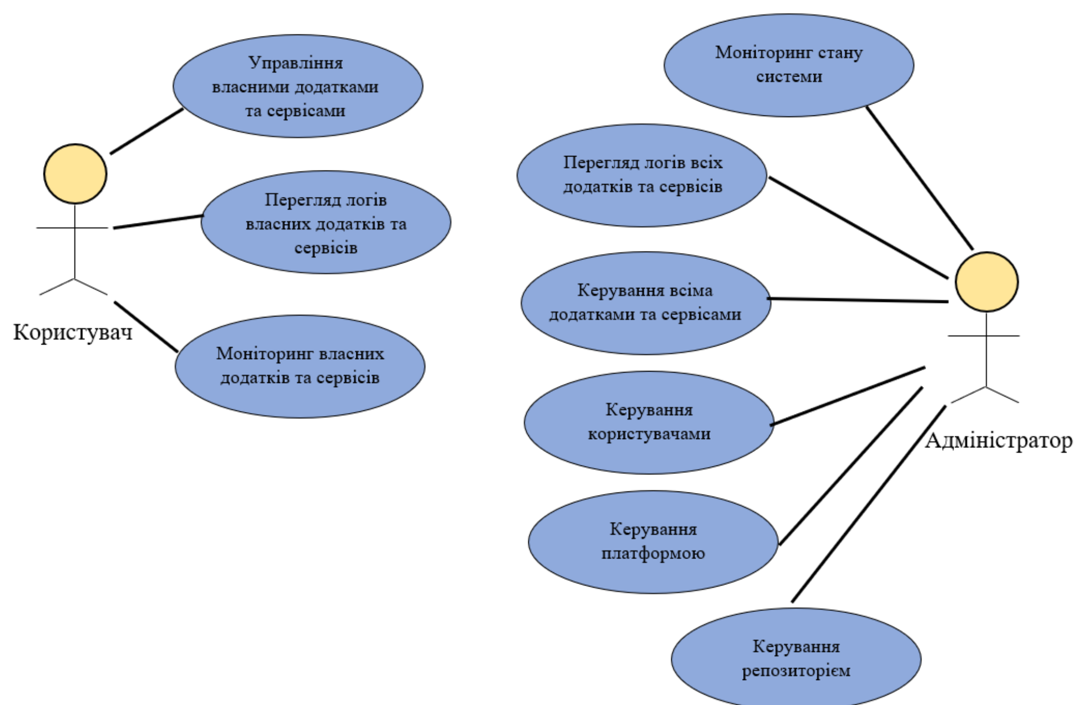

Рисунок 3.1 – UML-діаграма прецедентів обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою

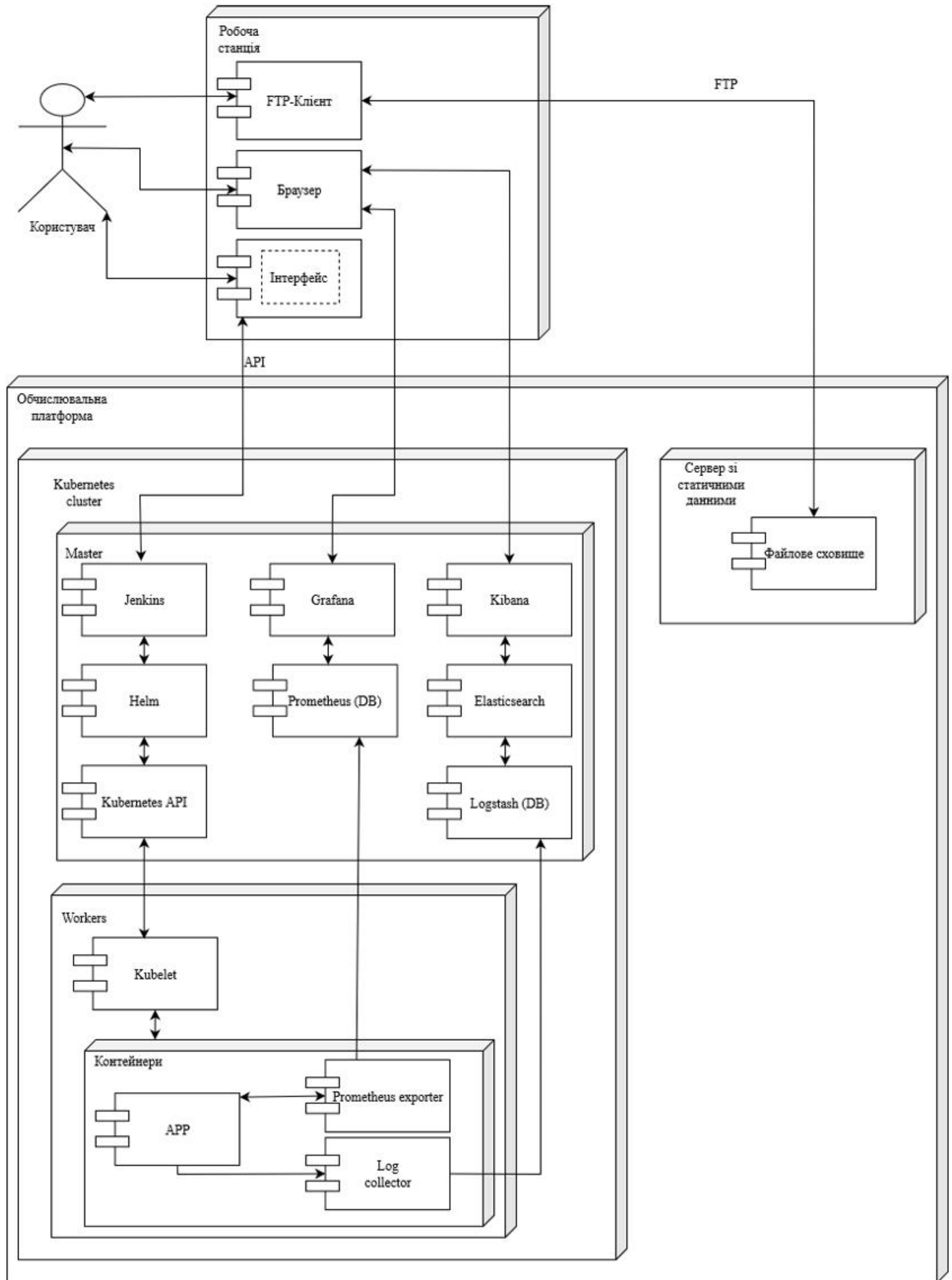

Рисунок 3.2 – UML-діаграма компонентів

3.2 Розробка структури платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою

Відповідно до поставленої мети було розроблено загальну структурну схему платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою, що представлена на рисунку 3.3.

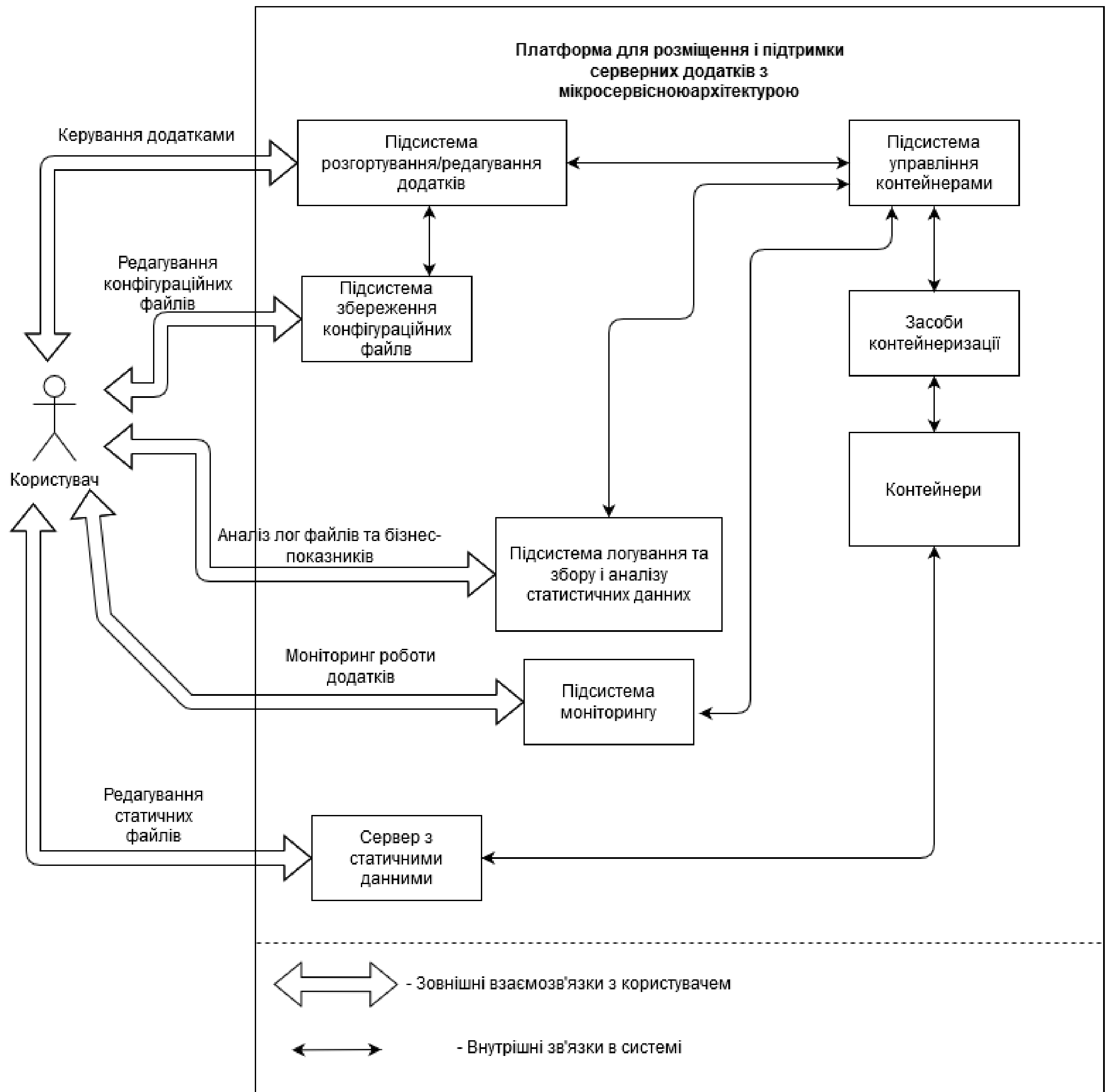

Рисунок 3.3 – Загальна структурна схема платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою

Користувач зможе розгортувати нові та редагувати параметри вже існуючих додатків, аналізувати логи та статичні данні, слідкувати за станом своїх додатків. Для збереження статичних файлів він матиме доступ на окремий сервер, де зможе керувати статичними даними своїх додатків.

При необхідності внести зміни в працюючий додаток або створити новий, користувач завантажує нові чи відредаговані конфігураційні файли у спеціалізоване сховище. Після цього він ініціює запуск потрібних дій через підсистему розгортування та редагування додатків.

Підсистема розгортування та редагування додатків виконує скачування відповідних конфігураційних файлів, обробляє їх, та створює завдання для підсистеми управління контейнерами. Яка, в свою чергу, отримує відповідні конфігураційні файли, після чого інтерпретує їх та за допомогою спеціальних інструментів створює потрібні маніфести, відповідно до заданих значень у конфігураційних файлах. Після чого виконується розгортування потрібних сервісів та ПЗ.

Сервіси та ПЗ, по мірі необхідності, можуть звертатись до серверу статичних даних та отримувати потрібні файли.

Користувач в процесі використання системи може виконувати моніторинг своїх додатків, окремих сервісів і т.д., а також, отримувати доступ до лог-файлів та даних статистики з метою проведення аналізу роботи додатку на певному відрізку часу.

Використання такої структури дозволяє застосувати підхід DevOps. DevOps дозволяє поліпшити якість ПЗ шляхом прискорення циклу випуску версій за допомогою хмарних методик і автоматизації. Швидкість, динамічність, взаємодія, автоматизація та якість ПО є ключовими цілями DevOps. Підхід DevOps привносить практики написання ПО в світ системного адміністрування та надає інструменти і робочі процеси для швидкої і динамічної спільної побудови складних систем. З цією концепцією нерозривно переплітається ідея інфраструктури як коду (IaaS) [59].

3.3 Розробка схеми алгоритму роботи платформи

Для автоматичного встановлення платформи застосовуються плейбуки для Ansible. Плейбуки написані на мові YAML, яку підтримує Ansible. Кожен плейбук виконує відповідну роль при запуску. В ролі знаходяться завдання, які ця роль виконує, файли які може знадобитись скопіювати на сервер, шаблони, спеціальні змінні для конкретної ролі і т.д. Схему алгоритму роботи плейбуку для встановлення кластера Kubernetes можна побачити на рисунку 3.4. А схему роботи плейбуку по створенню середовища для нового додатку на платформі можна зайти на рисунку 3.5.

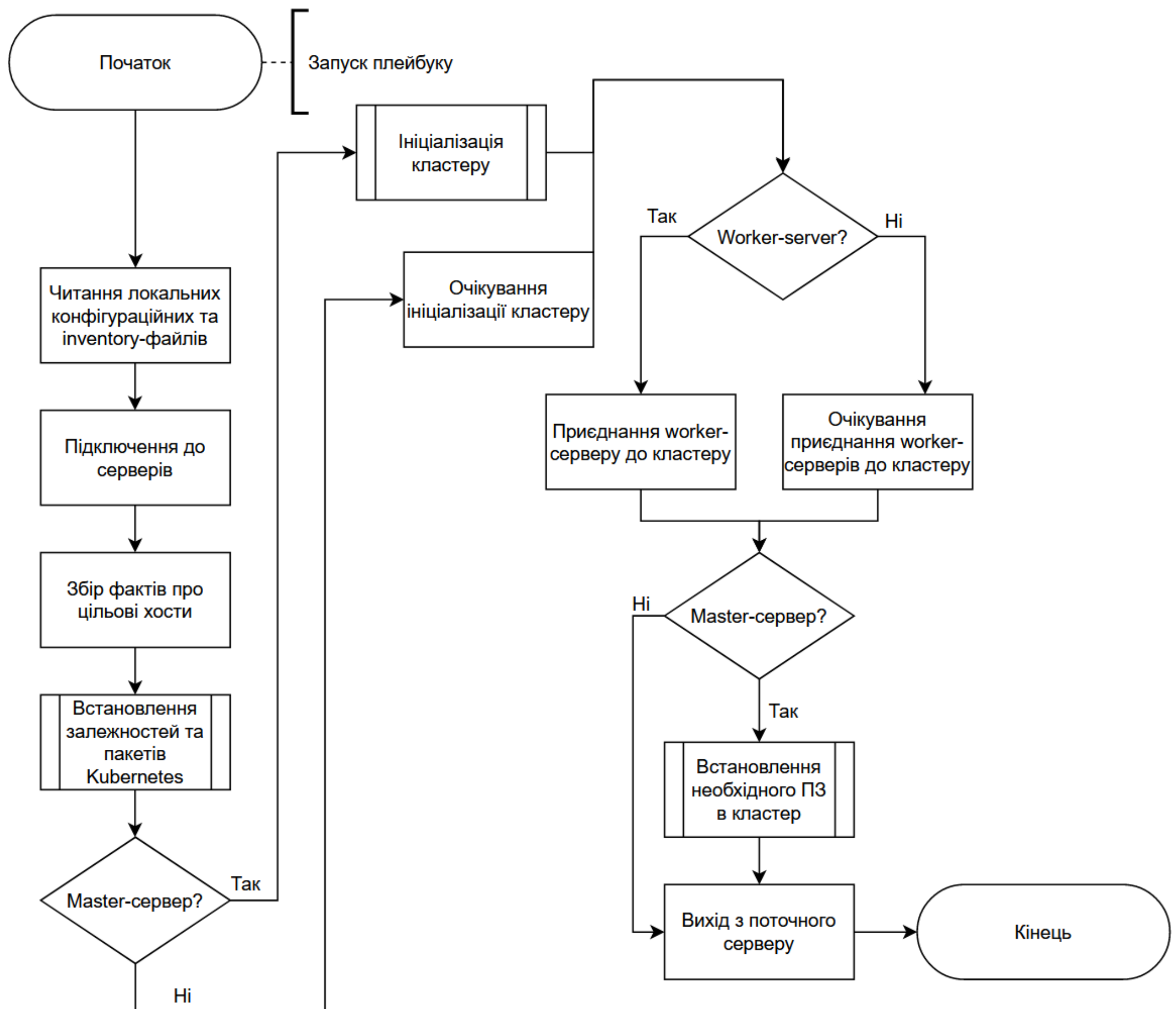

Рисунок 3.4 – Схема алгоритму роботи плейбуку для встановлення кластера Kubernetes

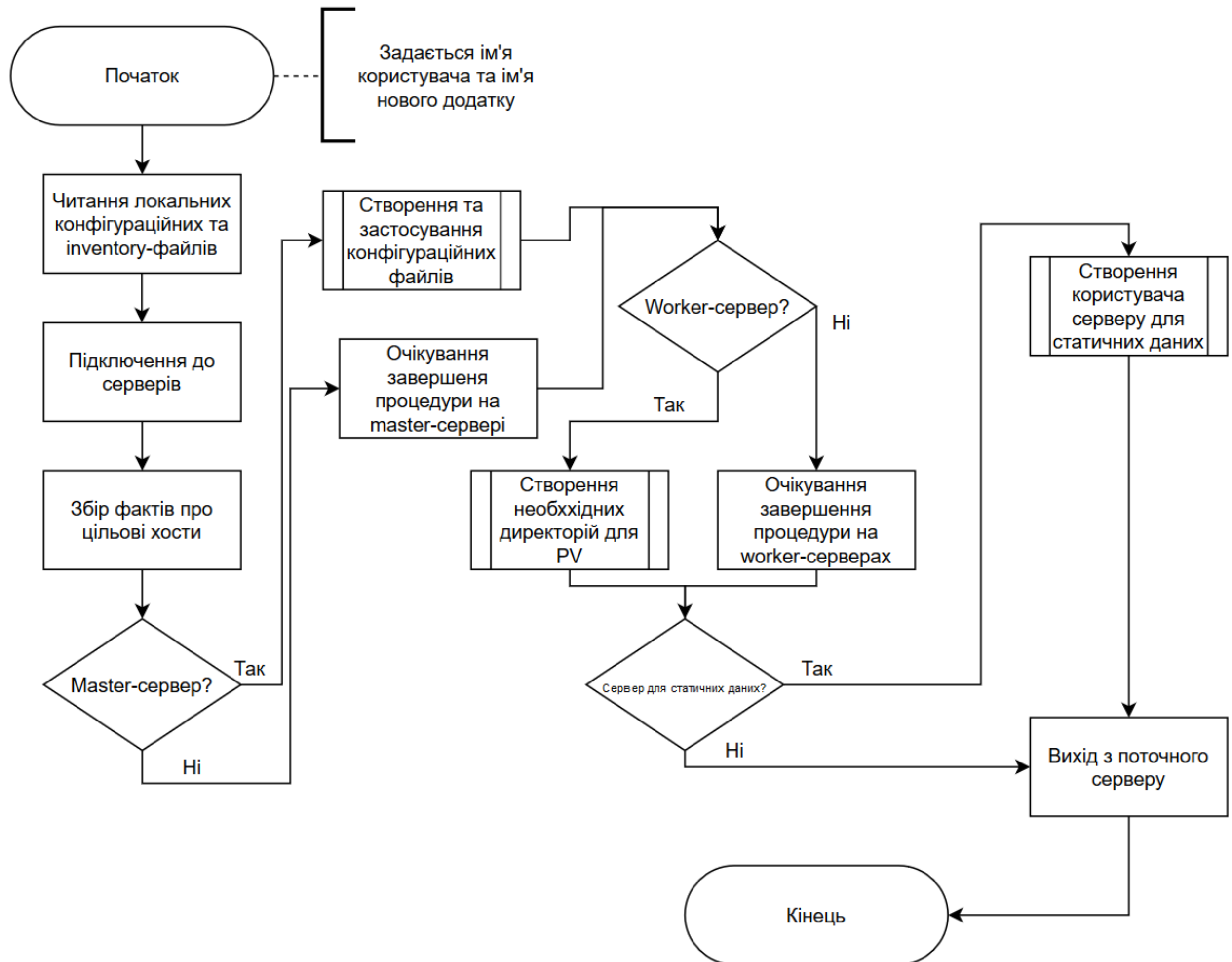

Рисунок 3.5 – Схема алгоритму роботи плейбуку для створення середовища для нового додатку на платформі

Для розгортування всіх додатків застосовується пакетний менеджер для Kubernetes – Helm [64]. За допомогою нього можна створити вже готові шаблони, як для, безпосередньо, додатків, так і для допоміжного ПЗ, такого як БД, веб-сервери та ін. Для розгортування додатків за допомогою Helm було створено шаблони для різних видів вузлів додатку. Наприклад, для власне коду додатку, БД MySQL та веб-серверу Nginx. Користувач, редагуючи шаблон values-файлу задає потрібні опції для розгортування майбутнього додатку та його складових. Розгортуються Helm-чарти в кластері Kubernetes за допомогою Jenkins. Схему розгортування додатку можна побачити на рисунку 3.6.

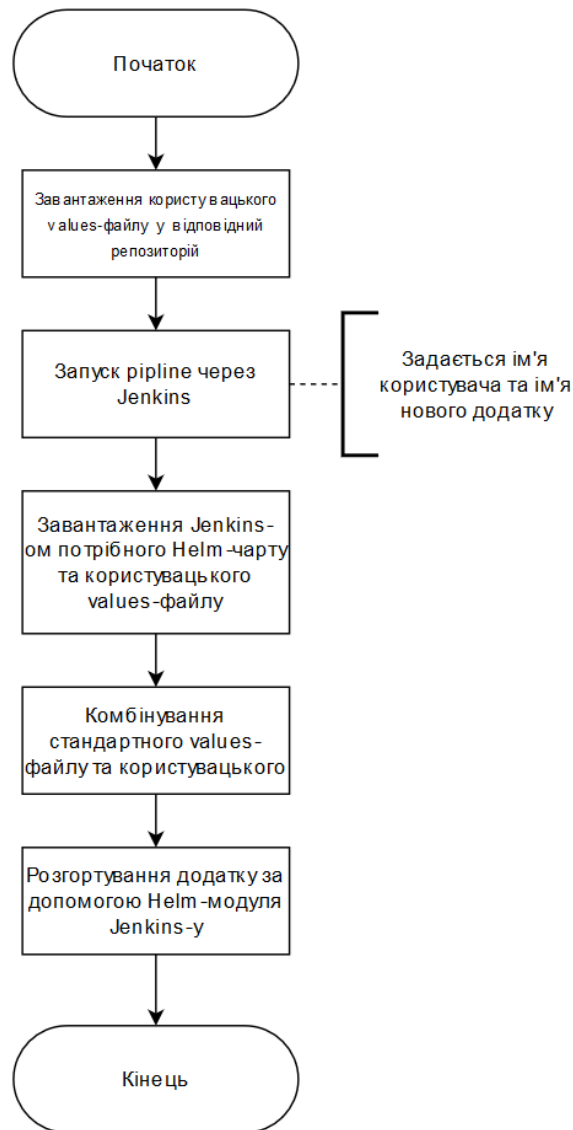

Рисунок 3.6 – Схема алгоритму розгортування додатку з Helm-чарту за допомогою Jenkins

Сам Kubernetes завжди слідкує за тими додатками, які запущені в системі, за заданими критеріями. Для прикладу, він контролює мінімальну кількість реплік конкретного вузлу додатку, контролює балансування трафіку на конкретні репліки, контролює кількість реплік, виходячи з навантаження кожної з реплік. Схему контролю реплік Kubernetes-ом представлено на рисунку 3.7.

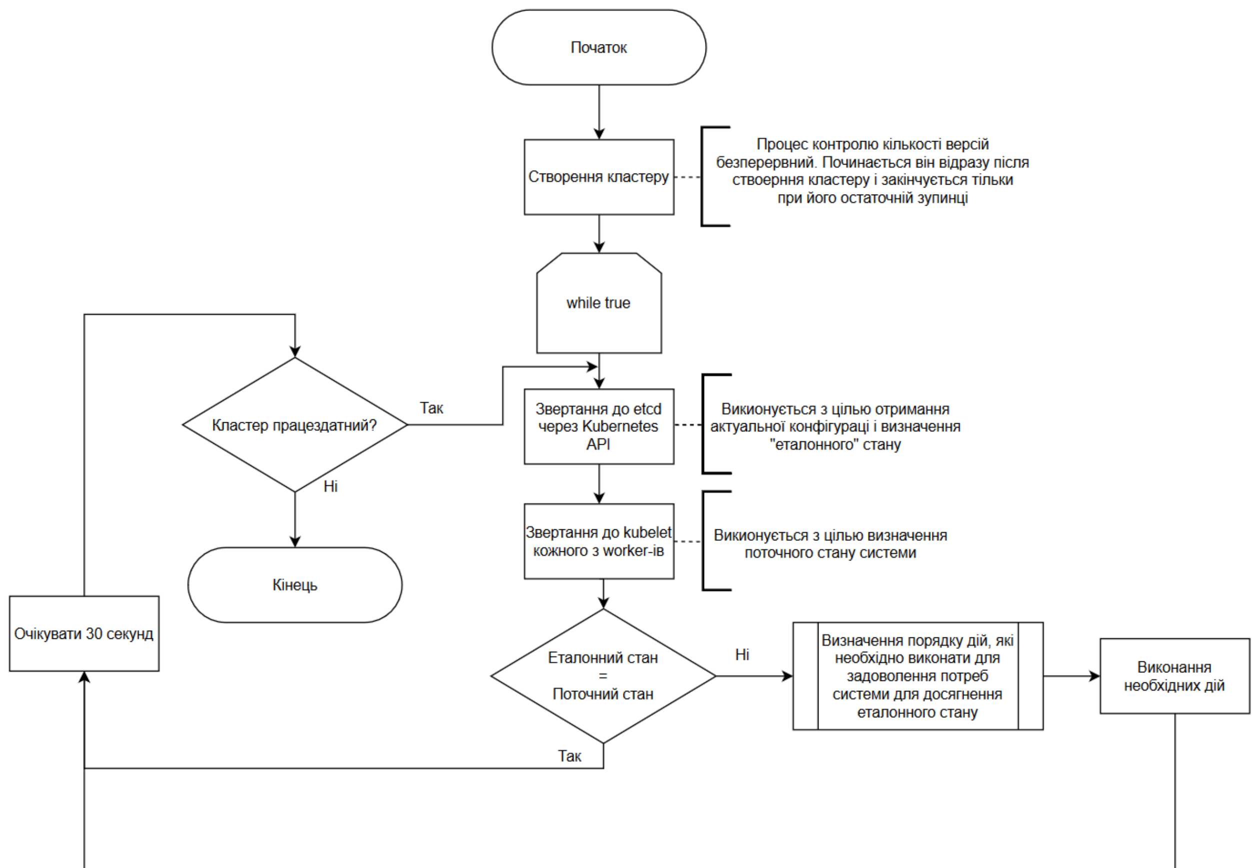

Рисунок 3.7 – Схема алгоритму контролю реплік додатків Kubernetes-ом

3.4 Тестовий приклад роботи платформи і аналіз результатів

Тестування роботи платформи проводилось за допомогою тестування окремих стадій роботи системи. Тестувалось автоматичне встановлення платформи, а також автоматизоване розгортування додатків на платформі і підтримка їх роботи.

3.4.1 Тестування автоматичного встановлення платформи

Для автоматичного встановлення платформи було створено плейбуки для Ansible. Структура директорії з плейбуками відповідає Best Practice [65], які рекомендовані до застосування розробниками. Кінцева структура представлена на рисунку 3.8.

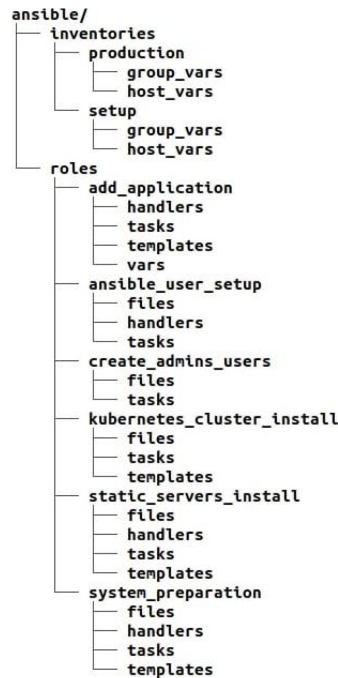

Рисунок 3.8 – Структура репозиторію з плейбуками Ansible

Були створені такі плейбуки як:

- playbook_ansible_user_setup.yaml – плейбук, який запускається для створення спеціального користувача для Ansible, так як на новому сервері або після того як сервер був перевстановлений немає такого користувача з доступом по ключу. А доступ до серверу як користувач «root» та з паролем є не надто безпечним. Для цього плейбуку використовується окремий inventory-файл, який дозволяє використовувати пароль та містить в собі інші налаштування, які дозволяють підключатись до нових серверів;
- playbook_creating_admins_users.yaml – плейбук, який створює користувачів-адміністраторів на всіх серверах та додає їх ключі для доступу за ними;
- playbook_system_preparation.yaml – плейбук, який виконує загальні налаштування для всіх серверів одночасно. Наприклад, встановлення системного програмного забезпечення, підключення репозиторіїв і т.д.;

- playbook_kubernetes_cluster_install.yaml – плейбук, який встановлює потрібне ПЗ для роботи кластеру Kubernetes, ініціює створення кластеру, та подальше встановлення ПЗ в кластері, такого як Jenkins, Helm, Ingress і т.п.;
- playbook_static_servers_install.yaml – плейбук, що виконує встановлення та налаштування потрібного ПЗ на сервері з статичними файлами.
- playbook_add_application.yaml – плейбук, який створює всіх необхідних користувачів, необхідні папки та виконує налаштування для нового додатку, який буде розгорнутий на платформі.

Для тестування роботи плейбуків ОС на серверах було перевстановлено і плейбуки були запущені у відповідному порядку. Кожен з плейбуків виводить лог виконання кожного з етапів на екран. Приклад такого логу можна побачити на рисунку 3.9.

```
TASK [system_preparation : Update cache and install service package] *************************************
changed: [node2.k8smd.pp.ua]
changed: [static1.k8smd.pp.ua]
changed: [node1.k8smd.pp.ua]
changed: [node3.k8smd.pp.ua]
changed: [master1.k8smd.pp.ua]

TASK [system_preparation : Add Docker GPG apt Key] ******************************************************
skipping: [static1.k8smd.pp.ua]
changed: [node1.k8smd.pp.ua]
changed: [node2.k8smd.pp.ua]
changed: [node3.k8smd.pp.ua]
changed: [master1.k8smd.pp.ua]

TASK [system_preparation : Add Docker repo] *************************************************************
skipping: [static1.k8smd.pp.ua]
changed: [node2.k8smd.pp.ua]
changed: [master1.k8smd.pp.ua]
changed: [node1.k8smd.pp.ua]
changed: [node3.k8smd.pp.ua]

TASK [system_preparation : Install Docker] **************************************************************
skipping: [static1.k8smd.pp.ua]
changed: [master1.k8smd.pp.ua]
changed: [node1.k8smd.pp.ua]
changed: [node2.k8smd.pp.ua]
changed: [node3.k8smd.pp.ua]
```

Рисунок 3.9 – Лог роботи Ansible

Так як, навіть при схожих логічних діях, які потрібно виконати на всіх серверах, частина з них не повинна виконуватись на деяких з них. Наприклад, в логові, наведеному вище, можна побачити такі стрічки, як «skipping: [static1.k8smd.pp.ua]». Вони з’являються, так як, для прикладу, завдання по встановленню Docker не повинно виконуватись на сервері для статичних файлів, так як Docker там не використовується. Реалізовано це за допомогою спеціальних

міток в Inventory для окремих серверів, які наведені на рисунку 3.10, та за допомогою так званого «селектору» в завданні, що наведений на рисунку 3.11.

```
1 kube_role: "none"
2 role: "static"
```

```
1 kube_role: "master"
2 role: "k8s"
```

Рисунок 3.10 – Мітки в Inventory-файлі для різних серверів

```
- name: Add Docker GPG apt Key
  when: role == "k8s"
  apt_key:
    url: https://download.docker.com/linux/ubuntu/gpg
    state: present

- name: Add Docker repo
  when: role == "k8s"
  apt_repository:
    repo: deb https://download.docker.com/linux/ubuntu bionic stable
    state: present

- name: Install Docker
  when: role == "k8s"
  apt:
    update_cache: yes
    name: docker-ce
```

Рисунок 3.11 – Селектор типу «when» в завданнях плейбуку

Після запуску всіх відповідних плейбуків можна перевірити стан кластеру, запущене стандартне ПЗ на ньому, стан ПЗ на сервері для статичних даних. Результати перевірки наведені на рисунках 3.12 - 3.17.

```
root@master1:~# kubectl get no
NAME      STATUS   ROLES                     AGE     VERSION
master1   Ready    ingress,jenkins,master    3m48s   v1.19.3
node1     Ready    <none>                    3m10s   v1.19.3
node2     Ready    <none>                    3m10s   v1.19.3
node3     Ready    <none>                    3m10s   v1.19.3
root@master1:~# kubectl get po --all-namespaces
NAMESPACE       NAME                                        READY   STATUS      RESTARTS   AGE
ingress-nginx   ingress-nginx-admission-create-7m7ks        0/1     Completed   0          3m51s
ingress-nginx   ingress-nginx-admission-patch-l7khr         0/1     Completed   0          3m51s
ingress-nginx   ingress-nginx-controller-7c5db75b75-4g6p5   1/1     Running     0          4m1s
jenkins         jenkins-f5bf95df-qgs4k                      2/2     Running     0          3m12s
kube-system     calico-kube-controllers-6dfcd885bf-xr7lj    1/1     Running     0          4m45s
kube-system     calico-node-5sgfv                           1/1     Running     0          4m45s
kube-system     calico-node-8zg78                           1/1     Running     0          4m29s
kube-system     calico-node-bqcxn                           1/1     Running     0          4m27s
kube-system     calico-node-hrzk8                           1/1     Running     0          4m33s
kube-system     coredns-f9fd979d6-bxzn4                     1/1     Running     0          4m45s
kube-system     coredns-f9fd979d6-djh7k                     1/1     Running     0          4m45s
kube-system     etcd-master1                                1/1     Running     0          4m53s
kube-system     kube-apiserver-master1                      1/1     Running     0          4m53s
kube-system     kube-controller-manager-master1             1/1     Running     0          4m53s
kube-system     kube-proxy-l8kc5                            1/1     Running     0          4m45s
kube-system     kube-proxy-n4wbp                            1/1     Running     0          4m33s
kube-system     kube-proxy-qs9d7                            1/1     Running     0          4m27s
kube-system     kube-proxy-sdn4p                            1/1     Running     0          4m29s
kube-system     kube-scheduler-master1                      1/1     Running     0          4m53s
```

Рисунок 3.12 – Перевірка готовності кластеру Kubernetes та встановленого ПЗ з консолі

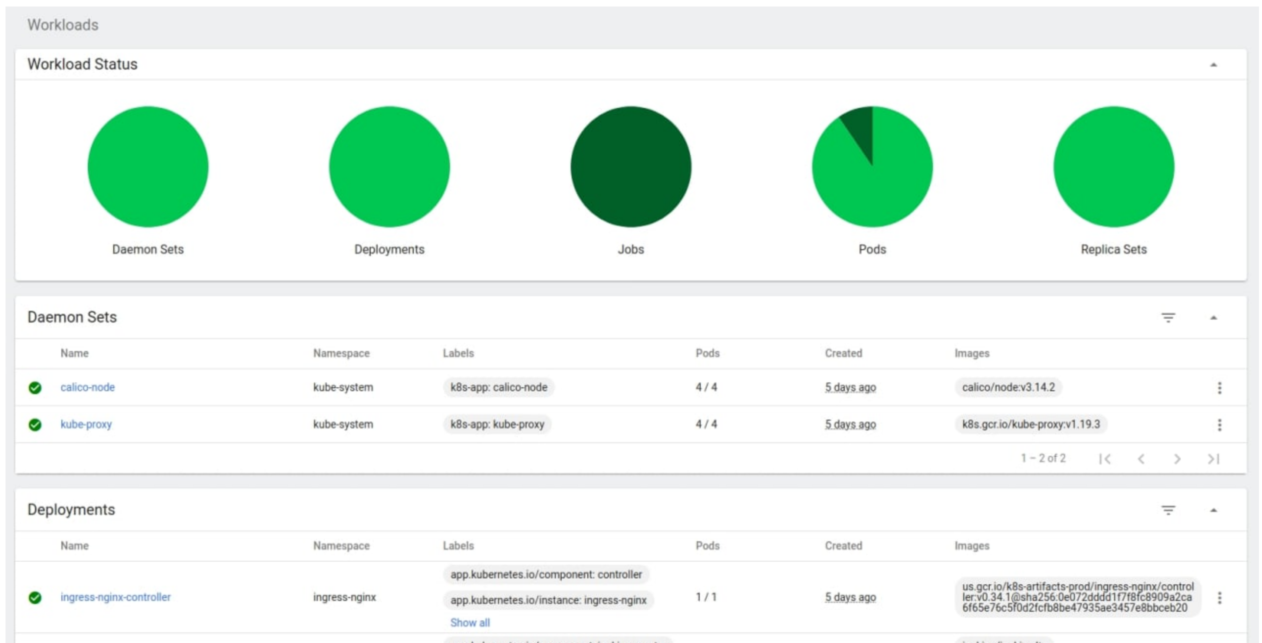

Рисунок 3.13 – Перевірка готовності кластеру Kubernetes та встановленого ПЗ за допомогою графічного інтерфейсу

```
root@static1:/home/atkachuk# ps auSfx| grep -E "(nginx|proftpd)" | grep -v grep
root       521  0.0  0.0 141128  1548 ?        Ss   12:41   0:00 nginx: master process /usr/sbin/nginx -g daemon on; master_process on;
www-data   522  0.0  0.3 143592  6524 ?        S    12:41   0:00  \_ nginx: worker process
www-data   523  0.0  0.3 143592  6524 ?        S    12:41   0:00  \_ nginx: worker process
proftpd    567  0.0  0.1 123512  3660 ?        Ss   12:41   0:00 proftpd: (accepting connections)
```

Рисунок 3.14 – Перевірка встановленого ПЗ на сервері для статичних даних

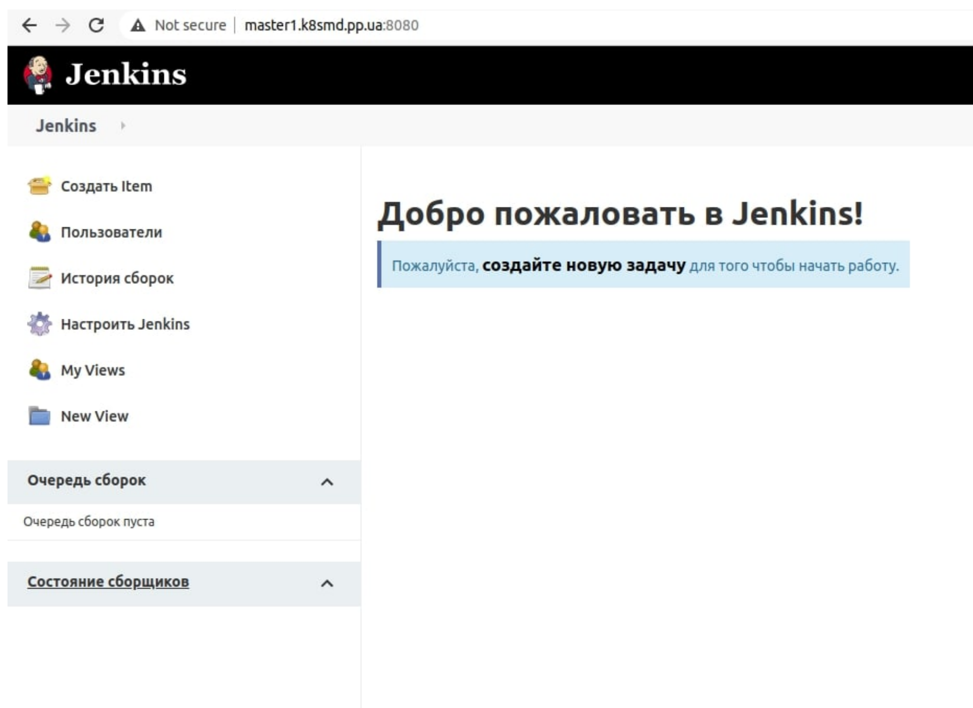

Рисунок 3.15 – Перевірка безпосередньої роботи автоматично встановленого Jenkins

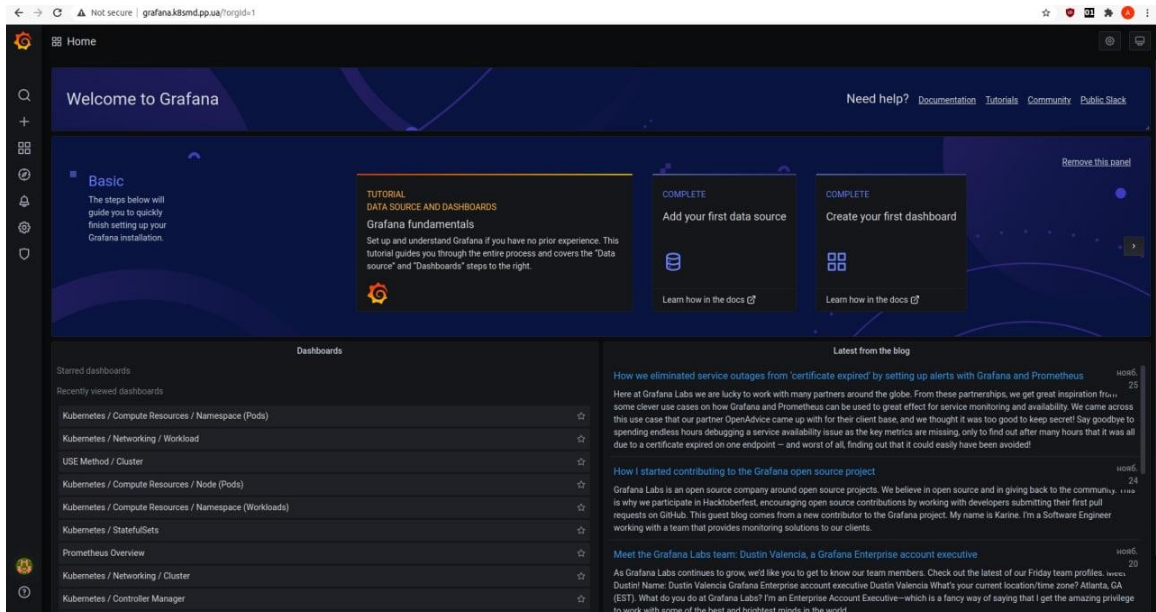

Рисунок 3.16 – Перевірка безпосередньої роботи автоматично встановленого моніторингу

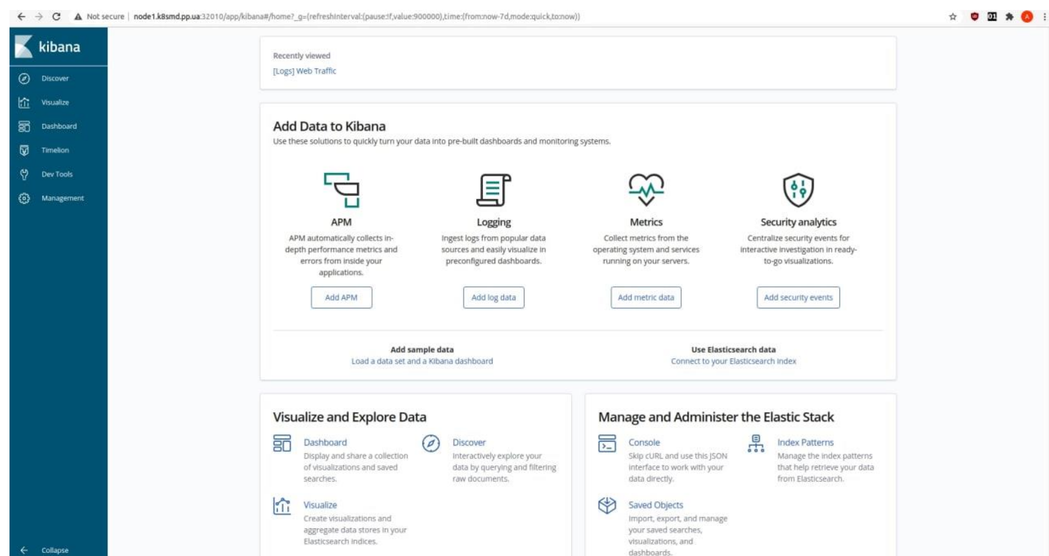

Рисунок 3.17 – Перевірка безпосередньої роботи автоматично встановленого ELK-кластеру

3.4.2 Тестування автоматизованого розгортування додатків на платформі

Для тестування автоматизованого розгортування додатків було створено простий додаток на PHP, з використанням БД MySQL та веб/проксі-серверу Nginx.

Docker-образ для PHP було створено власноруч, на базі вже існуючого, офіційного образу PHP-FPM. При збірці образу застосовувався Docker-файл, який зображено на рисунку 3.18.

```
FROM php:7.4-fpm
RUN apt-get update \
    && apt-get install -y libmcrypt-dev libicu-dev libonig-dev libpq-dev \
    && docker-php-ext-install mysqli \
    && docker-php-ext-install pdo_mysql \
    && docker-php-ext-install iconv \
    && docker-php-ext-install intl \
    && docker-php-ext-install opcache \
    && docker-php-ext-install mbstring
COPY ./code/ /opt/www/
RUN chown -R www-data: /opt/www
```

Рисунок 3.18 – Docker-файл для тестового додатку

У Docker-образ було включено всі необхідні модулі та, власне, код додатку.

Для автоматичного розгортування додатку через Jenkins та Helm було створено спеціальні чарти для Helm які дозволяють згенерувати з шаблонів yaml-файли Kubernetes для будь-якого ПЗ, pipeline для Jenkins, який відповідає за автоматичне встановлення додатку на платформі, а також values-файл, який буде застосовуватись при розгортуванні саме цього додатку за допомогою вже створеного раніше Helm-чарту. Helm-шаблон можна побачити на рисунку 3.19. Jenkins pipeline показано на рисунку 3.20. Values-файл для тестового PHP-додатку зображено на рисунку 3.21.

Для розгортування за допомогою Jenkins також було створено спеціальний Docker-образ для контейнеру-агенту, в якому буде створюватись готовий для використання values-файл згенерований з стандартного та користувацького values-файлів. Docker-файл для образу контейнеру-агенту зображено на рисунку 3.22.

```
{{- $appName := .Values.app.name }}

{{- if eq .Values.app.type "stateless" -}}
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.app.name }}-deployment
  labels:
    app: {{ .Values.app.name }}
spec:
  replicas: {{ .Values.app.replicas }}
  selector:
    matchLabels:
      app: {{ .Values.app.name }}
  template:
    metadata:
      labels:
        app: {{ .Values.app.name }}
    spec:
      containers:
      - name: {{ .Values.app.name }}
        image: {{ .Values.app.image }}
        imagePullPolicy: IfNotPresent
        {{- if .Values.probes.enabled }}
        {{- if eq .Values.probes.type "http" }}
        livenessProbe:
          httpGet:
            path: {{ .Values.probes.path }}
            port: {{ .Values.probes.port }}
          initialDelaySeconds: 30
          periodSeconds: 10
          timeoutSeconds: 5
        readinessProbe:
          httpGet:
            path: {{ .Values.probes.path }}
            port: {{ .Values.probes.port }}
          initialDelaySeconds: 10
          periodSeconds: 5
          timeoutSeconds: 1
```

Рисунок 3.19 – Helm-шаблон для генерації готових yaml-файлів для Kubernetes

```
pipeline {
    agent {
        kubernetes {
            label 'deploy-service-pod'
            defaultContainer 'jnlp'
            yaml """
apiVersion: v1
kind: Pod
metadata:
  labels:
    job: deploy-service
spec:
  nodeSelector:
    node-role.kubernetes.io/jenkins: "true"
  tolerations:
  - key: "node-role.kubernetes.io/master"
    operator: "Exists"
    effect: "NoSchedule"
  containers:
  - name: deployer
    image: k8smd/helm_deployer:latest
    command: ["cat"]
    tty: true
"""
        }
    }

    stages {
        stage ('Clone chart from git') {
            steps {
                container ('deployer') {
                    git credentialsId:'bitbucket',
                    url:'ssh://git@bitbucket.org/k8s-md/dev.git', branch:"helm/${SERVICE_TYPE}"
                }
            }
        }
        stage ('Save helm files') {
            steps {
                container ('deployer') {
                    sh "cp -R helm/${SERVICE_TYPE} /tmp/"
                }
            }
        }
        stage ('Clone application custom values from git') {
            steps {
                container ('deployer') {
                    git credentialsId:'bitbucket',
                    url:"ssh://git@bitbucket.org/k8s-md/${PROJECT_NAME}.git", branch:"service/${SERVICE_NAME}"
                }
            }
        }
        stage ('Merge values') {
            steps {
                container ('deployer') {
                    sh "perl /home/values_generator.pl /tmp/${SERVICE_TYPE}/values.yaml values.yaml"
                }
```

Рисунок 3.20 – Jenkins pipeline для автоматичного розгортування додатків

```yaml
1   user: user1
2   app:
3     name: test-app
4     # type can be 'stateless' or 'stateful'. stateless by default
5     type: stateless
6     replicas: 3
7     maxreplicas: 5
8     image: k8smd/test_app_php:latest
9     resources:
10      req:
11        cpu: 50m
12        mem: 100Mi
13      lim:
14        cpu: 100m
15        mem: 120Mi
16
17  service:
18    # 'true' or 'false'. true by default
19    enabled: true
20    appPort: 9000
21
22  probes:
23    enabled: true
24    type: port
25    port: 9000
26
27  job:
28    enabled: true
29    useGeneralConfs: false
30    useGeneralSecrets: false
31    useGeneralEnvs: true
32    command: php /root/dbcreate_test.php
33    configs:
34      enabled: true
35      dbcreatescript:
36        name: dbcreate_test.php
37        mountDir: /root
38        data: |+
39          <?php
```

Рисунок 3.21 –Values-файл для тестового додатку

```dockerfile
FROM alpine

ENV HELM_LATEST_VERSION="v3.2.4"

RUN apk add --update ca-certificates \
 && apk add --update -t deps wget git openssl bash curl \
 && wget -q https://get.helm.sh/helm-${HELM_LATEST_VERSION}-linux-amd64.tar.gz \
 && tar -xf helm-${HELM_LATEST_VERSION}-linux-amd64.tar.gz \
 && mv linux-amd64/helm /usr/local/bin \
 && apk del --purge deps \
 && rm /var/cache/apk/* \
 && rm -f /helm-${HELM_LATEST_VERSION}-linux-amd64.tar.gz \
 && apk add --no-cache curl tar make gcc build-base wget gnupg ca-certificates g++ git gd-dev gzip \
 && apk add --no-cache zlib zlib-dev \
 && apk add --no-cache perl perl-dev

RUN apk add --no-cache perl-app-cpanminus

RUN cpanm App::cpm
WORKDIR /usr

RUN cpm install YAML \
 && cpm install Hash::Merge::Simple

COPY values_generator.pl /home

ENV PERL5LIB=/usr/local/lib/perl5
ENV PATH=/usr/local/bin:$PATH

ENTRYPOINT ["helm"]
CMD ["help"]
```

Рисунок 3.22 – Docker-файл для образу контейнеру-агенту

Для генерації values-файл з стандартного та користувацького values-файлів було створений спеціальний скрипт на мові програмування Perl, що показаний на рисунку 3.23.

```
#!/usr/bin/perl

use feature qw(say);
use YAML 'LoadFile';
use YAML 'Dump';
use Hash::Merge::Simple 'merge';

#say Dump(merge(map{LoadFile($_)}@ARGV));
my $new_yaml = Dump(merge(map{LoadFile($_)}@ARGV));
open($fh, '>', '/tmp/variables.yaml') or die "Error";
print $fh $new_yaml;
close $fh;
```

Рисунок 3.23 – Perl-скрипт для генерації values-файлів готових до використання при розгортуванні

За ходом виконання Pipeline можна стежити з графічного інтерфейсу, як це показано на рисунку 3.24

Після його розгортування можна перевірити його наявність у системі через консоль, графічний інтерфейс, а також перевірити наявність доступу до моніторингу та ELK-кластеру. А також безпосередню роботу самого додатку. Виконання таких перевірок представлено на рисунках 3.25 – 3.30

Рисунок 3.24 –Хід виконання Pipeline-у

```
root@master1:~# kubectl get po -n test-app
NAME                                         READY   STATUS      RESTARTS   AGE
test-app-addrowdb-cronjob-1606579200-76fj6   0/1     Completed   0          25m
test-app-addrowdb-cronjob-1606580100-7p4tc   0/1     Completed   0          10m
test-app-deployment-6bc5578745-5r5wn         1/1     Running     0          33m
test-app-deployment-6bc5578745-mr58p         1/1     Running     0          33m
test-app-deployment-6bc5578745-mxd97         1/1     Running     0          33m
test-app-job-f9j4q                           0/1     Completed   0          33m
test-app-mysql-statefulset-0                 1/1     Running     0          37m
test-app-ngx-deployment-6957f64c9c-jd7x5     1/1     Running     0          32m
test-app-ngx-deployment-6957f64c9c-llfvn     1/1     Running     0          32m
test-app-ngx-deployment-6957f64c9c-nfprt     1/1     Running     0          32m
```

Рисунок 3.25 – Перевірка наявності додатку у системі через консоль

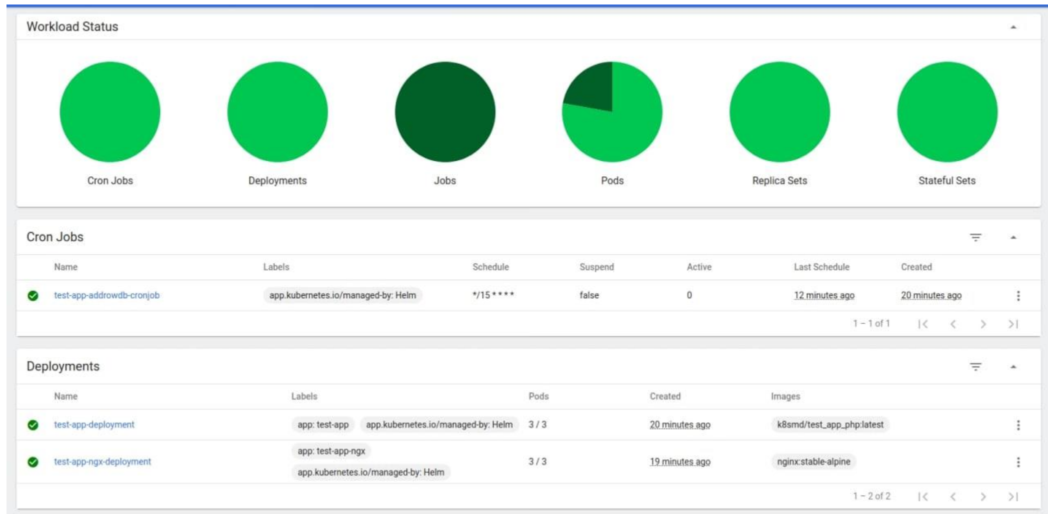

Рисунок 3.26 – Перевірка наявності додатку у системі в графічному інтерфейсі

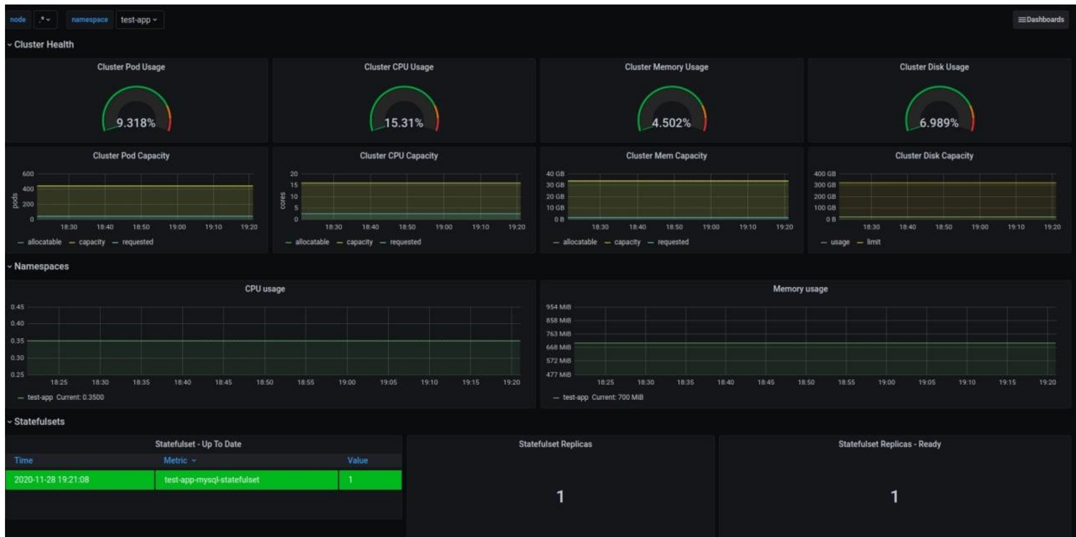

Рисунок 3.27 – Перевірка доступу до моніторингу додатку

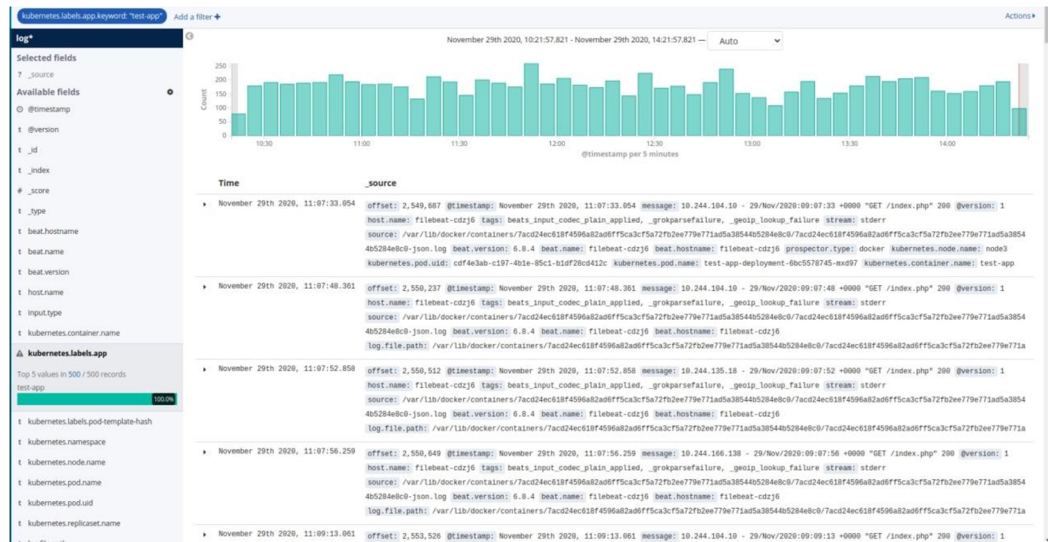

Рисунок 3.28 – Перевірка наявності логів у ELK

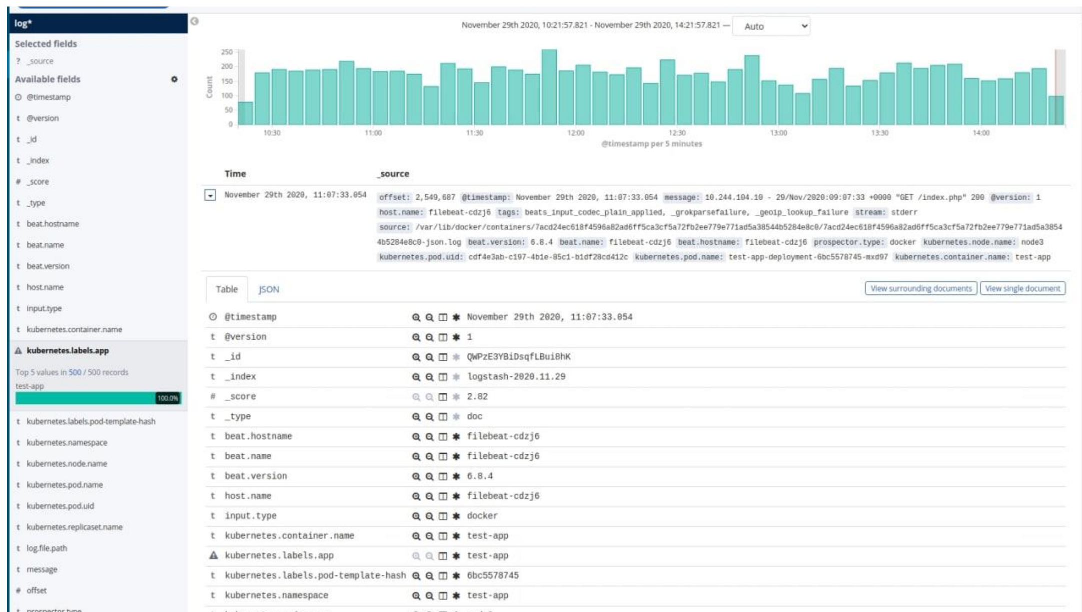

Рисунок 3.29 – Розширена інформація по кожному запису в логові у ELK

Рисунок 3.30 – Перевірка роботи додатку

3.4.3 Тестування автоматизованої підтримки роботи додатків на платформі

Для тестування автоматизованої підтримки роботи додатків на платформі було вручну відтворено умови відмови однієї з реплік додатку, а саме, знищення однієї з реплік. Для прикладу, знищимо репліку з іменем «test-app-deployment-6bc5578745-5r5wn»

Після знищення репліки, Controller Manager кластеру зафіксував зміни, перевірив поточний стан системи, «еталонний» стан, виявив невідповідність і запустив нову репліку додатку, з новим іменем, а саме «test-app-deployment-6bc5578745-2hpkz». Початковий стан реплік та процес видалення та автоматичного створення нової репліки зображено на рисунку 3.31.

```
root@master1:~# kubectl get po -n test-app -o wide --watch
NAME                                        READY   STATUS      RESTARTS   AGE   IP              NODE    NOMINATED NODE   READINESS GATES
test-app-addrowdb-cronjob-1606579200-76fj6  0/1     Completed   0          26m   10.244.135.19   node3   <none>           <none>
test-app-addrowdb-cronjob-1606580100-7p4tc  0/1     Completed   0          11m   10.244.135.20   node3   <none>           <none>
test-app-deployment-6bc5578745-5r5wn        1/1     Running     0          34m   10.244.166.137  node1   <none>           <none>
test-app-deployment-6bc5578745-mr58p        1/1     Running     0          34m   10.244.104.9    node2   <none>           <none>
test-app-deployment-6bc5578745-mxd97        1/1     Running     0          34m   10.244.135.17   node3   <none>           <none>
test-app-job-f9j4q                          0/1     Completed   0          34m   10.244.135.16   node3   <none>           <none>
test-app-mysql-statefulset-0                1/1     Running     0          38m   10.244.166.136  node1   <none>           <none>
test-app-ngx-deployment-6957f64c9c-jd7x5    1/1     Running     0          33m   10.244.166.138  node1   <none>           <none>
test-app-ngx-deployment-6957f64c9c-llfvn    1/1     Running     0          33m   10.244.135.18   node3   <none>           <none>
test-app-ngx-deployment-6957f64c9c-mfprt    1/1     Running     0          33m   10.244.104.10   node2   <none>           <none>




test-app-deployment-6bc5578745-5r5wn        1/1     Terminating   0          35m   10.244.166.137   node1   <none>           <none>
test-app-deployment-6bc5578745-2hpkz        0/1     Pending       0          0s    <none>           <none>  <none>           <none>
test-app-deployment-6bc5578745-2hpkz        0/1     Pending       0          0s    <none>           node1   <none>           <none>
test-app-deployment-6bc5578745-2hpkz        0/1     ContainerCreating   0          0s    <none>           node1   <none>           <none>
test-app-deployment-6bc5578745-5r5wn        0/1     Terminating         0          35m   <none>           node1   <none>           <none>
test-app-deployment-6bc5578745-2hpkz        0/1     ContainerCreating   0          3s    <none>           node1   <none>           <none>
test-app-deployment-6bc5578745-2hpkz        0/1     Running             0          4s    10.244.166.141   node1   <none>           <none>
test-app-deployment-6bc5578745-5r5wn        0/1     Terminating         0          35m   <none>           node1   <none>           <none>
test-app-deployment-6bc5578745-5r5wn        0/1     Terminating         0          35m   <none>           node1   <none>           <none>
test-app-deployment-6bc5578745-2hpkz        1/1     Running             0          15s   10.244.166.141   node1   <none>           <none>
```

Рисунок 3.31 – Стан реплік додатку до початку тестування та хід роботи Controller Manager

3.4.4 Аналіз результатів

При виконанні аналізу роботи платформи розглядалась не тільки функціональність і якість роботи платформи, а і досліджувалась і порівнювалась з системою-аналогом швидкість розгортування додатку. Результати досліджень порівнювались з системою-аналогом Heroku.

Тестувалась швидкість розгортування додатку на обох платформах. Для цього було розроблено скрипти, які за допомогою API виконують розгортування додатку на обох платформах та вимірюють час, що було затрачене на кожне розгортування.

Виконувалось 25 автоматичних розгортувань на кожній з платформ. Так як кожна з них надає інформацію по часу, який було витрачено на кожний етап розгортування та загальну кількість часу, яку було витрачено на розгортування, то аналізувати такі показники достатньо просто.

Для систематизування та порівняння отриманих результатів тестування, представимо результати у вигляді таблиці:

Таблиця 3.1 – Порівняння швидкості розгортування тестового додатку на платформах

Спроба, №	Власна розробка	Платформа «Heroku»
1	21 с.	24 с.
2	19 с.	26 с.
3	19 с.	26 с.
4	16 с.	23 с.
5	15 с.	25 с.
6	20 с.	25 с.
7	17 с.	28 с.
8	17 с.	24 с.
9	16 с.	26 с.
10	18 с.	26 с.
11	20 с.	27 с.
12	17 с.	24 с.
13	20 с.	23 с.
14	21 с.	25 с.
15	17 с.	25 с.
16	20 с.	27 с.
17	16 с.	26 с.
18	15 с.	24 с.
19	18 с.	24 с.
20	18 с.	22 с.
21	17 с.	24 с.
22	19 с.	25 с.
23	17 с.	26 с.
24	17 с.	24 с.
25	18 с.	23 с.
Середнє значення	17,92 с.	24,88 с.

3.5 Висновок

Під час проектування інформаційної системи побудови платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою було сформульовано вимоги до нової розробки, зокрема, зменшення часу розгортування додатку у порівнянні з аналогами та наявність наступних підсистем: підсистеми управління контейнерами, підсистеми автоматизування встановлення платформи, підсистеми автоматизованого розгортування клієнтських додатків, підсистеми логування та збору і аналізу статистичних даних, підсистеми моніторингу, підсистеми збереження конфігураційних файлів.

Створено UML-діаграму прецедентів для зображення основних можливостей системи. Розроблено UML-діаграму компонентів та загальну структурну схему платформи для представлення підсистем платформи та взаємозв’язків між ними.

Розроблено схеми алгоритмів роботи основних процесів системи, таких як розгортування додатку, контролю реплік додатків та інших.

Під час тестування роботи системи було протестовано, як її автоматичне встановлення, так і розгортування додатків на платформі та підтримку їх роботи. Інформаційна система оптимально працювала відповідно до поставлених перед нею задачами.

При порівнянні з аналогом виконувалось розгортування додатку як на власній розробці, так і на системі-аналогові. Збір показнику часу розгортування однакового додатку на платформах дозволив порівняти їх.

Згідно тестування, час розгортування, у порівнянні з аналогом, зменшився на 28%.

4 ЕКОНОМІЧНА ЧАСТИНА

4.1 Оцінювання комерційного потенціалу розробки

Добре відомо, що час виходу будь-якого продукту на ринок є критичною для багатьох компаній і компанії які займаються розробкою програмного забезпечення не є винятком. Час виходу продукту залежить від багатьох факторів, зокрема, не в останню чергу він залежить від швидкості розгортування додатку на платформі, де додаток буде функціонувати, тестуватись та оновлюватись протягом всього циклу життя програмного забезпечення.

З розвитком комп'ютерів і сучасних інформаційних систем питання проблематики швидкості розгортування систем на платформах стало ще більш актуальним і спричинило активний розвиток та створення нових таких систем.

Однак, як було доведено в попередніх розділах роботи, в існуючих системах існує низка недоліків, які можуть повпливати на зручність та функціональність платформи, а їх висока ціна робить подібні платформи майже недоступними для маленьких та середній компаній.

Тому у виконаній нами магістерській роботі було розв’язане завдання, яке передбачало розробку такої інформаційної системи, яка б підвищувала швидкість розгортування додатку на платформі.

Для цього було досліджено і проаналізовано існуючий програмний інструментарій, за допомогою якого можна реалізувати підсистеми платформи; розроблено схеми алгоритмів роботи вузлів системи; виконано моделювання платформи.

Результати дослідження, дозволили розробити нову інформаційну систему, яка збільшує швидкість розгортування додатку на платформі.

Проведемо технологічний аудит розробленої нами інформаційної системи.

Метою проведення технологічного аудиту є оцінювання комерційного потенціалу розробки. Для проведення технологічного аудиту було залучено 3-х незалежних експертів. Такими експертами будуть Яровий А.А., Арсенюк І.Р та

Сілагін О.В.

Здійснюємо оцінювання комерційного потенціалу розробки за 12-ма критеріями за 5-ти бальною шкалою.

Результати оцінювання комерційного потенціалу розробки наведено в таблиці 4.1.

Таблиця 4.1 – Результати оцінювання комерційного потенціалу розробки

Критерії	Прізвище, ініціали, посада експерта		
	1. Яровий А.А..	2. Арсенюк І.Р	3. Сілагін О.В.
	Бали, виставлені експертами:		
1	4	4	4
2	2	2	3
3	4	4	4
4	4	3	3
5	4	4	4
6	3	3	2
7	3	3	3
8	4	3	4
9	3	3	3
10	4	4	3
11	3	4	4
12	4	4	4
Сума балів	$СБ_1 = 42$	$СБ_2 = 41$	$СБ_3 = 41$
Середньоарифметична сума балів $\overline{СБ}$	$\overline{СБ} = \frac{\sum_{1}^{3} СБ_i}{3} = 41{,}3$		

Отже, з отриманих даних таблиці 4.1 видно, що нова розробка має високий рівень комерційного потенціалу.

4.2 Прогнозування витрат на виконання науково-дослідної роботи та конструкторсько–технологічної роботи

Для розробки нового програмного продукту необхідні такі витрати.

Основна заробітна плата для розробників визначається за формулою (4.1):

$$З_о = \frac{M}{T_p} \cdot t, \qquad (4.1)$$

де М- місячний посадовий оклад конкретного розробника;

T_p - кількість робочих днів у місяці.

t - число днів роботи розробника.

Розрахунки заробітних плат для керівника і програміста наведені в таблиці 4.2.

Таблиця 4.2 – Розрахунки основної заробітної плати

Працівник	Оклад М, грн.	Оплата за робочий день, грн.	Число днів роботи, t	Витрати на оплату праці, грн.
Керівник-архітектор	12100	550	10	5500
Інженер-програміст	8000	363.6	45	16362
Всього:				21862

Розрахуємо додаткову заробітну плату:

$$З_{дод} = 0{,}1 \cdot 21862 = 2186.2 \text{ (грн.)}$$

Нарахування на заробітну плату операторів НЗП розраховується як 22% від суми їхньої основної та додаткової заробітної плати:

$$Н_{зп}= (З_о + З_р)\cdot\frac{\beta}{100}, \qquad (4.2)$$

$$Н_{зп}= (21862+2186.2)\cdot\frac{22}{100} = 5290.6 \text{ (грн.)}$$

Розрахунок амортизаційних витрат для програмного забезпечення виконується за такою формулою:

$$А=\frac{Ц\cdot Н_а}{100}\cdot\frac{Т}{12}, \qquad (4.3)$$

де Ц – балансова вартість обладнання, грн;

$Н_а$ – річна норма амортизаційних відрахувань % (для програмного забезпечення 25%);

Т – Термін використання (Т=3 міс.).

Таблиця 4.3 – Розрахунок амортизаційних відрахувань

Найменування програмного забезпечення	Балансова вартість, грн.	Норма амортизації, %	Термін використання, міс.	Величина амортизаційних відрахувань, грн
Персональний комп’ютер	16000	25	3	1000
Всього:				1000

Розрахуємо витрати на комплектуючі. Витрати на комплектуючі розрахуємо за формулою:

$$K = \sum_{1}^{n} Н_i \cdot Ц_i \cdot К_i, \qquad (4.4)$$

де n – кількість комплектуючих;

Hі - кількість комплектуючих і-го виду;

Ці – покупна ціна комплектуючих і-го виду, грн;

Кі – коефіцієнт транспортних витрат (приймемо Кі = 1,1).

Таблиця 4.4 - Витрати на комплектуючі, що були використані для розробки ПЗ.

Найменування матеріалу	Одиниці виміру	Ціна, грн.	Витрачено	Вартість витрачених матеріалів, грн.
Флешка	шт.	200	1	200
Пачка паперу	уп.	90	1	90
Ручка	шт.	7	3	21
Всього з урахуванням транспортних витрат				342.1

Витрати на силову електроенергію розраховуються за формулою:

$$В_е = В \cdot П \cdot Ф \cdot К_п \; ; \qquad (4.5)$$

де В – вартість 1кВт-години електроенергії (В=2,03 грн/кВт);

П – установлена потужність комп’ютера (П=0,8кВт);

Ф – фактична кількість годин роботи комп’ютера (Ф=440 год.);

$К_п$ – коефіцієнт використання потужності ($К_п < 1$, $К_п = 0{,}68$).

$$В_е = 2{,}03 \cdot 0{,}8 \cdot 440 \cdot 0{,}68 = 485{,}9 \text{ (грн.)}$$

Розрахуємо інші витрати $В_{ін}$.

Інші витрати $І_в$ можна прийняти як (100…300)% від суми основної заробітної плати розробників та робітників, які були виконували дану роботу, тобто:

$$В_{ін} = (1..3) \cdot (З_о + З_р). \quad (4.6)$$

Отже, розрахуємо інші витрати:

$$В_{ін} = 1 \cdot (21862 + 2186{,}2) = 24\,048{,}2 \text{ (грн)}.$$

Сума всіх попередніх статей витрат дає витрати на виконання даної частини роботи:

$$В = З_о + З_д + Н_{зп} + А + К + В_е + І_в$$

$$В = 21862 + 2186{,}2 + 5290{,}6 + 1000 + 342{,}1 + 485{,}9 + 24\,048{,}2 = 55215 \text{ (грн.)}$$

Розрахуємо загальну вартість наукової роботи $В_{заг}$ за формулою:

$$В_{заг} = \frac{В}{\alpha} \quad (4.7)$$

де α – частка витрат, які безпосередньо здійснює виконавець даного етапу роботи, у відн. одиницях = 1.

$$В_{заг} = \frac{55215}{1} = 55215$$

Прогнозування загальних витрат ЗВ на виконання та впровадження результатів виконаної наукової роботи здійснюється за формулою:

$$ЗВ = \frac{B}{\beta} \quad (4.8)$$

де β – коефіцієнт, який характеризує етап (стадію) виконання даної роботи.

Отже, розрахуємо загальні витрати:

$$ЗВ = \frac{55215}{0,9} = 61350 \text{ (грн.)}$$

4.3 Прогнозування комерційних ефектів від реалізації результатів розробки

Економічний ефект від можливого впровадження розробленої нами інтелектуальної системи полягає у тому, що вона дозволяє скоротити час, що затрачується на розгортування додатку на платформі, що в свою чергу дозволяє скоротити час виходу додатку та його оновлень на ринок. На даний ринок ПЗ стрімко розвивається і кожна з компаній хоче бути лідером у своїй ніші, краще задовольняти потреби користувачів, швидше покращувати якість своєї розробки. Тому, скорочення часу до виходу на ринок додатку, і як наслідок, прискорення отримання прибутку, викличе неабияку зацікавленість у компаній, які займаються розробкою та підтримкою ПЗ.

Аналіз місткості ринку показує, що в даний час в Україні та світі кількість потенційних користувачів нашої розробки складає щороку приблизно 130 компаній, але цей попит буде стрімко зростати. Середня ціна подібної системи, що виконує аналогічні функції, дорівнює приблизно 17 тисяч грн (за готову платформу для розробки та підтримки одного додатку). Оскільки розроблена нами інтелектуальна система має кращі функціональні характеристики, то її

можна реалізовувати на ринку дещо дорожче, наприклад в середньому за 20 тис. грн, або на 3 тисячі грн дорожче.

Припустимо, що наша розробка буде користуватися попитом на ринку протягом 4-х років після впровадження.

Якщо наша розробка буде впроваджена з 1 січня 2021 року, то її результати будуть виявлятися протягом 2021-го, 2022-го, 2023-го та 2024-го років. Прогноз зростання попиту на нашу розробку (відносно базового року) складає по роках:

- 1-й рік після впровадження (2021 р.): приблизно + 25 шт.;
- 2-й рік після впровадження (2022 р.): приблизно + 30 шт.;
- 3-й рік після впровадження (2023 р.): приблизно + 40 шт.;
- 4-й рік після впровадження (2024 р.): приблизно + 35 шт.

Розрахуємо очікуване збільшення чистого прибутку $\Delta П_i$, що його може отримати потенційний інвестор від фінансування нашої розробки, для кожного із років, починаючи з першого року впровадження:

$$\Delta П_i = \sum_1^n (\Delta Ц_o \cdot N + Ц_o \cdot \Delta N)_i \cdot \lambda \cdot \rho \cdot (1 - \frac{v}{100}), \qquad (4.9)$$

де $\Delta Ц_o$ – покращення основного якісного показника від впровадження розробки у цьому році. Зазвичай таким показником є збільшення ціни реалізації нової розробки, грн; $\Delta Ц_o$ = 3 тис. грн; N – основний кількісний показник, який визначає обсяг діяльності у цьому році до впровадження розробки; N = 130 шт.; N – покращення основного кількісного показника від впровадження розробки. Таке покращення по роках буде складати: ΔN_{21} = 25 шт.; ΔN_{22} = 30 шт.; ΔN_{23} = 40 шт.: ΔN_{24} = 35 шт.; $Ц_o$ – основний якісний показник (зазвичай, це ціна), який визначає обсяг діяльності у цьому році після впровадження результатів розробки, $Ц_o$ = 20 тис. грн; n – кількість років, протягом яких очікується отримання позитивних результатів від впровадження нашої розробки; n = 4; λ – коефіцієнт, який враховує сплату податку на додану вартість; $\lambda = 0{,}8333$; ρ – коефіцієнт, який враховує рентабельність продукту. Рекомендується приймати

$\rho = 0{,}2...0{,}4$; візьмемо $\rho = 0{,}3$; υ – чинна ставка податку на прибуток. У 2020 році $\upsilon = 18\%$.

Отже, збільшення чистого продукту $\Delta\Pi_1$ протягом першого року для потенційного інвестора складатиме:

$$\Delta\Pi_1 = [3000 \cdot 130 + (17000 + 3000) \cdot 25] \cdot 0{,}8333 \cdot 0{,}3 \cdot \left(1 - \frac{18}{100}\right) = 182442{,}7 \text{ (грн.)}$$

Протягом другого року:

$$\Delta\Pi_2 = [3000 \cdot 130 + (17000 + 3000) \cdot (25 + 30)] \cdot 0{,}8333 \cdot 0{,}3 \cdot \left(1 - \frac{18}{100}\right) = 305437{,}78 \text{ (грн.)}$$

Протягом третього року:

$$\Delta\Pi_3 = [3000 \cdot 130 + (17000 + 3000) \cdot (25 + 30 + 40)] \cdot 0{,}8333 \cdot 0{,}3 \cdot \left(1 - \frac{18}{100}\right) = 469432{,}22 \text{ (грн.)}$$

Протягом четвертого року:

$$\Delta\Pi_4 = [3000 \cdot 130 + (17000 + 3000) \cdot (25 + 30 + 40 + 35)] \cdot 0{,}8333 \cdot 0{,}3 \cdot \left(1 - \frac{18}{100}\right) = 612925{,}48 \text{ (грн.)}$$

4.4 Розрахунок ефективності вкладених інвестицій та період їх окупності

Основними показниками, які визначають доцільність фінансування наукової розробки певним інвестором, є абсолютна і відносна ефективність вкладених інвестицій та термін їх окупності. Розрахунок ефективності вкладених інвестицій передбачає:

1-й крок. Розрахунок теперішньої вартості інвестицій PV, що вкладаються в наукову розробку. Такою вартістю ми можемо вважати прогнозовану величину загальних витрат ЗВ на виконання та впровадження результатів НДДКР.

2-й крок. Розрахунок очікуваного збільшення прибутку ΔПі , що його отримає підприємство (організація) від впровадження результатів наукової розробки, для кожного із років, починаючи з першого року впровадження проведено вище.

3-й крок. Будуємо вісь часу, на якій відображаємо всі платежі (інвестиції та прибутки), що мають місце під час виконання науково-дослідної роботи та впровадження її результатів. Платежі показуємо у ті терміни, коли вони здійснюються. Рух платежів (інвестицій та додаткових прибутків) буде мати вигляд, наведений на рис. 4.1.

Рисунок 4.1 – Вісь часу з фіксацією платежів, що мають місце під час розробки та впровадження результатів НДДКР

Теперішню вартість інвестицій PV, що можуть бути вкладені в розроблену нами інтелектуальну систему, можна розрахувати за формулою:

$$PV = [(1...5) \times ЗВ] , \qquad (4.10)$$

де (1…5) – коефіцієнт, який враховує можливі додаткові витрати інвестора на можливе впровадження нашої розробки (оренда, підготовка персоналу, реклама тощо).

Для нашого випадку отримаємо:

$$PV = (1…5) \times ЗВ = 2 \times 61350 = 122700 \text{ (грн.)}.$$

Далі розраховуємо абсолютний ефект від вкладених інвестицій $Е_{абс}$. Для цього скористаємося формулою (4.11):

$$Е_{абс} = ПП – PV \qquad (4.11)$$

де ПП – приведена вартість всіх можливих чистих прибутків від реалізації розробки, грн;

PV – теперішня вартість інвестицій

Приведена вартість всіх чистих прибутків ПП розраховується за формулою (4.12):

$$ПП = \sum_1^m \frac{\Delta П_i}{(1+\tau)^t}, \qquad (4.12)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої роботи, грн; т – період часу, протягом якого виявляються результати впровадженої розробки, роки; т = 4

роки; τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні. Для України приймемо ставку $\tau = 0{,}12$ (12%); t – період часу (в роках) від моменту отримання прибутків до точки „0". Тоді приведена вартість збільшення всіх чистих прибутків ПП, що їх може отримати потенційний інвестор від можливої реалізації нашої розробки, складе:

$$ПП = \frac{122700}{(1+0{,}1)^0} + \frac{182442{,}7}{(1+0{,}1)^2} + \frac{305437{,}78}{(1+0{,}1)^3} + \frac{469431{,}22}{(1+0{,}1)^4} + \frac{612925{,}48}{(1+0{,}1)^5} = 1204165{,}35 \text{ (грн.)}$$

Тоді розрахуємо $E_{абс}$:

$$E_{абс} = 1204165{,}35 - 122700 = 1081465{,}35 \text{ (грн.)}$$

Оскільки $E_{абс} > 0$, то вкладання коштів на виконання та впровадження результатів НДДКР буде доцільним.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $E_в$ за формулою:

$$E_в = \sqrt[T]{1 + \frac{E_{абс}}{PV}} - 1 \qquad (4.12)$$

де $E_{абс}$ – абсолютна ефективність вкладених інвестицій, тис. грн;

PV – теперішня вартість інвестицій, тис. грн;

T – життєвий цикл наукової розробки, роки.

Тоді будемо мати:

$$E_в = \sqrt[4]{1 + \frac{1081465{,}35}{122700}} - 1 = 1{,}76 - 1 = 0{,}76 \text{ або } 76\ \%$$

Далі, розраховану величина $E_в$ порівнюємо з мінімальною (бар'єрною) ставкою дисконтування $\tau_{мін}$, яка визначає ту мінімальну дохідність, нижче за яку інвестиції вкладатися не будуть. У загальному вигляді мінімальна (бар'єрна) ставка дисконтування $\tau_{мін}$ визначається за формулою:

$$\tau = d + f \qquad (4.13)$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2020 році в Україні d = 0,2;

f – показник, що характеризує ризикованість вкладень, величина f = 0,1.

$$\tau = 0{,}2 + 0{,}1 = 0{,}3$$

Оскільки $E_в$ = 76% $>\tau_{мін}$ = 0,3 = 30%, то інвестор буде зацікавлений вкладати гроші в дану наукову розробку.

Термін окупності вкладених у реалізацію наукового проекту інвестицій $T_{ок}$ розраховується за формулою:

$$T_{ок} = \frac{1}{E_в} \qquad (4.14)$$

Тоді:

$$T_{ок} = \frac{1}{0{,}76} = 1{,}31 \text{ року}$$

Обрахувавши термін окупності даної наукової розробки, можна зробити висновок, що фінансування даної наукової розробки буде доцільним.

4.5 Висновок

В даному розділі було здійснено оцінювання комерційного потенціалу розробки інформаційної технології побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою.

Проведено технологічний аудит з залученням експертів. Аналіз експертних даних показав, що рівень комерційного потенціалу розробки вище середнього. Дослідження комерційного потенціалу розробки підтвердило, що програмний продукт за своїми характеристиками випереджає аналогічні програмні продукти і є перспективною розробкою. Він має кращі функціональні показники, а тому є конкурентоспроможним товаром на ринку.

Згідно із розрахунками всіх статей витрат на виконання науково-дослідної, дослідно-конструкторської та конструкторсько-технологічної роботи загальна вартість витрат на розробку і впровадження складає 61350 грн.

Розрахована абсолютна ефективність вкладених інвестицій в сумі 1081465,35 грн. свідчить про отримання прибутку інвестором від впровадження програмного продукту у діяльність підприємства.

Щорічна ефективність вкладених в наукову розробку інвестицій складає 76%, що вище за мінімальну бар'єрну ставку дисконтування, яка складає 30%. Це означає потенційну зацікавленість інвесторів у фінансуванні розробки.

Термін окупності складає 1,31 року, що також свідчить про доцільність фінансування.

Усе це, узяте разом, забезпечує прийняття рішення про доцільність виготовлення нового продукту.

ВИСНОВКИ

В ході виконання магістерської кваліфікаційної роботи розроблено інформаційну технологію побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою. Під час аналізу архітектури серверних додатків та платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою було розглянуто наявні архітектури побудови серверних додатків, виконано їх порівняння. Проаналізовано платформи для розміщення і підтримки розподілених серверних додатків з мікросервісною архітектурою. Розглянуто і проаналізовано системи-аналоги і визначено їх переваги та недоліки.

В другому розділі магістерської кваліфікаційної роботи виконано аналіз особливостей платформ для розміщення і підтримки серверних додатків з мікросервісною архітектурою та проведено аналіз інструментарію, що використовується для реалізації підсистем платформи. А саме, проведено аналіз програмного інструментарію для контейнерування, для реалізації підсистеми управління контейнерами, для реалізації підсистеми автоматизування встановлення платформи, для реалізації підсистеми автоматизованого розгортування клієнтських додатків, для реалізації підсистеми логування та збору і аналізу статистичних даних, для реалізації підсистеми моніторингу, аналіз сервісів для реалізації підсистеми збереження конфігураційних файлів та аналіз програмного інструментарію необхідного для функціонування серверу для статичних файлів. За результатами аналізу було обрано наступний програмний інструментарій: Docker, Kubernetes, Ansible, Jenkins, Elastic Stack, Prometheus, Grafana, BitBucket, Nginx, ProFTPd. Розглянуто специфічні аспекти побудови платформ на базі оркестратору Kubernetes.

Третій розділ описує процес практичної реалізації інформаційної технології, під час якого проведено проектування інформаційної технології із використанням UML, формування вимог до розробки. Визначено структурну організацію інформаційної технології, розроблено алгоритми роботи основних

вузлів інформаційної технології. Програмно реалізовано інформаційну технологію побудови обчислювальної платформи для розміщення і підтримки серверних додатків з мікросервісною архітектурою. Здійснено тестування програмної реалізації інформаційної технології, під час якого розроблена інформаційна технологія працювала відповідно до поставлених перед нею задач. Виявлено, що нова розробка дозволила зменшити час розгортування додатку, у порівнянні з аналогом, на 28%.

Під час економічного обґрунтування розробки проведено оцінювання економічного потенціалу розробки, проведений прогноз щодо витрат на розробку інформаційної технології, виконання науково-дослідної, дослідно-конструкторської, конструкторсько-технологічних робіт та комерційного ефекту від реалізації результатів розробки, розраховано ефективність вкладених інвестицій та періоду їх окупності. Щорічна ефективність вкладених в наукову розробку інвестицій склала 76%, а термін окупності - 1,31 року.

Результати досліджень було апробовано на XLIX науково-технічній конференції професорсько-викладацького складу, співробітників та студентів Вінницького національного технічного університету у 2020р. За основними результатами досліджень опубліковано одну публікацію та подано заявку про реєстрацію авторського права на твір (комп’ютерну програму) [4].

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Клиент-серверная архитектура в картинках / Хабр [Електронний ресурс] – Режим доступу: https://habr.com/ru/post/495698/.
2. What are microservices? [Електронний ресурс6] – Режим доступу: https://microservices.io/.
3. Как мигрировать нагруженный проект на микросервисы: опыт маркетплейса | DOU [Електронний ресурс] – Режим доступу: https://dou.ua/lenta/articles/migration-microservices/.
4. Ткачук А.С., Яровий А.А. Порівняльний аналіз програмно-апаратних платформ для реалізації інформаційної технології підтримки роботи розподілених додатків. [Електронний ресурс] / Ткачук А.С, Яровий А.А. – Тези доповіді XLIX науково-технічної конференції професорсько-викладацького складу, співробітників та студентів університету з участю працівників науково дослідних організацій та інженерно-технічних працівників підприємств м. Вінниці та області 2020 р. – М-во освіти і науки України, Вінницький нац. техн. ун-т – Режим доступу: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2020/paper/view/8727/7290.
5. Общие архитектуры веб-приложений | Microsoft Docs [Електронний ресурс] – Режим доступу: https://docs.microsoft.com/ru-ru/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures.
6. Монолитные приложения | Microsoft Docs [Електронний ресурс] – Режим доступу: https://docs.microsoft.com/ru-ru/dotnet/architecture/containerized-lifecycle/design-develop-containerized-apps/monolithic-applications.
7. Ричардсон Крис Микросервисы. Паттерны разработки и рефакторинга. — СПб.: Питер, 2019. — 544 с.: ил. — (Серия «Библиотека программиста»). - ISBN 978-5-4461-0996-8.
8. Станут ли микросервисы архитектурой будущего | DOU [Електронний ресурс] – Режим доступу: https://dou.ua/lenta/articles/microservices-for-future/.

9. Who is using microservices? [Електронний ресурс] – Режим доступу: https://microservices.io/articles/whoisusingmicroservices.html.

10. Зачем крупным компаниям микросервисы, контейнеры и Kubernetes: как эти три технологии выводят IT на новый уровень — Сервисы на vc.ru [Електронний ресурс] – Режим доступу: https://vc.ru/services/153410-zachem-krupnym-kompaniyam-mikroservisy-konteynery-i-kubernetes-kak-eti-tri-tehnologii-vyvodyat-it-na-novyy-uroven.

11. Микросервисные паттерны проектирования / Блог компании Издательский дом «Питер» / Хабр [Електронний ресурс] – Режим доступу: https://habr.com/ru/company/piter/blog/275633/.

12. Бёрнс Б. Распределенные системы. Паттерны проектирования. —СПб.: Питер, 2019. —224 с.: ил. —(Серия «Бестселлеры O'Reilly»). ISBN 978-5-4461-0950-0.

13. Что собой представляет Dedicated сервер? | Блог Deltahost [Електронний ресурс] – Режим доступу: https://deltahost.ua/chto-predstavlyaet-soboj-dedicated-server.html.

14. Отличия CaaS, PaaS и IaaS: Как выбрать модель построения облачной инфраструктуры - PCNEWS.RU [Електронний ресурс] – Режим доступу: https://pcnews.ru/blogs/otlicia_caas_paas_i_iaas_kak_vybrat_model_postroenia_oblacnoj_infrastruktury-673835.html.

15. Virtualization - Xen Project [Електронний ресурс] – Режим доступу: https://xenproject.org/users/virtualization/.

16. KVM [Електронний ресурс] – Режим доступу: https://www.linux-kvm.org/page/Main_Page.

17. What is virtualization technology & virtual machine? | VMware [Електронний ресурс] – Режим доступу: https://www.vmware.com/solutions/virtualization.html;

18. Hyper-V Architecture | Microsoft Docs [Електронний ресурс] – Режим доступу: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture.

19. Где размещать контейнеры: на выделенном сервере или на виртуальной машине? / Блог компании HOSTING.cafe / Хабр [Електронний ресурс] – Режим доступу: https://habr.com/ru/company/hosting-cafe/blog/319438/.
20. Что такое контейнеры, как они работают и зачем нужны в Big Data [Електронний ресурс] – Режим доступу: https://www.bigdataschool.ru/blog/containers-big-data.html.
21. Какие известные компании используют Docker в production и для чего? - Блог компании Флант – Хабр[Електронний ресурс] – Режим доступу: https://habr.com/ru/company/flant/blog/326784/.
22. Customers – Docker [Електронний ресурс] – Режим доступу: https://www.docker.com/customers.
23. Оркестрация контейнеров | Xelent [Електронний ресурс] – Режим доступу: https://www.xelent.ru/blog/chto-takoe-orkestratsiya-konteynerov/.
24. Что такое PaaS? Платформа как услуга | Microsoft Azure [Електронний ресурс] – Режим доступу: https://azure.microsoft.com/ru-ru/overview/what-is-paas/;
25. What is OpenShift - Red Hat OpenShift [Електронний ресурс] – Режим доступу: https://www.openshift.com/learn/what-is-openshift.
26. What is Heroku | Heroku [Електронний ресурс] – Режим доступу: https://www.heroku.com/what.
27. Облачные сервисы PaaS — для кого и зачем – Market.CNews [Електронний ресурс] – Режим доступу: https://market.cnews.ru/news/top/2019-11-21_oblachnye_servisy_paas_dlya.
28. Что такое DevOps?. Цель DevOps | by Grigory Pryalukhin | Medium [Електронний ресурс] – Режим доступу: https://medium.com/@pryalukhin/what-is-devops-37b365e7fee0.
29. Time To Market — Что Значит Этот Показатель [Електронний ресурс] – Режим доступу: https://leadstartup.ru/db/time-to-market.
30. Что такое контейнер? Обзор технологии контейнеризации [Електронний ресурс] – Режим доступу: https://www.cloud4y.ru/about/news/obzor-tekhnologii-konteynerizatsii/.

31. What is a Container? | Docker [Електронний ресурс] – Режим доступу: https://www.docker.com/resources/what-container.

32. Introducing the Docker Index: Insight from the World's Most Popular Container Registry - Docker Blog [Електронний ресурс] – Режим доступу: https://www.docker.com/blog/introducing-the-docker-index/.

33. Podman | podman.io [Електронний ресурс] – Режим доступу: https://podman.io/.

34. How does rootless Podman work? | Opensource.com [Електронний ресурс] – Режим доступу: https://opensource.com/article/19/2/how-does-rootless-podman-work.

35. Check Out Podman, Red Hat's daemon-less Docker Alternative – The New Stack [Електронний ресурс] – Режим доступу: https://thenewstack.io/check-out-podman-red-hats-daemon-less-docker-alternative/.

36. Docker Engine release notes | Docker Documentation [Електронний ресурс] – Режим доступу: https://docs.docker.com/engine/release-notes/prior-releases/#010-2013-03-23.

37. Сайфан Джиджи Осваиваем Kubernetes. Оркестрация контейнерных архитектур. — СПб.: Питер, 2019. — 400 с.: ил. — (Серия «Для профессионалов»). ISBN 978-5-4461-0973-9.

38. Система управления Ansible / Блог компании Selectel / Хабр [Електронний ресурс] – Режим доступу: https://habr.com/ru/company/selectel/blog/196620/.

39. Open source projects | Puppet | Puppet [Електронний ресурс] – Режим доступу: https://puppet.com/open-source/#osp.

40. Ansible is Simple IT Automation [Електронний ресурс] – Режим доступу: https://www.ansible.com/.

41. Хохштейн Л., Мозер Р. Запускаем Ansible / пер. с анг. Е. В. Филонова , А. Н. Киселева. – М.: ДМК Пресс, 2018. – 382 с.: ил. ISBN 978-5-97060-513-4.

42. Pierluigi Riti Pro DevOps with Google Cloud Platform: With Docker, Jenkins, and Kubernetes – 2018 – ISBN-13 (electronic): 978-1-4842-3897-4.

43. TeamCity vs. Jenkins: Picking The Right CI/CD Tool [Електронний ресурс] – Режим доступу: https://www.lambdatest.com/blog/teamcity-vs-jenkins-picking-the-right-ci-cd-tool/.

44. TeamCity vs Jenkins for Continuous Integration [2020 Update] [Електронний ресурс] – Режим доступу: https://www.upguard.com/blog/teamcity-vs-jenkins-for-continuous-integration.

45. Пранав Шукла, Шарат Кумар Elasticsearch, Kibana, Logstash и поисковые системы нового поколения . — СПб.: Питер, 2019. ISBN 978-5-4461-1024-7.

46. Log Management: Graylog vs ELK. Whether you are a developer or a DevOps | by JetRuby Agency | JetRuby [Електронний ресурс] – Режим доступу: https://expertise.jetruby.com/log-management-graylog-vs-elk-d6e8f0492323.

47. Log Management Comparison: ELK vs Graylog – Coralogix – Smarter Log Analytics [Електронний ресурс] – Режим доступу: https://coralogix.com/log-analytics-blog/log-management-comparison-elk-vs-graylog/.

48. Mike Julian Practical Monitoring — Published by O'Reilly Media, Inc — 2018. ISBN 978-1-491-95735-6.

49. Munin [Електронний ресурс] – Режим доступу: http://munin-monitoring.org/.

50. Что такое система мониторинга Zabbix и как её использовать [Електронний ресурс] – Режим доступу: https://eternalhost.net/blog/sistemnoe-administrirovanie/zabbix-chto-eto.

51. Brian Brazil Prometheus: Up & Running — Published by O'Reilly Media, Inc — 2018. ISBN 978-1-492-03414-8.

52. Чакон С., Штрауб Б. Git для профессионального программиста. — СПб.: Питер, 2016. — 496 с.: ил. — (Серия «Библиотека программиста»).ISBN 978-5-496-01763-3.

53. 8 reasons Bitbucket is better than GitHub [the ultimate 2020 smackdown] [Електронний ресурс] – Режим доступу: https://www.idalko.com/bitbucket-vs-github/.

54. OpenNET: статья - Настройка nginx в качестве front-end к apache (nginx apache web) [Електронний ресурс] – Режим доступу: https://www.opennet.ru/base/net/nginx_frontend_apache.txt.html.

55. Веб-сервер — Национальная библиотека им. Н. Э. Баумана [Електронний ресурс] – Режим доступу: https://ru.bmstu.wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80.

56. Настройка FTP-сервера на примере Ubuntu Linux | Xelent [Електронний ресурс] – Режим доступу: https://www.xelent.ru/blog/nastroyka-ftp-servera-na-primere-ubuntu-linux/.

57. Немет Эви, Снайдер Гарт, Хейн Трент, Уэйли Бен , Макни Дэн Unix и Linux: руководство системною администратора, 5-е изд.: Пер. с англ. - СПб. : ООО "Диалектика", 2020. - 1 168 с. : ил. - Парал. тит. англ. ISBN 978-5-907 1 44- 1 0- 1 (рус.).

58. Марко Лукша Kubernetes в действии / пер. с англ. А. В. Логунов. – М.: ДМК Пресс, 2019. – 672 с.: ил. ISBN 978-5-97060-657-5.

59. Арундел Джон, Домингус Джастин Kubernetes для DevOps: развертывание, запуск и масштабирование в облаке. — СПб.: Питер, 2020. — 384 с.: ил. — (Серия «Бестселлеры O'Reilly»). ISBN 978-5-4461-1602-7.

60. Persistent Volumes | Kubernetes [Електронний ресурс] – Режим доступу: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

61. Роїк О.М., Савчук Т.О. – Технології комп'ютерного проектування – 112 с.

62. Лекція 6. Основи UML – проектування розподілених систем [Електронний ресурс]. – Режим доступу: http://moodle.ipo.kpi.ua/moodle/mod/resource/view.php?inpopup=true&id=44416.

63. UML2 и ER диаграммы [Електронний ресурс]. – Режим доступу: https://proft.me/2013/05/27/uml-2-tipy-diagramm/#component.

64. Helm [Електронний ресурс]. – Режим доступу: https://helm.sh/.

65. Best Practices — Ansible Documentation [Електронний ресурс]. – Режим доступу: https://docs.ansible.com/ansible/2.3/playbooks_best_practices.html.