



Вінницький національний технічний університет  
Факультет інформаційних технологій та комп'ютерної інженерії  
Кафедра програмного забезпечення  
Рівень вищої освіти II-й (магістерський)  
Галузь знань 12 – Інформаційні технології  
Спеціальність 121 – Інженерія програмного забезпечення  
Освітньо-професійна програма – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ  
Завідувач кафедри ПЗ  
Романюк О. Н.  
« 13 » вересня 2021 р.

## **З А В Д А Н Н Я НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ**

Мельнику Денису Олександровичу

1. Тема роботи – розробка методу та програмного забезпечення у середовищі Kubernetes для інтегрування програмних продуктів.

Керівник роботи: Кательніков Денис Іванович, к.т.н., доцент кафедри ПЗ, затверджені наказом вищого навчального закладу від « 24 » вересня 2021 р. № 277.

2. Строк подання студентом роботи

1 грудня 2021 р.

3. Вихідні дані до роботи: розподілена архітектура програмної системи; методи зчитування результату виконання кроків; алгоритм виконання пайплайну у кластері користувача; алгоритм безпечної взаємодії серверу та агента; алгоритм логування вихідних даних контейнерів; програмне забезпечення у середовищі Kubernetes для інтегрування програмних продуктів;

4. Зміст розрахунково-пояснювальної записки: вступ; аналіз сучасного стану питання та обґрунтування задачі; розробка методу інтегрування програмних продуктів у середовищі kubernetes; розробка програмного додатку; тестування програми; економічна частина; висновки; перелік посилань; додатки.

5. Перелік графічного матеріалу: галузі застосування графічних засобів; основні етапи графічного конвеєру; адаптивний зафарбування; нові моделі відбивних здатностей поверхонь; інтерфейс 3D – редактора світлових ефектів.

#### 6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Кательніков Д. І., к.т.н., доцент кафедри ПЗ		
5	Ратушняк О.Г., к.т.н., доцент кафедри ЕПВМ		

7. Дата видачі завдання \_\_\_\_\_ 14 вересня 2021 р. \_\_\_\_\_

### КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської кваліфікаційної Роботи	Строк виконання етапів роботи	Примітка
1	Аналіз, вибір та обґрунтування актуальності розробки	15.09. 2021 – 25.09.2021	Вик.
2	Аналіз існуючих аналогів	26.09. 2021 – 30.09.2021	Вик.
3	Розробка методів та алгоритмів	01.10. 2021 – 20.10.2021	Вик.
4	Розробка серверної частини	20.10. 2021 – 05.11.2021	Вик.
5	Розробка клієнтської частини	06.11. 2021 – 14.11.2021	Вик.
6	Економічна частина	15.11. 2021 – 25.11.2021	Вик.
7	Оформлення матеріалів до захисту МКР	26.11.2021 - 30.11.2021	Вик.

Студент \_\_\_\_\_ **Мельник Д. О.**  
( підпис ) (прізвище та ініціали)

Керівник магістерської кваліфікаційної роботи \_\_\_\_\_ **Кательніков Д. І.**  
( підпис ) (прізвище та ініціали)

## АНОТАЦІЯ

УДК 004.912.032.26

Мельник Д.О. розробка методу та програмного забезпечення у середовищі Kubernetes для інтегрування програмних продуктів. Магістерська кваліфікаційна робота зі спеціальності 121 – інженерія програмного забезпечення, освітня програма – інженерія програмного забезпечення. Вінниця: ВНТУ, 2021. 128 с.

На укр. мові. Бібліогр.: 23 назв; рис.: 46; табл. 12.

У магістерській кваліфікаційній роботі розроблено метод та програмне забезпечення у середовищі Kubernetes для інтегрування програмних продуктів. Сформульовано мету досліджень – підвищення безпеки в роботі CI/CD рішень за рахунок використання Kubernetes як середовища виконання пайплайнів.

Проведено аналіз існуючих рішень і засобів з автоматизації інтеграції та розгортання програмного коду.

Розроблено алгоритм взаємодії серверу та агенту встановленого у Kubernetes кластері, алгоритм виконання пайплайну в середовищі Kubernetes, алгоритм збереження вихідних даних кроків. Також попередньо було розроблено архітектурну модель додатку. В результаті розробки було створено розподілену програмну систему. Тестування програмної системи показало, що вона працює коректно й відповідає поставленим завданням.

Результати магістерської кваліфікаційної роботи можна використати для побудови високопродуктивних програмних систем безперервної інтеграції з підвищеним рівнем безпеки.

Ключові слова: CI/CD, веб-технології, Kubernetes, логування вихідних даних контейнерів.

## ABSTRACT

In this master's qualification work developed a method and software in the Kubernetes environment for software integration. The purpose of the research is formulated - to increase the security of CI / CD solutions by using Kubernetes as a environment for executing pipelines.

An analysis of existing solutions and tools for automating the integration and deployment of software code.

The algorithm of interaction of the server and the agent installed in the Kubernetes cluster, the algorithm of execution of a pipeline in the Kubernetes environment, the algorithm of preservation of the initial data of steps are developed. An architectural model of the application was also previously developed. As a result of development, a distributed software system was created. Testing of the software system showed that it works correctly and meets the objectives.

The method of continuous integration processes has been improved in such a way that it allows not to disclose confidential data to the platform during the implementation of the pipeline, but to store them in its own cluster, which significantly reduces the risk of unauthorized distribution.

The results of the master's qualification work can be used to build high-performance software systems for continuous integration with a high level of security.

## ЗМІСТ

ВСТУП.....	8
1 АНАЛІЗ СУЧАСНОГО СТАНУ ПИТАННЯ ТА ОБҐРУНТУВАННЯ ЗАДАЧІ.....	13
1.1 Аналіз стану питання .....	13
1.2 Аналоги та їх основні характеристики.....	15
1.3 Аналіз методів виконання кроків у середовищі Kubernetes.....	21
1.4 Аналіз методів зчитування вихідних даних кроків .....	23
1.5 Постановка задачі.....	25
1.6 Висновки.....	25
2 РОЗРОБКА МЕТОДУ ІНТЕГРУВАННЯ ПРОГРАМНИХ ПРОДУКТІВ У СЕРЕДОВИЩІ KUBERNETES .....	26
2.1 Аналіз інформаційного забезпечення .....	26
2.2 Розробка розподіленої архітектури програмної системи.....	27
2.3 Розробка алгоритму взаємодії сервера й Kubernetes кластера.....	29
2.4 Розробка алгоритму виконання пайплайну в середовищі Kubernetes .....	31
2.5 Розробка методу зчитування вихідних даних контейнерів.....	33
2.6 Розробка структури графічного інтерфейсу .....	37
2.7 Висновки.....	38
3 РОЗРОБКА ПРОГРАМНОГО ДОДАТКУ .....	39
3.1 Варіантний аналіз і обґрунтування вибору засобів реалізації програмного продукту. ....	39
3.1.1 Робоче оточення, сервер та контейнеризація .....	39
3.1.2 Вибір мови програмування для розробки серверної частини.....	40
3.1.3 Вибір мови програмування для розробки агенту .....	42
3.1.4 Вибір середовища розробки .....	44
3.1.5 Вибір бази даних .....	46
3.1.6 Вибір зворотнього-проксі сервера.....	48
3.2 Розробка графічного інтерфейсу користувача.....	49
3.3 Розробка модуля прослуховування та логування вихідних даних кроків .....	52
3.4 Розробка модуля виконання пайплайну в середовищі Kubernetes .....	57
3.5 Розробка серверного додатку .....	64
3.6 Висновки.....	70
4 ТЕСТУВАННЯ ПРОГРАМИ.....	71
4.1 Тестування програмного забезпечення .....	71
4.2 Розробка інструкції користувача.....	80
4.3 Висновки.....	81
5 ЕКОНОМІЧНА ЧАСТИНА .....	82
5.1 Оцінювання комерційного потенціалу розробки .....	82
5.2 Прогнозування витрат на виконання науково-дослідної роботи.....	88
5.3 Розрахунок економічної ефективності науково-технічної розробки .....	94
ВИСНОВКИ .....	99

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	100
ДОДАТКИ.....	102
Додаток А – Технічне завдання.....	103
Додаток Б – Протокол перевірки .....	107
Додаток В – Лістинг коду.....	108
Додаток Г – Ілюстративна частина .....	119

## ВСТУП

**Актуальність роботи.** На сьогоднішній день швидкість з якою розробляються програмні продукти стає все більшою й більшою. Разом з цим зростає кількість проектів, ідей, які потребують якнайшвидшого вирішення. Є багато методологій й підходів до розробки програмного забезпечення, які намагаються вирішити ці проблеми, але часто проблема так і залишається на стороні розробки, а не її планування. Одним з найважливіших етапів розробки програмного продукту є розгортання програмного забезпечення. Це операції, які роблять програмну систему готовою до використання. Цей процес є частиною життєвого циклу програмного забезпечення.

Загальний процес розгортання складається з декількох взаємопов'язаних дій з можливими переходами між ними. Ці дії можуть відбуватися на стороні розробника або на стороні споживача або обох. Оскільки кожна програмна система є унікальною, точні процеси чи процедури в межах кожної діяльності важко визначити. Тому "розгортання" слід тлумачити як загальний процес, який повинен бути налаштований відповідно до конкретних вимог або характеристик.

Етап розгортки займає велику частину часу відносно безпосередньої розробки, що передбачає необхідність оптимізації. Найпростішим рішенням було створення скриптів, які в якійсь мірі автоматизували цей процес: не потрібно вручну збирати проекти, забезпечувати переміщення між машинами, видаляти попередні версії та встановлювати нові. Всі перелічені дії можна оптимізувати за допомогою звичайних скриптів. Виявилася необхідність в особі, хто буде займатися запуском та налаштуванням цим скриптів. Після цього прийшла черга автоматизувати процес запуску скриптів, а саме визначити умови за яких це має ставатися та вхідні дані. Таким чином були створені перші CI/CD рішення.



CI/CD (continuous integration/ continuous deployment, безперервна інтеграцій/безперервне розгортання) – процес постійної інтеграції та розгортання [1].

Безперервну інтеграцію можна підсумувати так: ви хочете, щоб усі частини програмного засобу, що розробляється, надходили до одного й того ж самого місця, й проганялися через ті самі процеси в результаті чого був отриманий звіт, опублікованими у легкодоступному місці.

Найпростіший приклад безперервної інтеграції - пуш всього коду програми в єдиний репозиторій. Незважаючи на все це, це може здатися незручним, мати єдине місце, де ви «інтегруєте» весь свій код, але це є основою для розширення інших, більш досконалих практик.

Після того, як код прибув до репозиторія, є можливість запустити деякі процеси в цьому сховищі кожного разу, коли щось змінюється. Це може включати:

- запуск автоматичного сканування якості коду з генеруванням звіту про те, наскільки добре останні зміни дотримуються належних методів кодування;
- збирання проекту й запуск усіх можливих автоматизованих тестів, щоб переконатися, що ваші зміни не порушили жодної функціональності;
- створення та публікація звіту про покриття коду тестами, щоб отримати уявлення про те, наскільки ретельними є автоматизовані тести.

Але побудування процесу безперервної інтеграції задача не із легких. Її організацією як правило займаються виключно фахівці в сфері DevOps. Організація відбувається у декілька етапів:

- отримання початкового коду з репозиторію;
- складання проекту;
- виконання тестів;
- розгортання готового проекту;
- відправлення звітів.

Для упорядкування й організації всіх кроків перелічених вище їх об'єднують в «пайплайни»(від англ. pipeline – трубопровід). Пайплайни складаються з кроків. Кожен крок являється виконанням вказаної користувачем команди в зазначеному ним середовищі. Для того що надати такий рівень свободи дій користувачу, в системах безперервної інтеграції використовуються віртуальні машини. Тобто кожен крок – це інструкція того, що буде зроблено в середині віртуальної машини. Але оскільки віртуальна машина це досить «важка» річ й потребує велику кількість ресурсів комп'ютера, розробники вирішили змінити їх контейнерами – полегшеною версією віртуальних машин, яка працює на основі ядра Linux [2].

Оскільки процес створення CI/CD рішень є важкою, до кінця не дослідженою й не тривіальною задачею було вирішено дослідити це питання й розробити свій прототип системи інтеграції програмного продукту в режимі реального часу.

**Мета та завдання дослідження.** Метою роботи є підвищення рівня безпеки в роботі CI/CD рішень за рахунок використання Kubernetes як середовища виконання пайплайнів.

Основними задачами дослідження є:

- провести аналіз існуючих рішень і засобів з автоматизації інтеграції та розгортання програмного коду;
- розробити розгалужену архітектуру програмної системи;
- розробити алгоритм взаємодії сервера й Kubernetes кластера;
- розробити алгоритм виконання пайплайну в середовищі Kubernetes;
- розробити алгоритм збереження вихідних даних кроків;
- розробити інтерфейс програмного продукту;
- розробити систему інтеграції програмного продукту в режимі реально часу.

**Об'єкт дослідження** – процес безперервної інтеграції та розгортання програмного забезпечення.

**Предмет дослідження** – методи та засоби виконання безпечних CI/CD процесів у середовищі Kubernetes.

**Методи дослідження.** У процесі досліджень використовувались: методи проектування архітектури програмного забезпечення, методи побудови розподілених інформаційних систем.

**Наукова новизна отриманих результатів.**

1. Подальшого розвитку отримав метод виконання процесів безперервної інтеграції та розгортання за допомогою середовища виконання Kubernetes, який на відміну від існуючих передбачає використання додатковий модуля-агента, який встановлюється у кластер користувача та підтримує з'єднання з CI/CD платформою шляхом постійного опитування на предмет наявності нових «задач», які повинні бути виконані у кластері користувача. В результаті, з'являється можливість для користувача з власним Kubernetes кластером використовувати його у якості середовища виконання, навіть, якщо цей кластер знаходиться у приватній мережі, на відміну від аналогів які мають прямий доступ до середовища виконання. Як наслідок, підвищується рівень безпеки в роботі CI/CD рішень, оскільки з'являється можливість не розголошувати секрети платформи під час виконання пайплайну, а зберігати їх у власному кластері, що значно зменшує ризики їх несанкціонованого поширення.

2. Подальшого розвитку отримав метод зчитування вихідних даних контейнерів, який відрізняється від відомих у можливості роботи з багатьма контейнерами одночасно й створенню мультиплексованого потоку даних, що дозволяє протоколювати вихідні дані контейнерів, запущених за допомогою команди `docker-compose`, в один вихідний потік.

**Практична цінність отриманих результатів.** Практична цінність одержаних результатів полягає в тому, що на основі розробленої в магістерській кваліфікаційній роботі методу інтегрування програмних

продуктів у середовищі Kubernetes було розроблено програмне забезпечення безпечного виконання пайплайнів.

**Особистий внесок здобувача.** Усі наукові результати, викладені у магістерській кваліфікаційній роботі, отримані автором особисто. У роботі автору належить ідея та реалізації CI/CD платформи з використанням Kubernetes як середовища виконання пайплайнів, що дозволить покращити продуктивність, безпеку та моніторинг процесів що відбуваються під час виконання пайплайну.

**Апробація матеріалів.** Основні положення й результати досліджень представлені на науково-практичній конференції «Електронні інформаційні ресурси: створення, використання, доступ - 2021» та міжнародній науково-практичній конференції «Інформаційні технології і автоматизація - 2021».

**Структура та обсяг роботи.** Магістерська кваліфікаційна робота містить вступ, п'ять розділів, висновки, список літератури, що містить 23 найменування, та 4 додатки. Робота містить 46 рисунків та 12 таблиць. Ілюстративний матеріал до роботи подано у додатку Г.

# 1 АНАЛІЗ СУЧАСНОГО СТАНУ ПИТАННЯ ТА ОБҐРУНТУВАННЯ ЗАДАЧІ

## 1.1 Аналіз стану питання

В даний час у розробці веб-додатків як підтипу програмного забезпечення існує чітка тенденція до прискорення циклу розробки: кодування - тестування - збирання - розгортання. Кінцевим результатом циклу є новий випуск програмного забезпечення. Дуже часто команді розробників доводиться випускати нові релізи кілька разів на день. У свою чергу, всі описані в циклі процеси займають відносно багато часу та є досить рутинними. Крім того, ми можемо згадати людський фактор, і незначне відхилення від процедури може призвести до помилок у новому релізі програмного продукту, на виявлення причини якого згодом буде витрачено ще більше часу.

Ця проблема може бути вирішена шляхом введення безперервної інтеграції (CI) у процес розробки. Це практика автоматичного виконання етапів збирання й розгортання необмежену кількість раз, що прагне до однакових результатів з періодичним виконанням.

На сьогоднішній день CI / CD є найбільш широко використовуваної технологією в розробці програмного забезпечення, і розробники прагнуть застосовувати її практично у всіх завданнях. Все, що потрібно – це довести існування статусу «золотого стандарту» і «водоспаду».

Методологія почала активно використовуватися паралельно з повсюдним поширенням гаджетів, коли кількість всіх видів програмного забезпечення для них значно збільшилася. Кожен день, якщо не кожну годину, створюються різні клієнтські програми. Подивіться на динаміку росту кількості додатків на платформах для дистрибутиву за останні кілька років. В 2018 році кількість додатків в Google Play збільшилося на третину, склавши 3,6 мільйона. Все це не могло не привести до зміни глобальних критеріїв швидкості створення програмного забезпечення як такого.

Наразі вже існує ціла низка рішень для впровадження методології постійної інтеграції. Їх є багато видів, починаючи від найпримітивніших і закінчуючи прогресивними сучасними платформами з великою кількістю можливостей. На ринку CI/CD рішень наразі панує величезна конкуренція, велика кількість застарілих рішень намагаються втримати клієнтів й не надавати їм причин дивитися в сторону більш сучасних платформ. Останні роки є особливо визначними для побільшої долі такого роду програмних засобів, адже стандартом «де-факто» для production оточення стає запуск додатків в контейнерах. А контейнери в свою чергу повинні існувати в межах інструменту для їх оркеструванням, найпопулярніший на сьогоднішній день є Kubernetes[3][4].

Що важливо для користувача й для чого це взагалі йому потрібно? Йому важливо щоб використання інструменту було зручним і не поступалося у функціональності вже доступним рішенням. А необхідність в такого роду рішень набуло попиту з розвитком контейнеризації та CI/CD.

Є багато готових рішень (CircleCI, TravisCI, DroneIO, Gitlab, Codefresh), які можуть покрити низку проблем даної галузі, але їх найбільша проблема – це ціна. Тому розробники все частіше намагаються створити щось «своє» з метою заміни, вже існуючих рішень. Також деякі з них є досить повільними через використання вже застарілих віртуальних машин в якості ізольованого середовища виконання кроків.

Створення такого роду платформи це не легка задача, це ціла низка надзвичайно складних програмних компонент, які мають працювати разом. Одною із найважчих в реалізації компонент, є логування вихідних даних контейнера. На разі не було знайдено компоненти/модуля який би зміг усунути цю складність під час розробки.

Розроблювана програмна система зможе усунути цю дірку, покращити швидкість виконання пайплайнів та надати ентузіастам зручний шаблон й

інтуїтивно зрозумілий інтерфейс для подальшого покращення й створення більш складніших рішень.

Головною перевагою програмної системи буде швидкість в роботі, можливість запускати пайплайни, прослуховувати вихідних дані контейнерів, та показувати їх користувачу.

## 1.2 Аналоги та їх основні характеристики

Розглянемо декілька інструментів інтеграції програмного продукту в режимі реального часу, які були, або залишилися популярними серед розробників.

Автоматизація шляхом написання скриптів (рисунок 1.1) – найпримітивніший з перелічених способів й інструментів. Раніше автоматизація процесів розгортання була виконана у досить прямий спосіб. Програміст писав код, який в якійсь мірі міг автоматизувати процес розгортання, але дане рішення залишалось досить проблемним довгий час.

На це є декілька причин:

- зав'язаність на спеціалісті, який автоматизував процес розгортання. Часто така автоматизація є не легкою задачею й стає проблемно розібратися у всіх рішеннях прийнятих попереднім програмістом. Іншим словом це можна перефразувати у складність в імпліmentaції;
- необхідність адаптації скриптів відповідно до проекту й оточення в якому відбувається його розгортання. У випадку з написання скриптів автоматизації власноруч, дуже важко уникнути ситуації коли доведеться модифікувати й адаптувати його під певну ситуацію;
- відсутність екосистеми з низкою додаткових функцій, які надаються «з коробки», як у випадку з повноцінними CI/CD платформами. Мова йдеться про такі життєво необхідні на сьогоднішній день механізми як відмовостійкість, безпечність, інтеграції з зовнішніми сервісами.

```

run.sh x
12 sshgit="git@github.com"
13 username=""
14 clonebranch=""
15 updatebranch=""
16
17 # imports
18 source $LIST
19 source $UTILS
20
21 # functions
22 clone_all () {
23     for i in "${repos[@]}"
24     do
25         echo "cloning $sshbase/$i.git"
26         gitoutput=$(git clone $sshbase/$i.git 2>&1)
27         echo $gitoutput
28         if echo $gitoutput | grep -q "Could not read from remote
29         repository"; then
30             open_url $github/$i
31         else
32             if [ -d "$i" ]; then
33                 cd $i
34                 git remote remove origin
35                 git remote add base $sshbase/$i.git
36             fi
37         fi
38     done
39 }
40
41 urlbat x
1 @echo off
2 rem version 1.0
3 rem sithum 20190111
4
5 start %1

```

Рисунок 1.1 – Приклад власноруч написаного скрипта автоматизації розгортання

Hudson (рисунок 1.2) – це система для реалізації безперервної інтеграцією (CI), написана на Java, яка працює на сервлет контейнерах, таких як Apache Tomcat або Glassfish. Він підтримує інструменти SCM, включаючи CVS, Subversion, Git, Perforce, Clear Case і RTC, і може реалізовувати проекти на основі Apache Ant і Apache Maven, а також довільні сценарії оболонки і пакетні команди Windows [5].

Hudson може бути розширений за допомогою архітектури плагінів. Багато плагінів стали загальнодоступними, розширивши його за межі просто інструменту побудови проектів Java. Плагіни доступні для інтеграції з Hudson в більшій мірі реалізують інтеграції з системами контролю версій та базами даних. Багато інструментів побудови підтримуються за допомогою відповідних плагінів. Плагіни також можуть змінювати вигляд та поведінку Хадсона або додавати нові функції.

Hudson є більш сучаснішим інструментом, але не позбавленого проблем:



- Налаштування має бути більш зручним для користувачів. На даний момент користувачі, які вперше користуються системою не розуміють як розпочати роботу.
- Відсутність підтримки від розробників

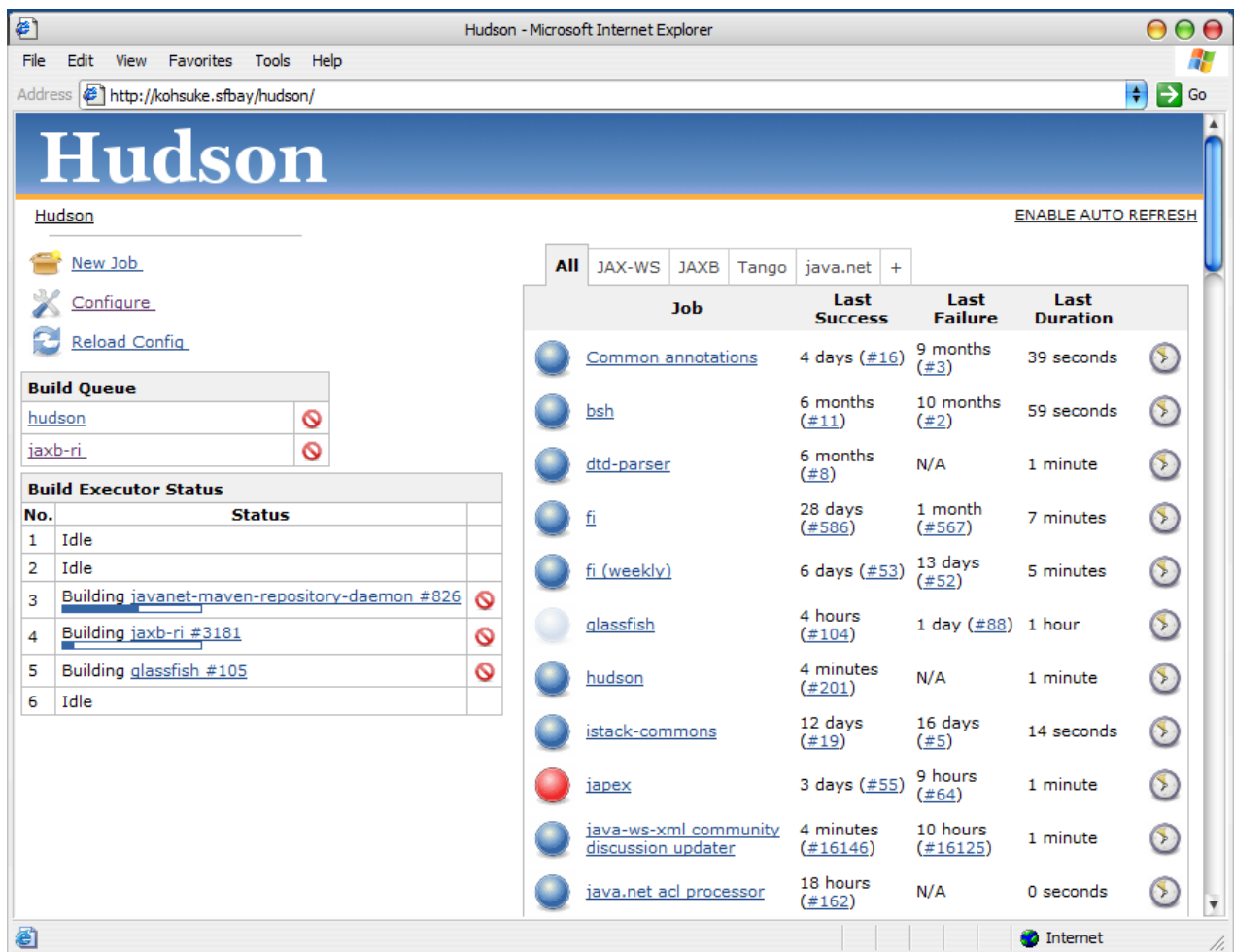


Рисунок 1.2 – Приклад використання Hudson

Jenkins (рисунок 1.3) – це сервер автоматизації з відкритим кодом з неперевершеною екосистемою плагінів, який підтримує практично кожен інструмент як частину пайплайнів [6]. Він є нащадком Hudson.

Jenkins пропонує простий спосіб налаштування середовища безперервної інтеграції або безперервної розгортки (CI / CD) для майже будь-якої комбінації мов та сховищ вихідного коду за допомогою конвеєрів, а також автоматизацію

інших завдань рутинної розробки. У той час як Jenkins не усуває необхідності створювати сценарії для окремих кроків, це дає швидший і надійніший спосіб інтегрувати весь ланцюжок інструментів побудови, тестування та розгортання, що ви можете легко створити самостійно.

Даний інструмент поширюється як архів WAR і як інсталяційний пакет для основних операційних систем: як пакет Homebrew, зображення Docker та вихідний код. Вихідний код здебільшого Java, з кількома файлами Groovy, Ruby та Antlr.

Jenkins досягає високої якості та зручності безперервної інтеграції за допомогою плагінів. Плагіни дозволяють інтегрувати різні етапи DevOps. Якщо ви хочете інтегрувати певний інструмент, вам потрібно встановити їх плагіни. Наприклад: Git, Maven, Amazon EC2, середовища виконання тестів, тощо.

Недоліки Jenkins:

- проблеми з повідомленнями про помилки;
- важке початкове налаштування;
- незручний інтерфейс;
- незручна документація.

The screenshot shows the Jenkins web interface. At the top, there is a search bar and user information (admin | log out). Below the search bar, there is a navigation menu on the left with options like 'New Item', 'People', 'Build History', 'Manage Jenkins', 'My Views', 'Credentials', 'Lockable Resources', and 'New View'. The main content area displays a table of builds with columns for 'S', 'W', 'Name', 'Last Success', 'Last Failure', and 'Last Duration'. The table lists four builds: 'Freestyle project', 'Github Org project', 'Multibranch', and 'Other'. Below the table, there are links for 'Legend', 'RSS for all', 'RSS for failures', and 'RSS for just latest builds'. At the bottom of the page, there is a footer with the text 'Page generated: Jan 9, 2019 9:22:23 PM GMT REST API Jenkins ver. 2.157'.

S	W	Name	Last Success	Last Failure	Last Duration
		<a href="#">Freestyle project</a>	29 sec - #1	N/A	41 ms
		<a href="#">Github Org project</a>	N/A	20 sec - log	2.9 sec
		<a href="#">Multibranch</a>	N/A	N/A	N/A
		<a href="#">Other</a>	N/A	N/A	N/A

Рисунок 1.3 – Приклад використання Jenkins

CircleCI (рисунок 1.4) – це потужний CI-продукт на основі SaaS, який дозволяє інтеграцію та розгортання у хмарі. Система конфігурації на основі YAML дозволяє окремим розробникам випробовувати свої зміни та ефективно й швидко просувати їх далі у процесі інтеграції. Різноманітність підтримуваних платформ дозволяє декільком людям централізуватися на одному рішенні, щоб уникнути поширення на інші продукти.

Велика свобода конфігурації для користувача робить його придатним для будь-яких видів програмних продуктів. Покроковий перегляд виконання задач є дуже корисним інтерфейсом для відстеження загального стану процесу. Має потужні функції, такі як паралельне виконання кроків.

Недоліки CircleCI:

- незручна конфігурація пайплайнів за допомогою YAML синтаксису;
- швидкість роботи;
- ціна.

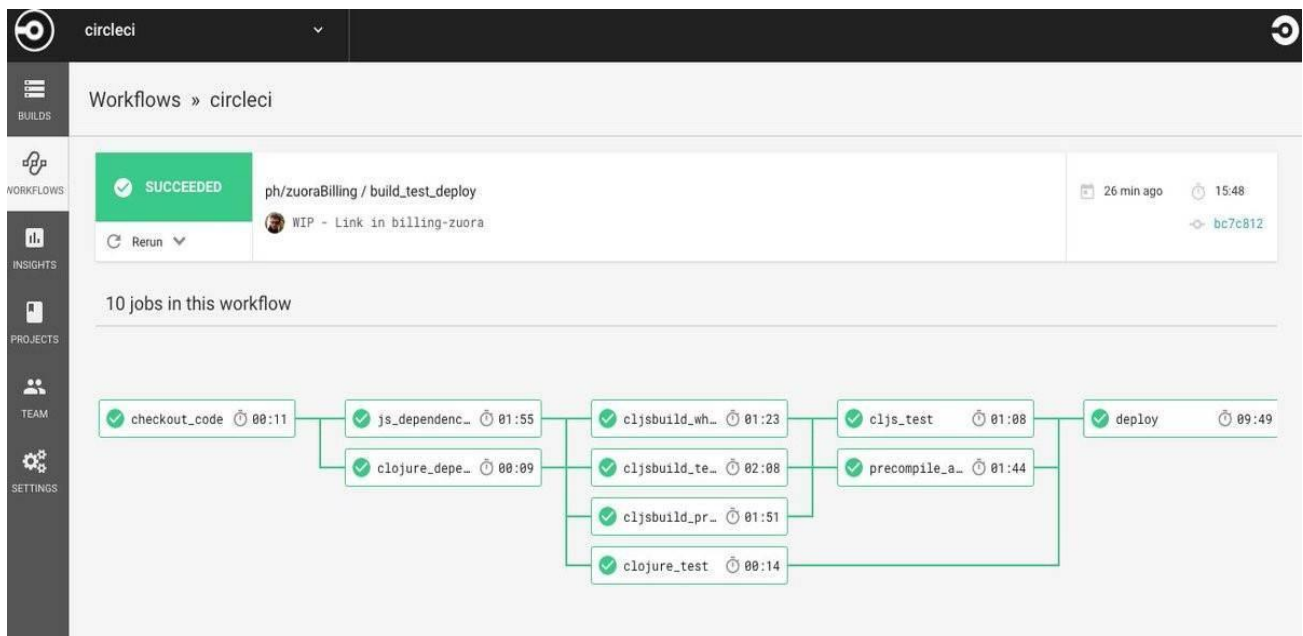


Рисунок 1.4 – Приклад використання CircleCI

Проаналізувавши усі аналоги, визначено їхні можливості та недоліки, які враховувались при створенні власного прототипа програмної системи інтеграції програмного продукту в режимі реального часу (табл. 1.1).

Таблиця 1.1 – Порівняльні характеристики доступних рішень

Критерій	Скрипти	Hudson	Jenkins	CircleCI	Власний прототип
Швидкість виконання процесу інтеграції	+	-	+	+	+
Екосистема плагінів	-	+	+	+	+
Можливість виконання в межах власної інфраструктури	+	-	+	-	+
Вибір місця збереження зчитаних вихідних даних контейнерів під час виконання процесу інтеграції	+	+	-	+	+
Можливість роботи з Docker контейнерами	+	-	+	+	+
Зрозумілий формат YAML файлу	-	-	-	-	+

Проаналізувавши аналоги доступних рішень було вирішено зробити акцент на швидкість виконання пайплайнів, зручність їх опису в YAML форматі та можливість подальшої кастомізації та розширення.

Таблиця порівняльних характеристик показала, що розробка програмної системи інтеграції програмного продукту в режимі реального часу є доцільною.

В результаті отримаємо систему, що покриває недоліки існуючих рішень й може слугувати заміною більшості з них.

### 1.3 Аналіз методів виконання кроків у середовищі Kubernetes

У якості середовища виконання кроків було обрано Kubernetes так як воно ідеально підходить для вирішення наступних проблем:

- жорстке управління доступом;
- стійкість до відмов;
- гнучкість у виборі додаткових інструментів (наприклад сховище даних);
- горизонтальна та вертикальна масштабованість;
- багато-кластерна конфігурація;
- управління конфігурація та секретами.

Кожен крок буде представлений конфігурацією, обов'язковим полем якої є `image`. На основі цієї інформації потрібно буде створити контейнер, в якому будуть виконуватися команди користувача. В Kubernetes це робиться за допомогою `Pod`.

`Pods` — це найменші, найпростіші об'єкти, які можна розгортати в Kubernetes. `Pod` являє собою один екземпляр запущеного процесу у вашому кластері [7].

`Pod` — це група з одного або кількох контейнерів із спільним сховищем та мережевими ресурсами, а також специфікацією для запуску контейнерів. Вміст `Pod` завжди розміщено разом і планується спільно, і працює в спільному контексті. `Pod` моделює специфічний для програми «логічний хост»: він містить один або кілька контейнерів програми, які відносно тісно пов'язані. У нехмарному контексті програми, що виконуються на одній фізичній або віртуальній машині, аналогічні хмарним додаткам, що виконуються на тому ж логічному хості.

`Pod` в кластері Kubernetes використовуються 2 способами:

- Pod, які запускають один контейнер. Модель «один контейнер на Pod» є найпоширенішим випадком використання Kubernetes; у цьому випадку ви можете уявити Pod як обгортку навколо окремого контейнера; Kubernetes керує Pods, а не безпосередньо керує контейнерами.
- Поди, які запускають кілька контейнерів, які повинні працювати разом. Pod може інкапсулювати програму, що складається з кількох спільно розташованих контейнерів, які тісно пов'язані між собою та потребують спільного використання ресурсів. Ці спільно розташовані контейнери утворюють єдину цілісну одиницю сервісу — наприклад, один контейнер обслуговує дані, що зберігаються в спільному томі, для загального доступу, тоді як окремий контейнер оновлює або оновлює ці файли. Pod об'єднує ці контейнери, ресурси сховища та недовговічний мережевий ідентифікатор разом як єдиний блок.

Pods призначені для підтримки кількох взаємодіючих процесів (як контейнери), які утворюють єдину одиницю обслуговування. Контейнери в Pod автоматично розташовуються і плануються спільно на одній фізичній або віртуальній машині в кластері. Контейнери можуть ділитися ресурсами та залежностями, спілкуватися один з одним і координувати, коли і як вони припиняються.

У одному поді може бути запущено декілька контейнерів, що виконуються на одному Docker демоні. Це дозволяє контейнерам взаємодіяти з Docker API. На основі цього було прийнято рішення реалізувати прослуховування вихідних даних контейнерів шляхом створення Pod таким чином, що поруч з основним контейнером (кроком) буде виконуватися контейнер слухач, задачою якого буде отримання логів контейнера за допомогою Docker API та надсилання їх до відповідного сховища.

Прикладом такого Pod'а може бути наступна конфігурація (рис. 1.5):

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: step-name
5  spec:
6    containers:
7      - name: step-name
8        image: user-provided-image
9    containers:
10     - name: listener
11       image: logs-listener
12       volumeMounts:
13         - name: dockersock
14           mountPath: "/var/run/docker.sock"
15
16   volumes:
17     - name: dockersock
18       hostPath:
19         path: /var/run/docker.sock
20

```

Рисунок 1.5 – Мінімальна конфігурація поду

#### 1.4 Аналіз методів зчитування вихідних даних кроків

Методів розв'язання зчитування вихідних даних не так багато, це через те що ми обмежені програмним інтерфейсом, який надає Docker. Виходячи з цього потрібно вивчити документацію й зрозуміти, які інструменти варто використати для вирішення задачі зчитування з контейнера.

В основі будь-якої реалізації буде зчитування з стандартних потоків даних контейнера: stdout та stdin. Читання з цих двох потоків надасть повну картину того що відбувається в середині контейнера.

Стандартні потоки введення/виведення в системах типу UNIX (і багато інших) [8] — потоки процесу, що мають номер (дескриптор), зарезервованій для виконання деяких «стандартних» функцій. Як правило (хоча і не обов'язково), ці дескриптори вже відкриті в момент запуску завдання.

В системі за замовчуванням завжди відкриті три "файли" - stdin (клавіатура), stdout (екран) і stderr (повідомлення про помилки на екран) [9]. Ці,

та будь-які інші відкриті файли, можуть бути перенаправлені. В реалізації програмного модуля Loggio буде використано `stdout` та `stderr`.

`stdout` – націлений на запис на пристрій виведення (монітор). Командна оболонка UNIX (і оболонки інших систем) дозволяють скерувати цей потік за допомогою символу «>». Для виконання програм у фоновому режимі цей потік зазвичай переводять у файл.

`stderr` – зарезервовано для виведення діагностики та повідомлень налаштування в текстовому вигляді. Частіше за все вивід цього потоку збігається з `stdout`, однак, на відміну від нього, місце призначення потоку `stderr` не змінюється при перепризначенні «>» і створенні конвеєрів ("|").

Часто ці потоки перенаправляєть в різні місця, для того щоб розділити помилки від інформації про виконання програми, але у нашому випадку це не доцільно, так як метою є створення єдиного потоку вихідної інформації контейнера. Вище приведені потоки будуть зчитуватися й зливатися в єдиний. Таким чином матимемо єдиний потік із всією вихідною інформацією контейнера. Це надає змогу без проблем реалізувати запис в вибране користувачем сховище, що є одною з ключових вимог до розроблюваного програмного модуля.

Docker API надає такі методи для роботи з вихідними даними контейнерів:

`logs` – отримує вихідні дані, присутні на час виконання `stdout` та `stderr`;

`attach` – прикріплюється до контейнера й відкриває потік в який потрапляє все що записується в `stdout` та `stderr` потоки.

Оскільки планується отримання всіх вихідних даних, а не тільки тих які присутні під час виклику метода, буде використано `attach` метод, що дозволяє прослуховувати стандартні потоки виводу контейнера, а також отримувати попередньо записані логи використовуючи опцію `logs`.



### 1.5 Постановка задачі

Після аналізу питання створення програмної системи інтеграції в режимі реального часу та модуля для зчитування вихідних даних контейнерів, було визначено наступні завдання, які необхідно виконати для розробки програмного продукту:

- провести аналіз існуючих рішень і засобів з автоматизації інтеграції та розгортання програмного коду;
- розробити розгалужену архітектуру програмної системи;
- розробити алгоритм взаємодії сервера й Kubernetes кластера;
- розробити алгоритм виконання пайплайну в середовищі Kubernetes;
- розробити алгоритм збереження вихідних даних кроків;
- розробити інтерфейс програмного продукту;
- розробити систему інтеграції програмного продукту в режимі реально часу.

### 1.6 Висновки

Під час аналізу стану питання і постановки задачі було розглянуто стан рішень для реалізації інтеграції в режимі реального часу на сьогоднішній день. Також було проаналізовано стан даного питання шляхом розгляду аналогів (Скрипти автоматизації, Hudson, Jenkins, CircleCI), та їх порівняння між собою, та розроблюваною програмною системою інтеграції програмного продукту в режимі реального часу. В результаті порівняння було відображено доцільність розробки програмної системи та проаналізовано можливі методи по вирішенню питання. Було обґрунтовано вибір мов програмування, які будуть використовуватися при розробці програмного модуля та наведено основні їх переваги. Також були

## 2 РОЗРОБКА МЕТОДУ ІНТЕГРУВАННЯ ПРОГРАМНИХ ПРОДУКТІВ У СЕРЕДОВИЩІ KUBERNETES

### 2.1 Аналіз інформаційного забезпечення

Структура інформаційної системи досить складна і включає різні комбінації інформаційних структур, що мають ієрархічну структуру побудови.

Для реалізації системи необхідно обрати технології, які в повній мірі зможуть задовольнити технічні та функціональні вимоги модулів.

Інформаційне забезпечення - це сукупність видів документів, нормативної бази та втілених рішень щодо обсягу, місця розташування та форм організації інформації, що циркулює в системі автоматизованої обробки інформації або в інформаційній системі [10].

Основними вимогами до інформаційного забезпечення є: цілісність, надійність, контроль, захист від несанкціонованого доступу, єдність та гнучкість, стандартизація та уніфікація, адаптованість, мінімізація помилок введення та виведення інформації.

Інформаційне забезпечення складається з таких частин: методичні та шаблонні матеріали, система розподілу та кодування, інформаційна база.

Основою інформаційного забезпечення є інформаційна база (ІБ). За складом та змістом вона повинна відповідати вимогам тих завдань, проектувати ті системи, які вирішуються на її основі. Відповідно до сфери експлуатації існують позамашинні та внутрішньомашинні ІБ.

Внутрішня інформаційна база - це частина інформаційної бази, яка представляє собою сукупність даних про машинні носії, що використовуються в інформаційній системі.

Інформаційна база даних зовнішніх машин - це частина, яка представляє собою набір повідомлень, сигналів та документів, призначених для безпосереднього сприйняття людиною без використання комп'ютерних технологій. Він складається з вхідних, вихідних та регуляторних повідомлень.

Програми пишуться для вирішення проблем. Щоб вирішити проблему, програмі потрібні вхідні дані.

Дані можна вводити різними способами:

- написати у коді програми. Це називається жорстким кодуванням;
- користувачем, під час роботи програми;
- з файлу чи іншого джерела, під час роботи програми.

Програмна система для інтеграції програмного продукту в режимі реального часу отримуватиме такі вхідні дані:

- адресу Kubernetes кластера;
- опис пайплайну у форматі yaml;
- тип сховища логів контейнерів.

Відповідно до переданого тексту у форматі yaml почнеться виконання пайплайну. Вихідними даними є результати виконання пайплайну.

## 2.2 Розробка розподіленої архітектури програмної системи

Архітектура програмного забезпечення є основною структурою програмної системи та критерієм створення таких структур та систем. Кожна структура містить програмні елементи, зв'язки між ними та атрибути елементів та відносин.

Основна мета створення якісної архітектури – виявити вимоги, які впливають на структуру програми. Добре закладена архітектура зменшує бізнес-ризик, пов'язані зі створенням технічного рішення та створює міст між бізнесом та технічними вимогами[11].

Узагальнена модель роботи системи представлена на рисунку 2.1.

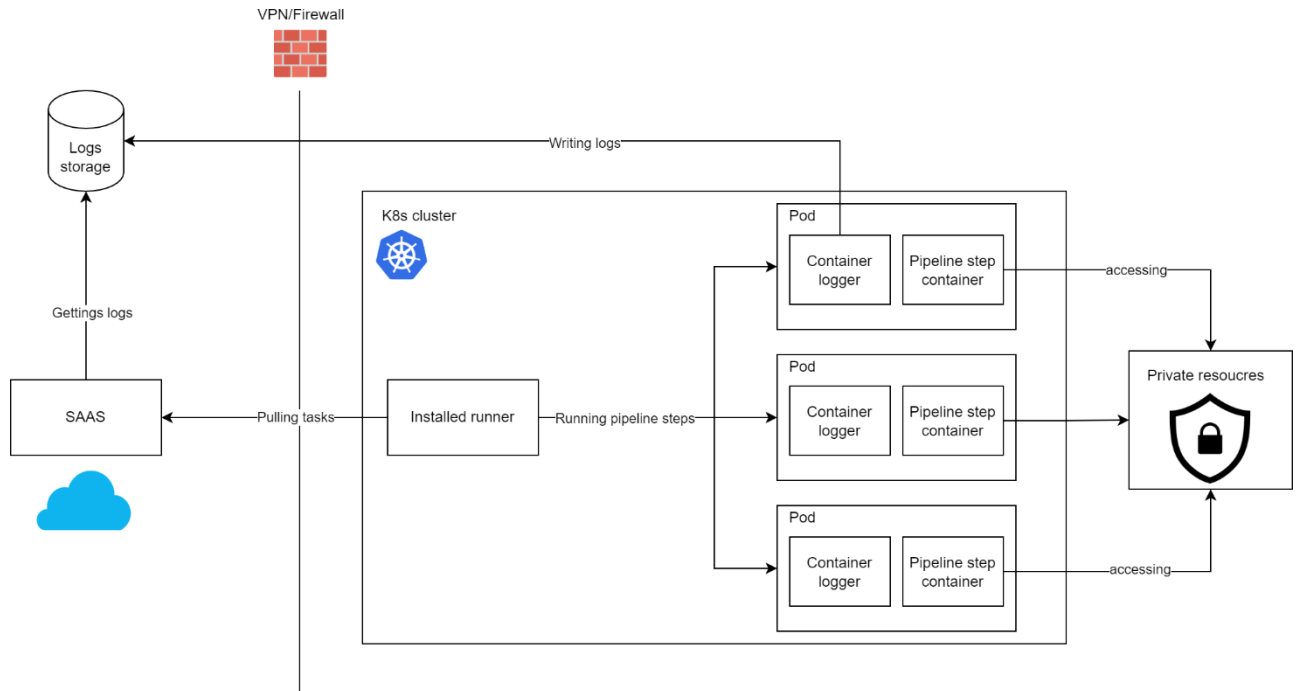


Рисунок 2.1 – Узагальнена модель роботи системи

Для коректної роботи програмної системи необхідно розгорнути SAAS частину у хмарі, базу даних для збереження вихідних даних контейнерів, та встановити компоненту “Runner” у кластері користувача.

Система складається з 3 основних компонент:

- SAAS;
- Runner;
- Container logger (модуль для логування вихідних даних контейнерів).

Під час запуску системи користувач налаштовує пайплайн за допомогою yaml синтаксису. Під час запуску пайплайну в базі даних буде додано запис про задачу, яку необхідно виконати. Runner в свою чергу постійно займається опитуванням серверу на предмет нових задач. Як тільки такі з’являються, він починає їх виконання в кластері користувача. Для кожного кроку пайплайну Runner створює Kubernetes Pod з двома контейнерами:

- крок пайплайну створений на основі іменю вказаного у конфігурації кроку;

- `container logger` – компонента, яка прослуховує вихідні дані сусіднього контейнера та записує їх до БД, що знаходиться у хмарі. Варто вказати що також можливе логування до БД що знаходиться у приватній мережі, але в такому випадку потрібно виконати додаткові налаштування для роботи в такий спосіб.

Роль даної компоненти в прослуховуванні `Docker` демона на вказаній користувачем адресі, фільтрування подій, які з нього надходять на предмет старту контейнера, та додаткове фільтрування відповідно до фільтрів які надав користувач. Якщо стартуючий контейнер успішно проходить фільтрацію, його ідентифікатор передається в `Container Logger` – компоненту відповідальну за зчитування вихідних даних контейнера. Як тільки зчитування відбулося, вихідні дані передаються в `StorageManager` – компоненту яка займається записом логів в вибране користувачем сховище.

Потрібно зазначити що зчитування вихідних даних це не одноразовий процес, це буде відбуватися за допомогою потоків даних, тобто відбуватися поступово. Чекаємо надходження порції даних – записуємо.

Через те що пайплайни виконуються в мережі користувача, у них є повний доступ до ресурсів, які знаходяться у цій мережі. Цими ресурсами можуть бути `Kubernetes` маніфести такі як `ConfigMap` та `Secret`, бази даних, захищені `API` сервіси та все інше, що може бути використано як вхідні дані для кроків.

### 2.3 Розробка алгоритму взаємодії сервера й `Kubernetes` кластера

Так як виконання кроків буде здійснено в `Kubernetes` кластері, буде доречною встановити агент, як частину кластеру. Він буде розгорнутий у вигляді `Pod`. Поруч з ним буде створено `ConfigMap`, з налаштуваннями агенту. Такими як інтервал опитування серверу, адреса серверу, максимальна кількість паралельно виконуваних задач.

Оскільки відповідно до розробленої архітектури агент, який відповідає за виконання пайплайнів, знаходиться у приватній мережі користувача, сервер не має доступу до нього. Таким чином потрібно відштовхуватися від того що зв'язок повинен бути одностороннім. Лише агент має доступ до серверу, але не навпаки. Отже задачею агенту буде постійне опитування сервера на предмет нових задач, які повинні бути виконані у кластері користувача.

Окрім цього агент повинен мати API токен, для виконання авторизованих запитів до серверу. Токен надається від час встановлення агенту у вигляді ConfigMap маніфесту, що знаходиться у кластері клієнта. У випадку коли токен не є валідним, Pod агенту не зможе виконувати опитування серверу. В такому випадку агент не зможе розпочати свою роботу й у його логах буде повідомлення про помилку.

Алгоритм взаємодії агента з сервером (рис. 2.2):

1. Ініціалізація агенту.
2. Отримання конфігурації з ConfigMap, що місти у собі API токен, інтервал опитування серверу, налаштування можливої кількості одночасно виконуваних кроків та пайплайнів.
3. Виконання аутентифікації за допомогою API токєну.
4. Якщо аутентифікація завершилася помилкою, закінчуємо роботу агенту та повідомляємо про необхідність оновлення токєну.
5. У випадку успіху починаємо нескінченний цикл опитування сервера з інтервалом отриманим під час ініціалізації.
6. Виконується запиті на отримання нових задач. Агент вказує їх бажану кількість на основі кількості активних задач та налаштувань одночасного виконання кроків/пайплайнів.
7. Виконання отриманих задач..
8. Витримка інтервалу.
9. Повторення циклу.



Рисунок 2.2 – Алгоритм взаємодії агента з сервером

## 2.4 Розробка алгоритму виконання пайплайну в середовищі Kubernetes

Пайплайн – це послідовність кроків, які необхідно виконати. CI/CD пайплайни – це практика, спрямована на покращення розгортання програмного забезпечення з використанням підходу DevOps або інженерної надійності сайту (SRE).

Пайплайн CI/CD впроваджує моніторинг та автоматизацію для покращення процесу розробки додатків, особливо на етапах інтеграції та тестування, а також під час доставки та розгортання. Хоча кожен із кроків пайплайну можна виконати вручну, справжня цінність CI/CD пайплайнів реалізується за допомогою автоматизації.

Наступний алгоритм буде використовуватися агентом для виконання задач, отриманих з серверу. Його принцип роботи досить лінійний і оснований на послідовному виконанні кроків пайплайну (рис. 2.3).



Рисунок 2.3 – Алгоритм виконання задач



## 2.5 Розробка методу зчитування вихідних даних контейнерів

Алгоритм — набір інструкцій, які описують порядок дій виконавця, щоб досягти результату розв'язання задачі за скінченну кількість дій; система правил виконання дискретного процесу, яка досягає поставленої мети за скінченний час. Для візуалізації алгоритмів часто використовують блок-схеми [12].

Ключовим модулем в розробці системи безперервної інтеграції програмного продукту є модуль для логування вихідних даних контейнерів. Розробка полягає в тому що необхідно розробити алгоритми прослуховування (рисунок 2.4), логування (рисунок 2.5), та загальний алгоритм його роботи (рисунок 2.6).

Алгоритм прослуховування демона:

1. Отримання фільтрів.
2. Ітерування масиву фільтрів і їх обробка відповідно до формату даних, який необхідний для правильного функціонування подальшого коду. У випадку виконання пайплайну, фільтром буде унікальний ідентифікатор пайплайну, який також буде прив'язаний до кожного запущеного контейнера у вигляді LABEL.
3. Використовуючи Docker API почати прослуховування подій на демоні. На цьому етапі можливе налаштування фільтрів які підтримуються Docker API. До таких належить тип події, назва контейнера, image який він використовує.
4. Якщо подія є стартом контейнера, продовжити виконання. Інакше закінчити обробку події.
5. Якщо позначки та ім'я контейнера який спричинив подію співпадають з тими, що ввів користувач, почати зчитування логів контейнеру. Інакше закінчити обробку події.

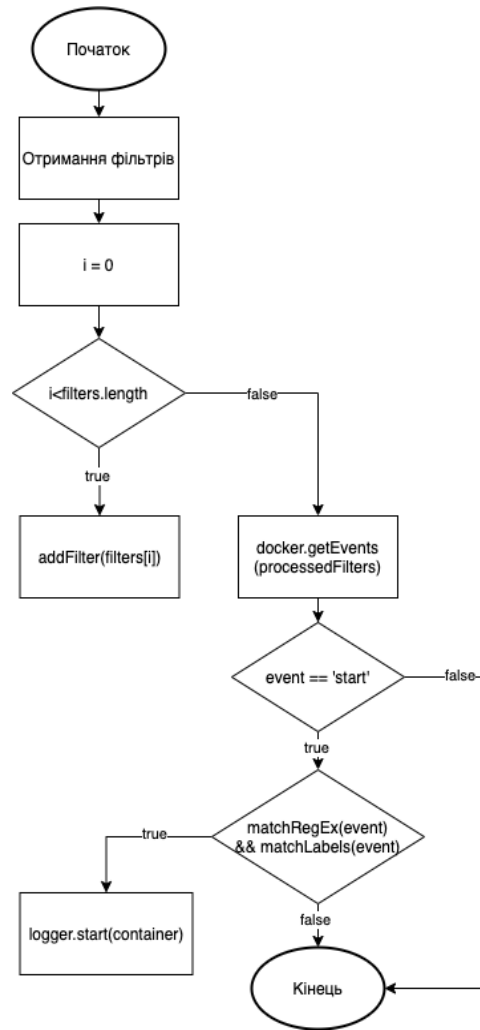


Рисунок 2.4 Блок-схема алгоритму прослуховування Docker демона

Алгоритм логування контейнера:

1. Отримання вибраного сховища та іd контейнера.
2. Ініціалізація сховища(наприклад створення з'єднання з базою даних).
3. Створення потоку вихідних даних шляхом під'єднання до контейнера (docker attach).
4. Поєднання стандартних потоків stdout та stderr в один.
5. Передача результуючого потоку в сховище.



Рисунок 2.5 – Блок-схема алгоритму логування вихідних даних контейнера

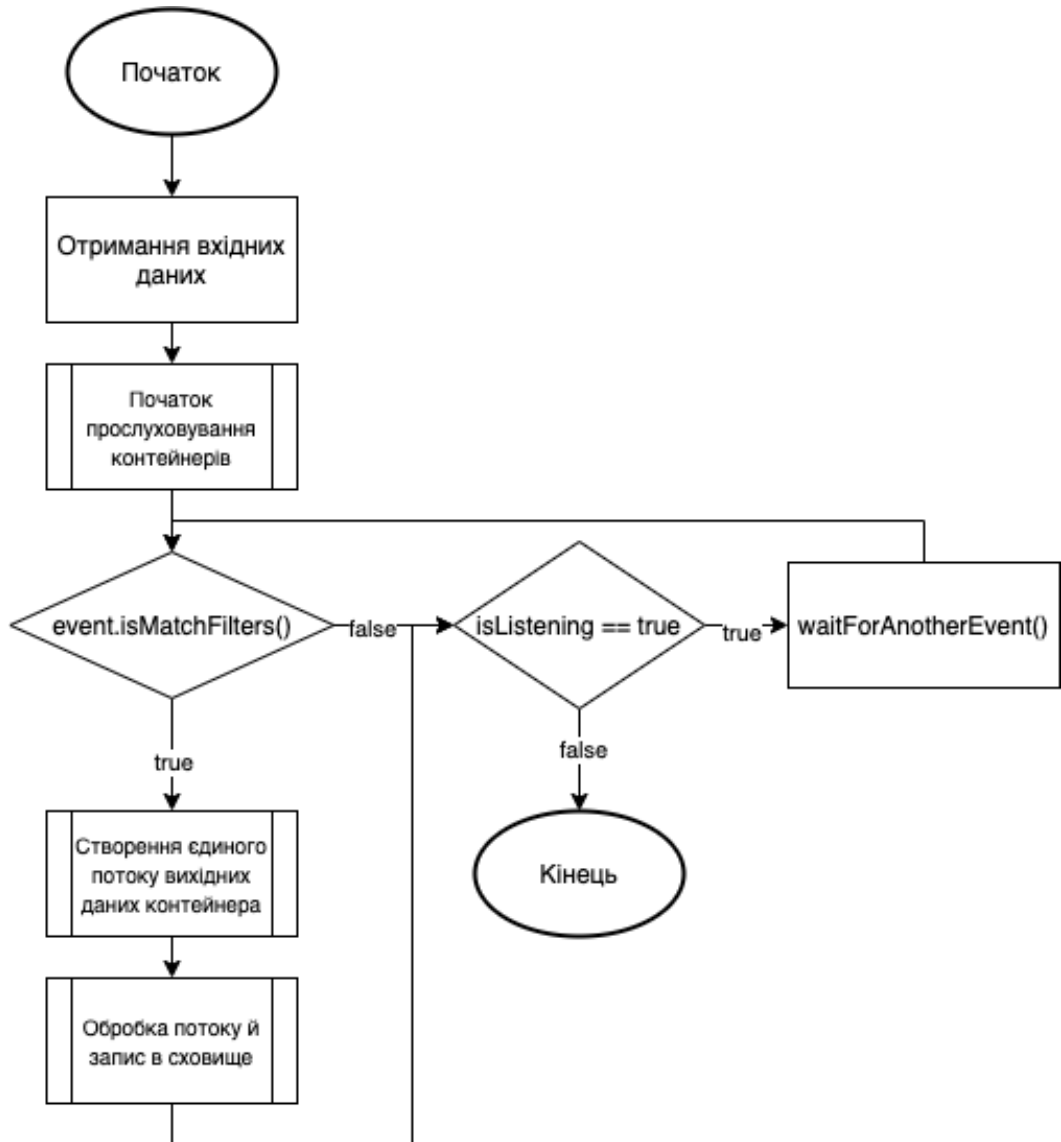


Рисунок 2.6 – Блок-схема загального алгоритму роботи модуля

Для відображення взаємозв'язку між всіма підпроцесами побудовано діаграму компонентів програмного модулю для прослуховування вихідних даних контейнерів (рисунок 2.7).

Діаграма компонент відображає залежності між компонентами програмного забезпечення, включаючи компоненти вихідних кодів, бінарні компоненти, та компоненти, що можуть виконуватись.

Вона дозволяє визначити архітектуру розроблюваної системи, встановив залежності між програмними модулями, компонентами, файлами. Пунктирні лінії, зображені на діаграмі, представляють відношення взаємозалежності.

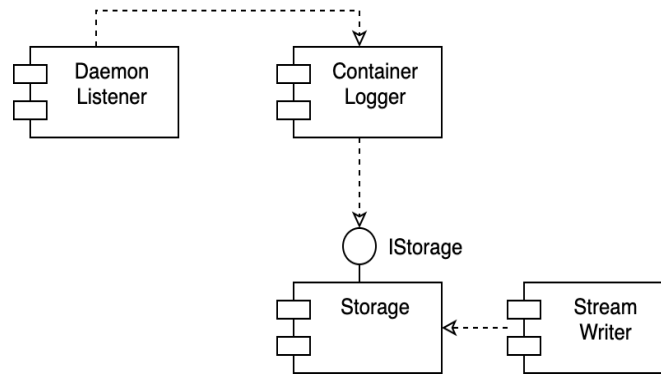


Рисунок 2.7 – Діаграма компонентів програмного модулю

## 2.6 Розробка структури графічного інтерфейсу

Інтерфейс – це сукупність засобів, методів і правил взаємодії між елементами системи [13].

Розробка візуальної композиції та тимчасової поведінки графічного інтерфейсу є важливою частиною програмування прикладних програм у сфері взаємодії людини та комп'ютера. Його мета — підвищити ефективність і простоту використання для базового логічного дизайну збереженої програми, дисципліни проектування під назвою юзабіліті. Методи дизайну, орієнтованого на користувача, використовуються для того, щоб візуальна мова, введена в дизайн, була добре пристосована до завдань.

Так як графічний інтрефейс є невід'ємною складовою додатків у сучасному світі, його створенню було прикладено багато зусиль. Було вирішено розробити графічний інтерфейс у вигляді веб додатку. Це не змушує користувача виконувати додаткові встановлення на його комп'ютер. Графічний інтерфейс виконаний в мінімалістичному стилі без використання різноманітних

бібліотек (рисунок. 2.8). Його задача – показати ефективність програмного модуля логування вихідних даних контейнерів.

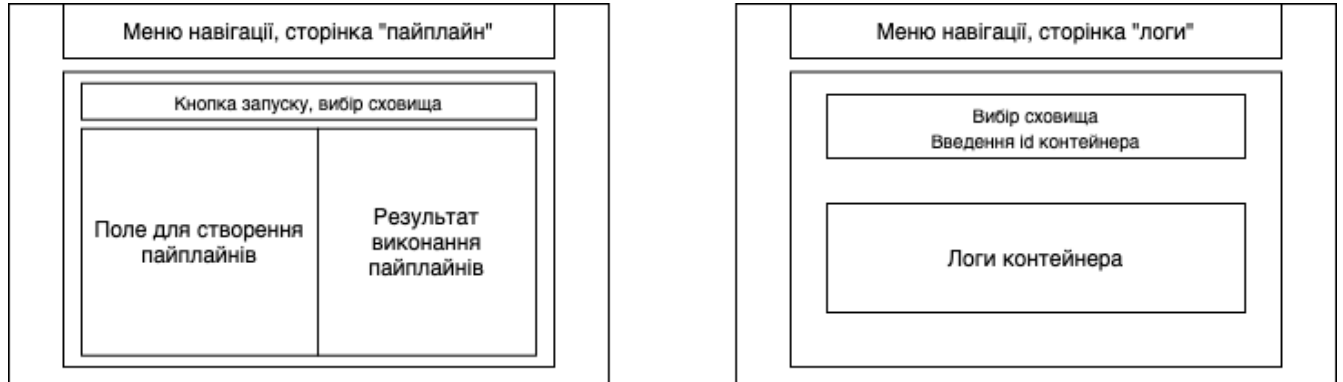


Рисунок 2.8 – Схема графічного інтерфейсу

## 2.7 Висновки

Було проаналізовано інформаційне забезпечення необхідне для розробки методу та програмного засобу. Також було розроблено архітектуру програмного додатку. Було розроблено алгоритми взаємодії сервера та Kubernetes кластера, виконання пайплайнів в середовищі Kubernetes, збереження вихідних даних контейнерів. Для останнього з них було розроблено діаграму компонентів для ілюстрації всіх залежностей між основними компонентами програмного модуля.

Було прийнято рішення про інтерфейс, який буде використовуватися при створенні програмної системи, та розроблено його прототип. Цей інтерфейс буде зручним для користувачів, тому він буде використовуватися в остаточній версії продукту.

### 3 РОЗРОБКА ПРОГРАМНОГО ДОДАТКУ

#### 3.1 Варіантний аналіз і обґрунтування вибору засобів реалізації програмного продукту.

Програмний продукт, який розробляється є не звичайним монолітним додатком, а системою з багатьма компонентами. Тільки працюючи разом вони створюють враження повноцінного продукту. Серед них є клієнтська частина, серверна частина, агент, який повинен бути встановленим у кластер клієнта та логер, який буде використовуватися для зчитування вихідних даних контейнерів. Отже, потрібно обрати мови програмування, яку будуть застосовані до розробки компонентів. Різноманітність задач дає можливість вибору різних мов програмування. Розгортання програмних компонентів теж вимагає дослідження.

Розглянемо технології, за допомогою яких це все було реалізовано.

##### 3.1.1 Робоче оточення, сервер та контейнеризація

Найперше що потрібно для функціонування програмного забезпечення – сервер на якому буде виконуватися більша частина бізнес логіки та буде розміщено статичні файли клієнтського застосунку. Одною з основних критерій є підтримка розгортання Docker контейнерів, так як це зробить процес розробки та розгортання значно простішим та гнучкішим, так як майбутня зміна хостингу буде безболісною. В якості такого серверу було обрано сервер, що працює на операційній системі Linux Alpine.

Alpine Linux побудований на базі musl libc і busybox. Це робить його меншим і більш ресурсоефективним, ніж традиційні дистрибутиви GNU/Linux. Контейнер вимагає не більше 8 МБ, а мінімальна установка на диск вимагає близько 130 МБ пам'яті. Ви отримуєте не тільки повноцінне середовище Linux, але й великий вибір додаткових утиліт із репозиторію.

Бінарні пакети зменшуються та розбиваються, що дає вам ще більше контролю над тим, що ви встановлюєте, що, у свою чергу, робить ваше середовище якомога меншим та ефективним.

Для того, щоб розгортання Docker контейнерів стало можливим, потрібно також встановити виконавчий файл, за допомогою якого буде запущено Docker daemon та інтерфейс командного рядка, що надасть змогу взаємодіяти з демоном. Це можна зробити використовуючи утиліту apk.

apk - менеджер пакунків Alpine Linux. Ви використовуєте команду apk для видалення, встановлення, оновлення або списку програмного забезпечення на запущеній системі на базі Alpine Linux. Як і більшість сучасних дистрибутивів Linux, усі програмні пакети для Alpine Linux мають цифровий підпис, щоб уникнути проблем із безпекою. Ви можете інсталювати пакунки з локального диска (наприклад, CDRом або USB-накопичувач) або з Інтернет-архіву (сховища), наприклад <http://dl-cdn.alpinelinux.org/alpine/v3.5/main>. Список сховищ зберігається у файлі конфігурації `/etc/apk/repositories`.

### 3.1.2 Вибір мови програмування для розробки серверної частини

Вибір мови програмування дуже важливий для розробки будь-якого програмного продукту, оскільки від неї залежить не тільки простота реалізації алгоритму, а також ефективності та швидкості роботи програми. Для реалізації цього програмного модуля потрібно обрати мову програмування, яка вміє чудово працювати з потоками даних. Це означає що вона повинна мати можливість підтримувати й обробляти велику кількість одночасних з'єднань, час життя яких значно більший ніж при звичайних HTTP запитах. На даний момент чудовим рішенням є використання подійно-орієнтованих мов програмування [14].

Ключовим моментом, чому ми обираємо подійно-орієнтовані мови програмування, є уникнення породження обслуговуючих процесів. Окрім того що їх створення є досить ресурсоємкою операцією, самі процеси будуть досить



довго використовуватися, так як будуть працювати з потоками. Це зробить програмний модуль повільним на системах з малою кількістю потоків процесора й навіть на більш потужних, якщо кількість прослуховуваних контейнерів буде великою [15].

Популярні подійно-орієнтовані рішення:

- JavaScript (Node.js);
- Python(Twisted);
- C++( libsigc++).

Node.js – платформа, заснована на движку V8 (здійснює трансляцію JavaScript в машинний код), що перетворює JavaScript з вузькоспеціалізованого мови в мову загального призначення. Node.js додає можливість JavaScript взаємодіяти з пристроями введення-виведення через свій API (написаний на C++), підключати інші зовнішні бібліотеки, написані на різних мовах, забезпечуючи виклики до них з JavaScript-коду. Node.js застосовується переважно на сервері, виконуючи роль веб-сервера, але є можливість розробляти на Node.js і десктопні віконні додатки (за допомогою NW.js, AppJS або Electron для Linux, Windows і macOS) і навіть програмувати мікроконтролери (наприклад, tessel і espruino). В основі Node.js лежить подійно-орієнтоване і асинхронне (або реактивне) програмування з неблокуючим вводом/виводом [16].

Node.js в основному використовується для створення мережеских програм, таких як веб-сервери. Найбільш істотна відмінність між Node.js і PHP полягає в тому, що більшість функцій у PHP блокують до завершення (команди виконуються лише після завершення попередніх команд), тоді як функції Node.js не блокують (команди виконуються одночасно або навіть паралельно, і використовувати зворотні виклики, щоб сигналізувати про завершення або невдачу).

libsigc++ - це бібліотека C ++ для зворотних викликів. Вона реалізує систему зворотного виклику для використання в абстрактних інтерфейсах та

загальному програмуванні. `libsigc++` - одна з найбільш ранніх реалізацій концепції сигналів і слотів, реалізованої за допомогою метапрограмування шаблонів C++. Він був створений як альтернатива використанню мета-компілятора, такого як в реалізації сигналів і слотів у Qt [17].

Таблиця 3.1 – Порівняння мов програмування

Мова програмування	JS(Node.js)	Python(Twisted)	C++(libsigc++)
Динамічна типізація	+	+	-
Event loop «з коробки»	+	-	-
Stream API «з коробки»	+	+	+
Швидкі операції з пам'яттю	+	-	+
Загалом	4	2	2

Згідно з результатами порівняння мова JavaScript має перевагу над іншими мовами, тому для реалізації серверної частини було обрано саме її.

### 3.1.3 Вибір мови програмування для розробки агенту

Для розробки агента встановленого в кластер користувача було обрано мову Golang.

Go (a.k.a. Golang) — мова програмування, вперше розроблена в Google [18]. Це статично типізована мова з синтаксисом, похідним від C, але з додатковими функціями, такими як збір сміття, безпека типів, деякі можливості

динамічного введення, додаткові вбудовані типи (наприклад, масиви змінної довжини та карти ключ-значення), і велика стандартна бібліотека.

Go виразний, лаконічний, чистий та ефективний. Його механізми паралельності дозволяють легко писати програми, які отримують максимальну віддачу від багатоядерних і мережевих машин, в той час як його нова система типів забезпечує гнучку і модульну конструювання програм. Go швидко компілюється в машинний код, але має зручність збирання сміття та можливість відображення під час виконання. Це швидка, статично типізована, скомпільована мова, яка схожа на динамічно типізовану, інтерпретовану мову.

Вибір впав саме на цю мову через її адаптованість під хмарну розробку. Оскільки Kubernetes було створено за допомогою цієї мови програмування, вона стала стандартом де-факто у світі хмарних технологій. Вона активно просувається проектом CNCF й є основною в більшості його проєктів.

The Cloud Native Computing Foundation (CNCF) — це проєкт Linux Foundation, який був заснований у 2015 році, щоб допомогти просунути технологію контейнерів та налаштувати технологічну галузь навколо її еволюції.

Golang має величезну екосистему бібліотек створену навколо Kubernetes. За допомогою цієї мови створюються розширення таких компонентів Kubernetes:

- controllers;
- scheduler;
- kubectl;
- network plugins;
- storage plugins.

### 3.1.4 Вибір середовища розробки

На сьогоднішній день найбільш зручними та сучасними середовищами для розробки веб додатків за допомогою мови програмування JavaScript є Visual Studio Code та WebStorm.

Visual Studio Code - це легкий, але потужний редактор вихідного коду, який доступний для Windows, MacOS та Linux. Він оснащений вбудованою підтримкою JavaScript, TypeScript та Node.js і має багату екосистему розширень для інших мов (таких як C ++, C #, Java, Python, PHP та Go) та режимів виконання (таких як .NET та Unity).

Основною його перевагою є можливість розширення функціоналу середовища розробки за допомогою розширень. У багатьох аналогів VSCode вони теж присутні, але саме у цьому редакторі їх буквально «безліч», що дозволяє його кастомізацію та використання для майже будь-яких цілей: програмування на будь-якій високорівневій мові, використання інструментів розробки.

VS Code постачається під стандартну ліцензію на продукт Microsoft, оскільки він має невеликий відсоток налаштувань, характерних для Microsoft. Це безкоштовно, незважаючи на комерційну ліцензію.

Інтерфейс редактору Visual Studio Code зображено на рисунку 3.1.

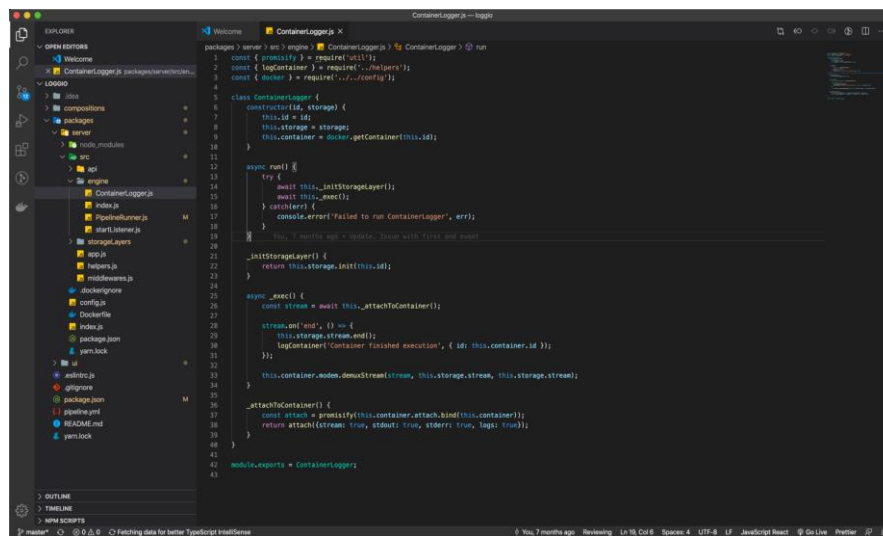


Рисунок 3.1 – Інтерфейс редактору Visual Studio Code

WebStorm - це інтегроване середовище розробки (IDE) від JetBrains для мови програмування JavaScript. Крім того, WebStorm підтримує HTML5, Node.js, Bootstrap, Angular / AngularJS, TypeScript, PhoneGap / Cordova, Dart та багато інших технологій, тому в основному використовується для розробки веб-мобільних додатків. WebStorm заснований на IntelliJ IDEA від JetBrains, але це версія, спеціалізована на JavaScript.

Діапазон функцій можна розширити за допомогою плагінів, які частково розроблені JetBrains, а частково громадою.

WebStorm містить інструменти для рефакторингу, контролю версій, виділення коду та синтаксису, тестування та для часткової генерації коду. Він пропонує вам інтелектуальну допомогу в програмуванні на JavaScript та мовах які компілюються в нього, Node.js, HTML та CSS. Надає потужні рішення для авто-завершенням коду, навігації між файлами та проектами, виявленням помилок на ходу та рефакторинга для всіх перелічених мов програмування. IDE аналізує проект, щоб забезпечити найкращі результати заповнення коду для всіх підтримуваних мов. Сотні вбудованих інспекцій повідомляють про всі можливі проблеми під час написання коду та пропонують їх рішення.

Окрім цього дана IDE має виключну підтримку тестування, оскільки WebStorm інтегрується з такими рушіями тестів як Karma, Mocha, Jest і Protractor. Це дозволяє запускати й відлагоджувати тести прямо в IDE, переглядати результати їх виконання у хорошому візуальному форматі з можливістю перейти до будь-якого виконаного тесту.

Інтерфейс інтегрованого середовища Webstorm зображено на рисунку 3.2.

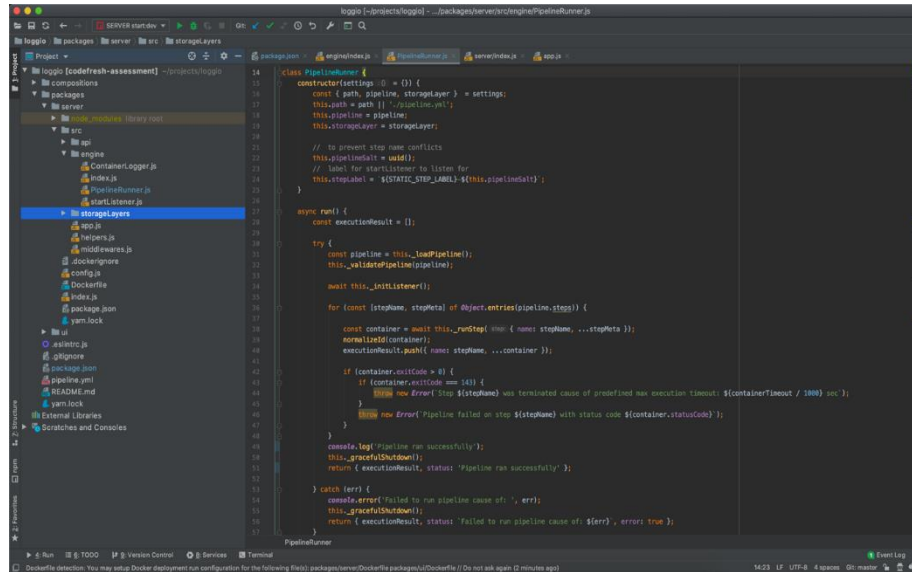


Рисунок 3.2 – Інтерфейс IDE Webstorm

### 3.1.5 Вибір бази даних

База даних — це організована сукупність даних, які зберігаються та доступні в електронному вигляді з комп'ютерної системи. Там, де бази даних є складнішими, вони часто розробляються з використанням формальних методів проектування та моделювання.

При виборі сучасної бази даних одним із найважливіших рішень є вибір реляційної (SQL) або нереляційної (NoSQL) структури даних. Хоча обидва варіанти є життєздатними, між ними є ключові відмінності, які користувачі повинні мати на увазі, приймаючи рішення.

На самому базовому рівні найбільша відмінність між цими двома технологіями полягає в тому, що бази даних SQL є реляційними, а бази даних NoSQL нереляційними [19].

Бази даних SQL використовують структуровану мову запитів і мають попередньо визначену схему для визначення та маніпулювання даними. SQL є однією з найбільш універсальних і широко використовуваних мов запитів, що робить його безпечним вибором для багатьох випадків використання. Він ідеально підходить для складних запитів. Однак SQL може бути занадто обмежувачим. Перш ніж працювати з нею, потрібно використовувати

попередньо визначені схеми, щоб визначити структуру даних. Усі ваші дані повинні мати однакову структуру. Цей процес вимагає серйозної попередньої підготовки. Якби ви коли-небудь захотіли змінити структуру даних, це було б складно і зашкодило б всій вашій системі.

Бази даних NoSQL мають динамічні схеми для неструктурованих даних, і дані зберігаються багатьма способами. Ви можете використовувати для своїх даних сховище, стовпчико-орієнтоване, документо-орієнтоване, графіко орієнтоване або Key Value. Ця гнучкість означає, що:

- ви можете створювати документи без попереднього визначення їх структури;
- кожен документ може мати свою унікальну структуру;
- синтаксис може відрізнитися від бази даних до бази даних;
- ви можете додавати поля по ходу.

Бази даних SQL у більшості ситуацій вертикально масштабовані. Ви можете збільшити навантаження на один сервер, додавши більше ємності CPU, RAM або SSD. Бази даних NoSQL мають горизонтальне масштабування. Ви можете обробляти більший трафік за допомогою сегментування, що додає більше серверів до вашої бази даних NoSQL. Горизонтальне масштабування має більшу загальну ємність, ніж вертикальне масштабування, що робить бази даних NoSQL кращим вибором для великих і часто змінних наборів даних.

Підсумовуючи, п'ять основних відмінностей між SQL і NoSQL:

- бази даних SQL є реляційними, бази даних NoSQL нереляційними.
- бази даних SQL використовують структуровану мову запитів і мають попередньо визначену схему.
- бази даних NoSQL мають динамічні схеми для неструктурованих даних.
- бази даних SQL мають вертикальне масштабування, а бази даних NoSQL горизонтально.

- бази даних SQL засновані на таблицях, тоді як бази даних NoSQL є сховищами документів, ключів і значень, графіків або широких стовпців.
- бази даних SQL краще для багаторядкових транзакцій, тоді як NoSQL краще для неструктурованих даних, таких як документи або JSON.

Оцінивши всі переваги та недоліки вказані вище було прийнято рішення використовувати NoSQL базу даних MongoDB. Це БД з відкритим вихідним кодом, що дає змогу безкоштовно її використовувати та розгорнути на власній інфраструктурі.

### 3.1.6 Вибір зворотнього-проксі сервера

Зворотний проксі-сервер — це тип проксі-сервера, який розгортається між клієнтами та внутрішніми/початковими серверами, наприклад, HTTP-сервером, таким як NGINX, Apache тощо, або серверами програм, написаними на Nodejs, Python, Java, Ruby, PHP та багато інших мов програмування.

Це шлюз або проміжний сервер, який приймає запит клієнта, передає його одному або кільком внутрішнім серверам і згодом отримує відповідь від сервера і доставляє її назад клієнту, таким чином створюючи враження, ніби вміст походить із самого зворотного проксі-сервера.

Як правило, зворотний проксі-сервер — це внутрішній проксі-сервер, який використовується як «фронт-енд» для контролю та захисту доступу до внутрішніх серверів у приватній мережі: зазвичай він розгортається за мережесим брандмауером.

Це допомагає внутрішнім серверам досягти анонімності для підвищення їх безпеки. В IT-інфраструктурі зворотний проксі-сервер також може функціонувати як брандмауер програми, балансувальник навантаження,



термінатор TLS, веб-прискорювач (за допомогою кешування статичного та динамічного вмісту) та багато іншого.

В якості зворотнього проксі серверу було обрано NGINX.

Це безкоштовний, високопродуктивний і дуже популярний HTTP-сервер і зворотний проксі-сервер з відкритим вихідним кодом. NGINX добре відомий своєю високою продуктивністю, стабільністю, багатим набором функцій, простою та гнучкою конфігурацією та низьким споживанням ресурсів (особливо малим обсягом пам'яті).

NGINX підтримує прискорене зворотне проксі-сервер із кешуванням за допомогою модуля `ngx_http_proxy_module`, який дозволяє передавати запити на інший сервер через протоколи, відмінні від HTTP, наприклад FastCGI, uwsgi, SCGI та memcached.

Важливо, що він підтримує балансування навантаження та відмовостійкість, які є життєво важливими аспектами великомасштабних розподілених обчислювальних систем. Модуль `ngx_http_upstream_module` дозволяє визначати групи серверних серверів для розподілу запитів, що надходять від клієнтів. Це робить ваші програми більш надійними, доступними та надійними, високомасштабованими, з часом відгуку та пропускнуою здатністю. Крім того, що стосується безпеки, він підтримує припинення SSL/TLS та багато інших функцій безпеки.

### 3.2 Розробка графічного інтерфейсу користувача

Для розробки клієнтського інтерфейсу було вирішено використовувати технологію Single Page Application.

Односторінкові сайти допомагають підтримувати користувача в єдиному, зручному веб-просторі, де вміст представляється користувачеві просто, легко та зручно.

Плюси використання SPA:

- Легке відлагодження в Chrome, оскільки є можливість контролювати мережеві операції, досліджувати елементи сторінки та пов'язані з нею дані.
- Швидкість. Оскільки більшість ресурсів (HTML + CSS + скрипти) завантажуються лише один раз протягом усього життя програми. Повторно можуть бути завантажені лише дані.
- SPA дозволяє ефективно кешувати дані. Додаток надсилає лише один запит й зберігає всі дані, тоді він може використовувати ці дані та працювати навіть офлайн.

Однією із найзручніших технологій для реалізації SPA є бібліотека для JavaScript – React.js.

React.js - це бібліотека JavaScript, створена для створення швидких та інтерактивних інтерфейсів користувачів для веб- та мобільних додатків. Це бібліотека, що працює на компонентах, з відкритим кодом, на основі компонентів, що відповідає лише за шар перегляду програми. У архітектурі Model View Controller (MVC) шар перегляду відповідає за те, як виглядає і почувається додаток:

- React використовує підхід, заснований на компонентах. У ньому кожен елемент DOM може бути компонентом. Це забезпечує найкращі умови створення ізольованих фрагментів коду та логіки, які простіше відлагоджувати, підтримувати та використовувати повторно.
- здатність боротися із поширеною помилкою пошукової системи для читання важких JavaScript додатків. Як рішення, React може працювати на сервері, надаючи та повертаючи віртуальний DOM у браузер як звичайну веб-сторінку.

UML-діаграми активності алгоритму запуску пайплайну та отримання результатів виконання відображено на рисунках 3.3 та 4.4 відповідно.

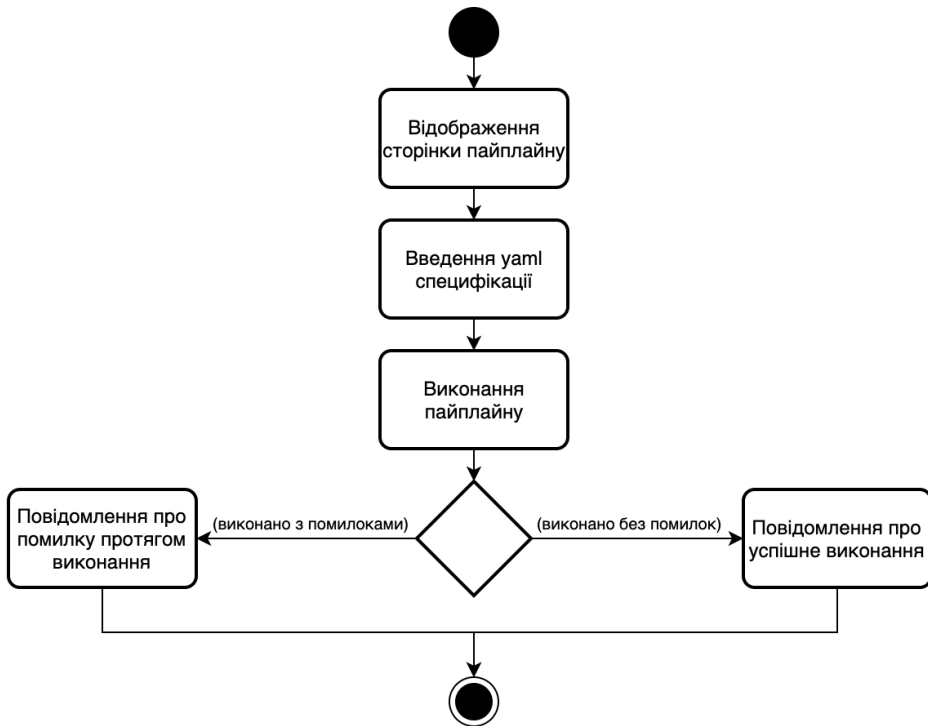


Рисунок 3.3 – Діаграма активності алгоритму запуску пайплайну



Рисунок 3.4 – Діаграма активності алгоритму отримання логів

### 3.3 Розробка модуля прослуховування та логування вихідних даних кроків

Ключовими програмними компонентами є модулі прослуховування Docker демона та логування вихідних даних контейнера.

Docker надає API для взаємодії з демоном Docker (так званий Docker Engine API), а також пакети SDK для Go і Python. Пакети SDK дозволяють створювати та масштабувати програми та рішення Docker швидко й легко. Якщо Go або Python не працюють для вас, ви можете використовувати Docker Engine API безпосередньо. API Docker Engine — це RESTful API, до якого звертаються HTTP клієнти, наприклад wget або curl, або бібліотека HTTP, яка є частиною більшості сучасних мов програмування.

Більшість CLI команд безпосередньо відповідають кінцевим точкам API (наприклад, `docker ps GET /containers/json`). Важливим винятком є запуск контейнерів, який складається з кількох викликів API.

Обидва модулі реалізовано за допомогою Docker API, а саме таких HTTP запитів:

- GET /events;
- POST /containers/(id or name)/attach.

GET /events – запит на отримання потоку подій які відбуваються на Docker демоні. Під час його надсилання є можливість вказати тип подій, які ми хочемо отримувати в потоку подій.

POST /containers/(id or name)/attach – запит на отримання потоку вихідних даних контейнера. Він надсилається після того як подія про старт контейнера, яка була отримана з потоку подій, пройшла валідацію відповідно до фільтрів та id контейнера було передано в ContainerLogger модуль.

Код алгоритму прослуховування наведено на рисунку 3.5:

```

1  const _ = require('lodash');
2  const { docker } = require('.././config');
3
4  class StartListener {
5      start(selector, containerHandler) {
6          return new Promise((resolve, reject) => {
7              const filters = this._getApiFilter(selector);
8              docker.getEvents({ filters }, (err, events) => {
9                  if (err) {
10                     reject(err);
11                 } else {
12                     console.log('Listening for starting containers...');
13                     this.events = events;
14                     events.on('data', (chunk) => {
15                         const containerData = JSON.parse(chunk.toString());
16                         if (this._matchLocalFilters(containerData, selector)) {
17                             containerHandler(containerData)
18                         }
19                     });
20                     resolve();
21                 }
22             });
23         });
24     }
25
26     stop() {
27         if (this.events) {
28             this.events.destroy();
29             this.events = null;
30             console.log('No longer listen to docker `start` events');
31         } else {
32             console.log('StartListener already stopped');
33         }
34     }
35
36
37
38     _matchLocalFilters(eventData, selector) {
39         const name = _.get(eventData, 'Actor.Attributes.name');
40         if (selector.nameRegex && name) {
41             return (new RegExp(selector.nameRegex)).test(name);
42         }
43         return true;
44     }
45
46     _getApiFilter(selector) {
47         const filter = {};
48         _.set(filter, 'event', ['start']);
49
50         if (selector.labels) {
51             _.set(filter, 'label', _.castArray(selector.labels));
52         }
53
54         return JSON.stringify(filter);
55     }
56 }
57
58 module.exports = new StartListener();

```

Рисунок 3.5 – Код алгоритму прослуховування

Код алгоритму логування наведено на рисунку 3.6:

```

1  const { promisify } = require('util');
2  const { logContainer } = require('../helpers');
3  const { docker } = require('../config');
4
5  class ContainerLogger {
6    constructor(id, storage) {
7      this.id = id;
8      this.storage = storage;
9      this.container = docker.getContainer(this.id);
10   }
11
12   async run() {
13     try {
14       await this._initStorageLayer();
15       await this._exec();
16     } catch(err) {
17       console.error('Failed to run ContainerLogger', err);
18     }
19   }
20
21   _initStorageLayer() {
22     return this.storage.init(this.id);
23   }
24
25   async _exec() {
26     const stream = await this._attachToContainer();
27
28     stream.on('end', () => {
29       this.storage.stream.end();
30       logContainer('Container finished execution', { id: this.container.id });
31     });
32
33     this.container.modem.demuxStream(stream, this.storage.stream, this.storage.stream);
34   }
35
36   _attachToContainer() {
37     const attach = promisify(this.container.attach.bind(this.container));
38     return attach({stream: true, stdout: true, stderr: true, logs: true});
39   }
40 }
41
42 module.exports = ContainerLogger;

```

Рисунок 3.6 – Код алгоритму логуювання

В початковій версії програмної системи передбачено 2 способи збереження вихідних даних контейнерів:

- Mongo DB;
- файлова система.

MongoDB — документо-орієнтована система керування базами даних (СКБД) з відкритим вихідним кодом, яка не потребує опису схеми таблиць. MongoDB займає нішу між швидкими і масштабованими системами, що оперують даними у форматі ключ/значення, і реляційними СКБД, функціональними і зручними у формуванні запитів. При запуску програмного модуля користувач може вказати адресу запущеного MongoDB демона через

змінну оточення MONGO\_URI. По замовчуванню вона дорінює «mongodb://mongo/docker-logger».

При збереженні в файлову систему для кожного контейнера будуть створені файли, де ім'я файла – id контейнера. По замовчуванню вони буду створені за шляхом «/<os\_home\_dir>/<current\_user>/fs-logs-storage».

На рисунку 3.7 наведений код базових абстрактних класів модуля збереження даних.

```
1  const WriterStream = require('./WriterStream');
2
3  class StorageLayer {
4    constructor(container) {
5      this.container = container
6    }
7
8    /**
9     * Will be called before writing to stream
10   */
11   async init() {}
12 }
13
14 class StorageLayerReader extends StorageLayer {
15   async read() {}
16 }
17
18 class StorageLayerWriter extends StorageLayer {
19   constructor(container) {
20     super(container);
21     this.stream = new WriterStream({
22       onData: this.write.bind(this),
23       onFinish: this.onFinish.bind(this),
24     });
25   }
26
27   /**
28    * Will be called every time stream receive data
29    */
30   write(log) {
31     console.log(log);
32   }
33
34   /**
35    * Will be called on stream 'finish' event
36    */
37   onFinish() {}
38 }
39
40 module.exports = {
41   Writer: StorageLayerWriter,
42   Reader: StorageLayerReader
43 };
```

Рисунок 3.7 – Код модуля збереження даних

На рисунку 3.8 та 3.9 наведений код реалізації класів модуля збереження даних для MongoDB.

```
1  const mongoose = require('mongoose');
2
3  const BaseStorageLayer = require('../Base');
4  const ContainerLog = require('../Log');
5  const { mongo } = require('.././../config');
6  const { notFoundMessage } = require('../constants');
7
8  class MongoStorageRead extends BaseStorageLayer.Reader {
9    async init() {
10     if (mongoose.connection.readyState === 0) {
11       return this._initMongo();
12     }
13   }
14
15   async read() {
16     const { id } = this.container;
17     const res = await ContainerLog.findById(id);
18     if (res) {
19       return res.data;
20     } else {
21       throw new Error(notFoundMessage);
22     }
23   }
24
25   _initMongo() {
26     return mongoose.connect(mongo.uri, { useNewUrlParser: true, useUnifiedTopology: true });
27   }
28 }
29
30 module.exports = MongoStorageRead;
```

Рисунок 3.8 – Код зчитування даних з MongoDB



```
1 const mongoose = require('mongoose');
2
3 const BaseStorageLayer = require('../Base');
4 const ContainerLog = require('../Log');
5 const { mongo } = require('.././../config');
6 const { getImageAndName } = require('.././../helpers');
7
8 class MongoStorageWrite extends BaseStorageLayer.Writer {
9   onFinish() {
10     this.containerLog.save().then(() => console.log('Saved successfully!')).catch(console.error);
11   }
12
13   async init() {
14     this.containerLog = new ContainerLog({
15       _id: this.container.id,
16       data: '',
17       ..getImageAndName(this.container)
18     });
19
20     if (mongoose.connection.readyState === 0) {
21       return this._initMongo();
22     }
23   }
24
25   write(log) {
26     this.containerLog.data += log;
27   }
28
29   _initMongo() {
30     return mongoose.connect(mongo.uri, { useNewUrlParser: true, useUnifiedTopology: true });
31   }
32 }
33
34 module.exports = MongoStorageWrite;
```

Рисунок 3.9 – Код збереження даних у MongoDB

### 3.4 Розробка модуля виконання пайплайну в середовищі Kubernetes

Так як у якості мови програмування для модуля виконання пайплайну було обрано Go, розпочнемо з ініціалізації проекту. Створимо main.go файл та імплементуємо функцію main (рис. 3.10).

```

1  package main
2
3  import (
4      "github.com/zubroide/go-api-boilerplate/command"
5      "github.com/zubroide/go-api-boilerplate/dic"
6      "fmt"
7      "github.com/getsentry/raven-go"
8      "github.com/spf13/viper"
9      "os"
10 )
11
12 func readConfig() {
13     var err error
14
15     viper.SetConfigFile("base.env")
16     viper.SetConfigType("props")
17     err = viper.ReadInConfig()
18     if err != nil {
19         fmt.Println(err)
20         return
21     }
22
23     if _, err := os.Stat(".env"); os.IsNotExist(err) {
24         fmt.Println("WARNING: file .env not found")
25     } else {
26         viper.SetConfigFile(".env")
27         viper.SetConfigType("props")
28         err = viper.MergeInConfig()
29         if err != nil {
30             fmt.Println(err)
31             return
32         }
33     }
34
35     // Override config parameters from environment variables if specified
36     for _, key := range viper.AllKeys() {
37         viper.BindEnv(key)
38     }
39 }
40
41 func main() {
42     readConfig()
43
44     dic.InitContainer()
45
46     client := dic.Container.Get(dic.RavenClient).(*raven.Client)
47     if client != nil {
48         client.CapturePanicAndWait(func() {
49             command.Execute()
50         }, nil)
51     } else {
52         command.Execute()
53     }
54 }

```

Рисунок 3.10 – main.go

Одною з основних задач агента є опитування сервера на предмет нових задач. Але це не єдина задача агенту, так що виконуватися опитувач повинен в окремому потоці. Для цього необхідно використати go рутину. Код ініціалізації агенту зображено на рисунку 3.11.

```

106 // New creates a new Agent instance
107 func New(opt *Options) (*Agent, error) {
108     if err := checkOptions(opt); err != nil {
109         return nil, err
110     }
111     id := opt.ID
112     cf := opt.Codefresh
113     runtimes := opt.Runtimes
114     log := opt.Logger
115     taskPullingInterval := defaultTaskPullingInterval
116     if opt.TaskPullingSecondsInterval != time.Duration(0) {
117         taskPullingInterval = opt.TaskPullingSecondsInterval
118     }
119     statusReportingInterval := defaultStatusReportingInterval
120     if opt.StatusReportingSecondsInterval != time.Duration(0) {
121         statusReportingInterval = opt.StatusReportingSecondsInterval
122     }
123     taskPullerTicker := time.NewTicker(taskPullingInterval)
124     reportStatusTicker := time.NewTicker(statusReportingInterval)
125     wg := &sync.WaitGroup{}
126
127     if opt.Monitor == nil {
128         opt.Monitor = monitoring.NewEmpty()
129     }
130     httpClient.HTTPClient.Transport = opt.Monitor.NewRoundTripper(httpClient.HTTPClient.Transport)
131
132     return &Agent{
133         id,
134         cf,
135         runtimes,
136         log,
137         taskPullerTicker,
138         reportStatusTicker,
139         false,
140         Status{},
141         wg,
142         opt.Monitor,
143     }, nil
144 }
145
146 // Start starting the agent process
147 func (a *Agent) Start(ctx context.Context) error {
148     if a.running {
149         return errAlreadyRunning
150     }
151     a.running = true
152     a.log.Info("Starting agent")
153
154     go a.startTaskPullerRoutine(ctx)
155     go a.startStatusReporterRoutine(ctx)
156
157     reportStatus(ctx, a.cf, codefresh.AgentStatus{
158         Message: "All good",
159     }, a.log)
160
161     return nil
162 }

```

Рисунок 3.11 –agent.go – ініціалізація агенту

Код циклу опитування зображено на рисунку 3.12.

```

182 func (a *Agent) startTaskPullerRoutine(ctx context.Context) {
183     for {
184         select {
185             case <-ctx.Done():
186                 return
187             case <-a.taskPullerTicker.C:
188                 a.wg.Add(1)
189                 go func(runtimes map[string]runtime.Runtime, wg *sync.WaitGroup, logger logger.Logger, monitor monitoring.Monitor) {
190                     tasks := pullTasks(ctx, client, logger)
191                     startTasks(ctx, tasks, runtimes, logger, monitor)
192                     time.Sleep(time.Second * 10)
193                     wg.Done()
194                 }(a.cf, a.runtimes, a.wg, a.log, a.monitor)
195         }
196     }
197 }

```

Рисунок 3.12 –agent.go – виконання опитування серверу

Результатом успішного опитування буде список задач, які необхідно виконати. Вони передаються до функції startTasks, імпліmentaція якої зображена на рисунку 3.13.

```

238 func startTasks(ctx context.Context, tasks []task.Task, runtimes map[string]runtime.Runtime, logger logger.Logger, monitor monitoring.Monitor) {
239     creationTasks := []task.Task{}
240     deletionTasks := []task.Task{}
241     agentTasks := []task.Task{}
242
243     // divide tasks by types
244     for _, t := range tasks {
245         logger.Debug("Received task", "type", t.Type, "tid", t.Metadata.Workflow, "runtime", t.Metadata.ReName)
246         switch t.Type {
247             case task.TypeCreatePod, task.TypeCreatePVC:
248                 creationTasks = append(creationTasks, t)
249             case task.TypeDeletePod, task.TypeDeletePVC:
250                 deletionTasks = append(deletionTasks, t)
251             case task.TypeAgentTask:
252                 agentTasks = append(agentTasks, t)
253             default:
254                 logger.Error("unrecognized task type", "type", t.Type, "tid", t.Metadata.Workflow, "runtime", t.Metadata.ReName)
255         }
256     }
257
258     // process agent tasks
259     for i := range agentTasks {
260         t := agentTasks[i]
261         logger.Info("executing agent task", "tid", t.Metadata.Workflow)
262         txn := newTransaction(monitor, t.Type, t.Metadata.Workflow, t.Metadata.ReName)
263         go func(tid string) {
264             if err := executeAgentTask(&t, logger); err != nil {
265                 logger.Error(err.Error())
266                 txn.NoticeError(err)
267             }
268             txn.End()
269             logger.Info("finished agent task", "tid", t.Metadata.Workflow)
270         }(t.Metadata.Workflow)
271     }
272
273     // process creation tasks
274     for _, tasks := range groupTasks(creationTasks) {
275         reName := tasks[0].Metadata.ReName
276         runtime, ok := runtimes[reName]
277         txn := newTransaction(monitor, "start-workflow", tasks[0].Metadata.Workflow, reName)
278
279         if !ok {
280             logger.Error("Runtime not found", "workflow", tasks[0].Metadata.Workflow, "runtime", reName)
281             txn.NoticeError(errRuntimeNotFound)
282             txn.End()
283             continue
284         }
285         logger.Info("Starting workflow", "workflow", tasks[0].Metadata.Workflow, "runtime", reName)
286         if err := runtime.StartWorkflow(ctx, tasks); err != nil {
287             logger.Error(err.Error())
288             txn.NoticeError(err)
289         }
290         txn.End()
291     }
292
293     // process deletion tasks
294     for _, tasks := range groupTasks(deletionTasks) {
295         reName := tasks[0].Metadata.ReName
296         runtime, ok := runtimes[reName]
297         txn := newTransaction(monitor, "terminate-workflow", tasks[0].Metadata.Workflow, reName)
298
299         if !ok {
300             logger.Error("Runtime not found", "workflow", tasks[0].Metadata.Workflow, "runtime", reName)

```

## Рисунок 3.13 –agent.go – виконання задач

Для спрощення створення та видалення Kubernetes ресурсів було створено невеликий набір утилітарних функцій. Вони були реалізовані як частина пакету Kubernetes у файлі kubernetes.go код якого зображено на рисунках 3.14 та 3.15.

```

84  func (k kube) CreateResource(ctx context.Context, spec interface{}) error {
85
86      bytes, err := json.Marshal(spec)
87      if err != nil {
88          return err
89      }
90
91      obj, _, err := kubeDecode(bytes, nil, nil)
92      if err != nil {
93          return err
94      }
95
96      var namespace string
97      switch obj := obj.(type) {
98      case *v1.PersistentVolumeClaim:
99          namespace = obj.ObjectMeta.Namespace
100         _, err = k.client.CoreV1().PersistentVolumeClaims(namespace).Create(ctx, obj, metav1.CreateOptions{})
101         if err != nil {
102             return err
103         }
104         k.logger.Info("PersistentVolumeClaim has been created")
105
106      case *v1.Pod:
107          namespace = obj.ObjectMeta.Namespace
108          _, err = k.client.CoreV1().Pods(namespace).Create(ctx, obj, metav1.CreateOptions{})
109          if err != nil {
110              return err
111          }
112          k.logger.Info("Pod has been created")
113      }
114      return err
115  }
116  }

```

Рисунок 3.14 –kubernetes.go – створення ресурсу

```

118 func (k kube) DeleteResource(ctx context.Context, opt DeleteOptions) error {
119     switch opt.Kind {
120     case task.TypeDeletePVC:
121         err := k.client.CoreV1().PersistentVolumeClaims(opt.Namespace).Delete(ctx, opt.Name, metav1.DeleteOptions{})
122         if err != nil {
123             return err
124         }
125         k.logger.Info("PersistentVolumeClaim has been deleted")
126
127     case task.TypeDeletePod:
128         err := k.client.CoreV1().Pods(opt.Namespace).Delete(ctx, opt.Name, metav1.DeleteOptions{})
129         if err != nil {
130             return err
131         }
132         k.logger.Info("Pod has been deleted")
133     }
134
135     return nil
136 }
137 }
138

```

Рисунок 3.15 –kubernetes.go – видалення ресурсу

У процесі виконання створюється Kubernetes ресурс типу Workflow. Цей ресурс не підтримується з коробки, тому його підтримку потрібно оголосити під час встановлення агенту. Ресурси які не постачаються з коробки прийнято називати CRD – Custom Resource Definition. CRD – це розширення API Kubernetes.

Ресурс — це endpoint в Kubernetes API, який зберігає набір об'єктів API певного типу; наприклад, вбудований ресурс pods містить колекцію об'єктів Pod.

Спеціальний ресурс — це розширення Kubernetes API, яке не обов'язково доступне в стандартній установці Kubernetes. Він являє собою налаштування конкретної інсталяції Kubernetes. Однак багато основних функцій Kubernetes тепер створені за допомогою спеціальних ресурсів, що робить Kubernetes більш модульним. Спеціальні ресурси можуть з'являтися і зникати в запущеному кластері завдяки динамічній реєстрації, а адміністратори кластеру можуть оновлювати власні ресурси незалежно від самого кластера.

Після встановлення спеціального ресурсу користувачі можуть створювати його об'єкти та отримувати доступ до нього за допомогою kubectl, так само, як і для вбудованих ресурсів, таких як Pods.

Самі по собі спеціальні ресурси дозволяють зберігати та отримувати структуровані дані. Коли ви поєднуєте користувацький ресурс із користувацьким контролером, користувацькі ресурси забезпечують справжній декларативний API. Декларативний API Kubernetes забезпечує поділ відповідальності. Ви оголошуєте потрібний стан свого ресурсу. Контролер Kubernetes синхронізує поточний стан об'єктів Kubernetes із вашим оголошеним бажаним станом. Це на відміну від імперативного API, де ви інструктуєте сервер, що робити.

Метою створення Workflow ресурсу у кластері є збереження прогресу виконання пайплайну у випадку припинення його роботи з будь-яких причин.

```
! workflow.yml
1  apiVersion: argoproj.io/v1alpha1
2  kind: Workflow
3  metadata:
4  annotations:
5    pod-name-format: v1
6    creationTimestamp: "2021-12-07T16:03:34Z"
7    generateName: test-
8    generation: 7
9  labels:
10   test: test
11   name: test-tshvz
12   namespace: argo
13   resourceVersion: "2859"
14   uid: 91281f4e-4bd5-424c-9bc3-5cd48ce68307
15 spec:
16   arguments: {}
17   entrypoint: main
18   steps:
19   - name: main
20     commands: []
21     arguments:
22       parameters:
23       - name: param1
24         value: param1
25   status:
26     conditions:
27     - status: "False"
28       type: PodRunning
29     - status: "True"
30       type: Completed
31     finishedAt: "2021-12-07T16:05:35Z"
32     message: FInished successfully
33     nodes:
34     workflow-of-workflows-tshvz:
35       displayName: main
36       finishedAt: "2021-12-07T16:05:35Z"
37       id: main-tshvz
38       message: Success
39       name: main
40       outboundNodes:
41       - workflow-of-workflows-tshvz-933384804
42       phase: Success
43       progress: 1/1
44       resourcesDuration:
45         cpu: 41
46         memory: 25
47       startedAt: "2021-12-07T16:04:23Z"
48       templateName: main
49       templateScope: local/main
50       type: Steps
51     phase: Success
52     progress: 3/3
53     resourcesDuration:
54       cpu: 41
55       memory: 25
56     startedAt: "2021-12-07T16:04:23Z"
57
```

Приклад ресурсу наведено на рисунку 3.16.

Workflow містить інформацію про задачу та прогрес її виконання.

### Рисунок 3.16 – приклад створеного ресурсу Workflow

#### 3.5 Розробка серверного додатку

Під час розробки серверного додатку було використано TypeScript, що надає можливості статичної типізації коду, який в результаті буде компільовано в NodeJS.

В якості серверного фреймворку було використано NestJS.

Nest (NestJS) — це платформа для створення ефективних, масштабованих додатків Node.js на стороні сервера. Він використовує прогресивний JavaScript, побудований на основі TypeScript і повністю підтримує його (проте все ще дозволяє розробникам писати на чистому JavaScript) і поєднує елементи ООП (об'єктно-орієнтоване програмування), ФП (функціональне програмування) і ФРП (функціональне реактивне програмування).

Ініціалізацію NestJS дотаку зображено на рисунку 3.17:

```
1 import { ValidationPipe } from '@nestjs/common';
2 import { NestFactory } from '@nestjs/core';
3 import { useContainer } from 'class-validator';
4 import { TrimStringsPipe } from './modules/common/transformer/trim-strings.pipe';
5 import { AppModule } from './modules/main/app.module';
6 import { setupSwagger } from './swagger';
7
8 async function bootstrap() {
9   const app = await NestFactory.create(AppModule);
10  setupSwagger(app);
11  app.enableCors();
12  app.useGlobalPipes(new TrimStringsPipe(), new ValidationPipe());
13  useContainer(app.select(AppModule), { fallbackOnErrors: true });
14  await app.listen(3000);
15 }
16 bootstrap();
```

Рисунок 3.17 – Ініціалізація додатку, старт серверу

При роботі з NestJS для аутентифікації запитів використовуються Guards. Guards - це класи які імплементують метод canActivate. Приклад створеного Guard продемонстровано на рисунку 3.18



```

7
8  @Injectable()
9  export class RolesGuard implements CanActivate {
10     constructor(
11         private reflector: Reflector
12     ) {}
13
14     private static isUserPermitted(permittedRoles: string[] = [], userRoles: string[] = []): boolean {
15         return userRoles.some(role => permittedRoles.includes(role))
16     }
17
18     canActivate(context: ExecutionContext): boolean {
19         const permittedRoles = this.reflector.get<string[]>('roles', context.getHandler())
20         if (isEmpty(permittedRoles)) return true
21
22         const authEntity = AuthenticatedEntity.extractAuthEntityFromRequestContext(context)
23         const userRoles = authEntity.isAccountAdmin() ? [ Roles.ADMIN, Roles.USER ] : [ Roles.USER ]
24
25         return RolesGuard.isUserPermitted(permittedRoles, userRoles)
26     }
27 }
28

```

Рисунок 3.18 – Реалізація Guard в NestJS

В рамках серверу було створено 2 ендпоінти для створення задач

```

33
34  @Get('tasks')
35  async getTasks(
36      @Args('runtime') runtime: string,
37  ): Promise<DbTask | undefined> {
38      try {
39          const res = await this.service.findMany({
40              filter: {
41                  account: this.contextService.getActiveAccountId(),
42                  runtime: runtime,
43                  phase: "WAITING_FOR_EXECUTION"
44              },
45              sort: {
46                  // old one first
47                  created_at: 1
48              }
49          })
50      } catch (err) {
51          logger.error(err)
52          throw new GatewayInternalError('Failed to fetch tasks for runtime ' + runtime)
53      }
54  }
55

```

Рисунок 3.19 – Обробка запиту на отримання задач

```

56  @Post('task')
57  async getWorkflows(
58      @Args('yaml') yaml: string,
59  ): Promise<CreatedTask> {
60      try {
61          return await this.service.createTask(this.contextService.getActiveAccountId(), yaml)
62      } catch (err) {
63          logger.error(err)
64          throw new GatewayInternalError('Failed to create a task')
65      }
66  }
67

```

Рисунок 3.20 – Обробка запиту на створення задачі

(рисунок 3.19), та зчитування їх результатів (рисунок 3.20).

Працювати з БД можна і напряму, за допомогою нативного драйвера MongoDB, проте набагато краще використовувати ODM.

ODM (англ. Object-Document Mapping, укр. Об'єктно-документне відображення, або перетворення) - технологія програмування, яка дозволяє зв'язати БД з концепціями об'єктно-орієнтованих мов програмування, створюючи «віртуальну об'єктну базу даних». Готове та популярне рішення для Node.js та NestJS це Mongoose. Приклад задачі описаної за допомогою Mongoose схеми зображено на рисунку 3.21:

```

226 @Schema({ _id: false })
227 export class TaskStatus {
228   @Prop()
229   createdAt?: Date
230
231   @Prop()
232   startedAt?: Date
233
234   @Prop()
235   finishedAt?: Date
236
237   @Prop()
238   phase?: Phases
239
240   @Prop()
241   progress?: string // change to progress type
242
243   @Prop({ type: mongoose.Schema.Types.Mixed })
244   nodes?: NodeStatus[]
245
246   @Prop()
247   message?: string
248
249   @Prop()
250   statusesHistory?: string // change to type statusesHistory
251 }
252
253 export const TaskStatusSchema = SchemaFactory.createClass(TaskStatus)
254
255 @Schema({ collection: 'tasks', ...TIMESTAMPS })
256 export class TaskEntity extends BaseK8sEntity {
257   @Prop({ type: WorkflowSpecSchema })
258   spec: TaskSpec
259
260   @Prop({ type: TaskStatusSchema })
261   status: TaskStatus
262
263   @Prop({ type: mongoose.Schema.Types.Mixed }) //todo fix mixed type
264   eventsPayloadData?: EventPayloadData[]
265
266   @Prop({ type: mongoose.Schema.Types.Array })
267   eventsPayload?: string[]
268
269   @Prop({ type: EntityIdSchema })
270   pipeline?: EntityId
271
272   @Prop({ type: mongoose.Schema.Types.Mixed })
273   actualManifest: any
274
275   @Prop({ type: [ String ], index: true })
276   projects: string[]
277 }

```

Рисунок 3.21 – Mongoose схема задачі

Для успішного розгортання серверу необхідно описати Dockerfile й створити імідж, на основі якого в майбутньому будуть створені контейнери серверу. Приклад Dockerfile наведено на рисунку 3.22:

```
1 FROM node:14.17.0-alpine as base
2 WORKDIR /code
3
4 FROM base as build
5
6 COPY lerna.json .
7 COPY tsconfig.json .
8 COPY package.json .
9 COPY yarn.lock .
10
11 COPY libs libs
12
13 ARG PACKAGE
14 COPY ${PACKAGE} ${PACKAGE}
15
16 # Install with dev dependencies
17 RUN yarn install
18 RUN yarn build
19 # Clean up dev dependencies
20 RUN yarn install --production
21
22 FROM base
23 ARG PACKAGE
24
25 COPY --from=build /code/libs /code/libs
26 COPY --from=build /code/node_modules /code/node_modules
27 COPY --from=build /code/${PACKAGE} /code/${PACKAGE}
28
29 WORKDIR /code/${PACKAGE}
30 CMD [ "yarn", "start:prod" ]
31
```

Рисунок 3.22 – Dockerfile серверу

Як веб сервер було обрано Nginx. Використання проксі-сервера, Nginx, дозволяє рівномірно розподілити навантаження, зручно та правильно працювати зі статичними ресурсами і реалізувати протокол Transport Layer Security (TLS).

NGINX прискорює доставку вмісту та додатків, покращує безпеку, полегшує доступність і масштабованість для найбільш завантажених веб-сайтів в Інтернеті.

Оскільки ми використовуємо контейнеризацію, принцип дій буде приблизно аналогічний, проте засоби відрізнятимуться. Nginx буде встановлений за допомогою готового image для Docker.\

Приклад створеної nginx конфігурації наведено на рисунку 3.23:

```
nginx.conf
1  server {
2      listen          443 ssl http2;
3      listen          [::]:443 ssl http2;
4      server_name     example.com;
5      set             $base /var/www/example.com;
6      root            $base/public;
7
8      # SSL
9      ssl_certificate /etc/letsencrypt/live/example.com/fullchain.pem;
10     ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;
11     ssl_trusted_certificate /etc/letsencrypt/live/example.com/chain.pem;
12
13     # security
14     include          nginxconfig.io/security.conf;
15
16     # index.js
17     index            index.js;
18
19     # index.js fallback
20     location / {
21         try_files $uri $uri/ /index.js?$query_string;
22     }
23
24     # additional config
25     include          nginxconfig.io/general.conf;
26
27     # handle .php
28     location ~ /\.php$ {
29         fastcgi_pass unix:/var/run/php/php-fpm.sock;
30         include      nginxconfig.io/php_fastcgi.conf;
31     }
32 }
33
34 # subdomains redirect
35 server {
36     listen          443 ssl http2;
37     listen          [::]:443 ssl http2;
38     server_name     *.example.com;
39
40     # SSL
41     ssl_certificate /etc/letsencrypt/live/example.com/fullchain.pem;
42     ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;
43     ssl_trusted_certificate /etc/letsencrypt/live/example.com/chain.pem;
44     return          301 https://example.com$request_uri;
45 }
46
47 # HTTP redirect
48 server {
49     listen          80;
50     listen          [::]:80;
51     server_name     .example.com;
52     include          nginxconfig.io/letsencrypt.conf;
53
54     location / {
55         return 301 https://example.com$request_uri;
56     }
57 }
58
```

Рисунок 3.23 – Nginx конфігурація

### 3.6 Висновки

В цьому розділі було здійснено варіантний аналіз і обґрунтування вибору засобів реалізації програмного продукту. В наслідок чого було обрано мови програмування NodeJS та Golang, середовище розробки Visual Studio Code, база даних MongoDB, nginx в якості зворотнього проксі серверу.

Також було розроблено такі модулі програмного додатку як: прослуховування та логування вихідних даних кроків, виконання пайплайну в середовищі Kubernetes та серверну частину додатку.

## 4 ТЕСТУВАННЯ ПРОГРАМИ

### 4.1 Тестування програмного забезпечення

Тестування програмного забезпечення - процес технічних досліджень, призначений для отримання інформації про якість програмних продуктів; методи контролю якості, які використовують остаточний набір тестів, вибраних певним чином, для перевірки відповідності між фактичною та очікуваною поведінкою програми [20].

При тестуванні програмних систем застосовуються методи "білої" та "чорної скриньки". Ці методи – динамічні методи тестування програмного забезпечення.

Метод «білої скриньки» тестує внутрішні структури або роботу програми, на відміну від її функціональності (тобто тестування в чорному ящику). Для створення тестів використовуються внутрішні перспективи системи, а також навички програмування.

Зазвичай тестування «білої скриньки» засноване на аналізі керуючої структури програми. Програма вважається повністю перевіреною, якщо проведено вичерпне тестування маршрутів (шляхів) її графа управління [21].

Методи тестування тестування в білій коробці включають такі критерії покриття коду:

- тестування контрольного потоку;
- тестування потоку даних;
- гілкове тестування;
- покриття виразів;
- покриття умов;
- змінена умова/покриття рішення;
- тестування основного шляху;
- тестування побічних шляхів.

Тестування «чорної скриньки» - це метод тестування програмного забезпечення, який вивчає функціональність програми, не заглядаючи у її внутрішні структури чи роботи. Цей метод тестування може бути застосований практично до кожного рівня тестування програмного забезпечення: блочного, інтеграційного, системного. Іноді це називається тестуванням на основі специфікацій.

Тестування «чорної скриньки» забезпечує пошук наступних категорій помилок:

- некоректних чи відсутніх функцій;
- помилок у зовнішніх структурах даних або в доступі до зовнішньої бази даних;
- помилок характеристик.

Розроблений програмний продукт перевірятиметься на відповідність технічним вимогам, безпечність, коректність роботи у середовищі Windows 10 за допомогою браузера Google Chrome.

Першим кроком для роботи з програмним продуктом є встановлення агенту у кластері користувача. Це можна зробити за допомогою маніфестів зображених на рисунках 4.1 та 4.2.

```
50   apiVersion: v1
51   kind: ConfigMap
52   metadata:
53     name: agent-config
54     namespace: default
55     resourceVersion: "516"
56     uid: b4952dc3-d670-11e5-8cd0-68f728db1985
57   data:
58     apiToken: <api-token>
59     interval: <interval>
60
```

Рисунок 4.1 – ConfigMap агента



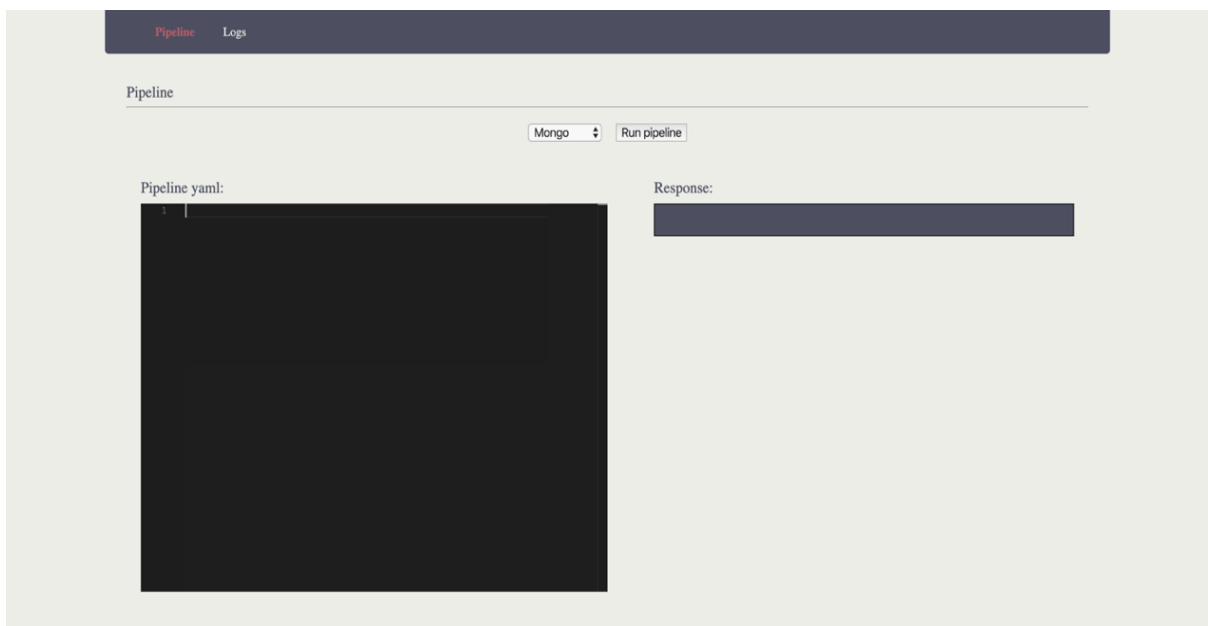
```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    labels:
5      version: 1.0
6    name: runner-agent
7    namespace: default
8  spec:
9    replicas: 1
10   revisionHistoryLimit: 5
11   strategy:
12     rollingUpdate:
13       maxSurge: 50%
14       maxUnavailable: 50%
15     type: RollingUpdate
16   template:
17     spec:
18       volumes:
19         - name: runnerconf
20           secret:
21             secretName: runnerconf
22       containers:
23         image: melden19/runner-agent
24         ports:
25         - containerPort: 8080
26           protocol: TCP
27         readinessProbe:
28           httpGet:
29             path: /health
30             port: 8080
31           periodSeconds: 5
32           timeoutSeconds: 5
33           successThreshold: 1
34           failureThreshold: 5
35         volumeMounts:
36         - name: runnerconf
37           mountPath: "/etc/secrets"
38           readOnly: true
39         imagePullPolicy: Always
40         name: {{ .AppName }}
41       resources:
42     {{ toYaml .Runner.Resources | indent 10 }}
43   securityContext:
44     runAsUser: 10001
45     runAsGroup: 10001
46     fsGroup: 10001

```

Рисунок 4.2 –Deployment агента

Протестуємо відкриття вебсайту у браузері. (рисунок 4.3).



### Рисунок 4.3 – Сторінка Pipeline

В блоці Pipeline yaml потрібно ввести інформацію про контейнери(steps) які бажаємо запустити у форматі yaml й, вибравши бажане сховище з випадуючого списку, натиснути «Run pipeline» (рисунок 4.4).

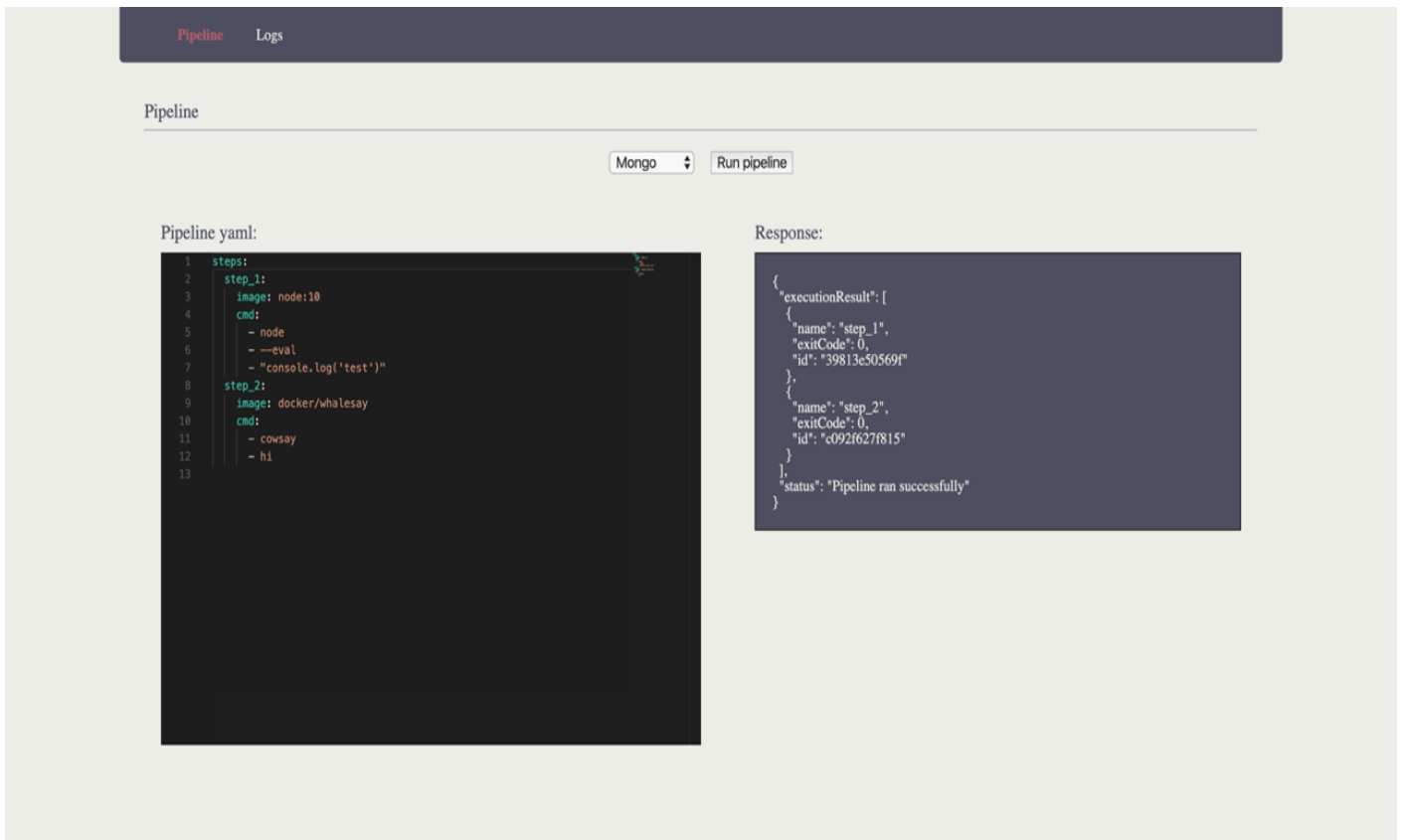


Рисунок 4.4 – Сторінка Pipeline після виконання пайплайну

Після цього було створено задачу у базі даних MongoDB. Далі встановлений агент повинен буде отримати цю задачу і розпочати її виконувати. На рисунку 4.5 зображено логи робочого агента. По них видно, що він з фіксованим інтервалом намагається отримати задачу з серверу.

```

time="2021-12-19T12:29:53.239Z" level=info msg="Get leases 200"
time="2021-12-19T12:29:53.254Z" level=info msg="Update leases 200"
time="2021-12-19T12:29:58.306Z" level=info msg="Get leases 200"
time="2021-12-19T12:29:58.317Z" level=info msg="Update leases 200"
time="2021-12-19T12:30:03.324Z" level=info msg="Get leases 200"
time="2021-12-19T12:30:03.339Z" level=info msg="Update leases 200"
time="2021-12-19T12:30:08.375Z" level=info msg="Get leases 200"
time="2021-12-19T12:30:08.384Z" level=info msg="Update leases 200"
time="2021-12-19T12:30:13.401Z" level=info msg="Get leases 200"
time="2021-12-19T12:30:13.415Z" level=info msg="Update leases 200"
time="2021-12-19T12:30:18.468Z" level=info msg="Get leases 200"
time="2021-12-19T12:30:18.478Z" level=info msg="Update leases 200"
time="2021-12-19T12:30:23.484Z" level=info msg="Get leases 200"
time="2021-12-19T12:30:23.497Z" level=info msg="Update leases 200"
time="2021-12-19T12:30:28.540Z" level=info msg="Get leases 200"
time="2021-12-19T12:30:28.550Z" level=info msg="Update leases 200"
time="2021-12-19T12:30:29.024Z" level=info msg="Match workflowtemplates 200"
time="2021-12-19T12:30:33.573Z" level=info msg="Get leases 200"
time="2021-12-19T12:30:33.588Z" level=info msg="Update leases 200"
time="2021-12-19T12:30:38.637Z" level=info msg="Get leases 200"
time="2021-12-19T12:30:38.644Z" level=info msg="Update leases 200"
time="2021-12-19T12:30:43.652Z" level=info msg="Get leases 200"
time="2021-12-19T12:30:43.666Z" level=info msg="Update leases 200"
time="2021-12-19T12:30:44.668Z" level=info msg="List workflows 200"
time="2021-12-19T12:30:44.669Z" level=info msg="healthz age=5m0s err=<nil>" instanceID= labelSelector=!workflows.argoproj.io/phase,!workflows.argoproj.io
time="2021-12-19T12:30:48.723Z" level=info msg="Get leases 200"
time="2021-12-19T12:30:48.730Z" level=info msg="Update leases 200"
time="2021-12-19T12:30:53.750Z" level=info msg="Get leases 200"
time="2021-12-19T12:30:53.766Z" level=info msg="Update leases 200"
time="2021-12-19T12:30:58.833Z" level=info msg="Get leases 200"
time="2021-12-19T12:30:58.841Z" level=info msg="Update leases 200"
time="2021-12-19T12:31:03.846Z" level=info msg="Get leases 200"
time="2021-12-19T12:31:03.861Z" level=info msg="Update leases 200"
time="2021-12-19T12:31:03.862Z" level=info msg="Watch pods 200"
time="2021-12-19T12:31:08.931Z" level=info msg="Get leases 200"
time="2021-12-19T12:31:08.938Z" level=info msg="Update leases 200"
time="2021-12-19T12:31:13.957Z" level=info msg="Get leases 200"
time="2021-12-19T12:31:13.971Z" level=info msg="Update leases 200"
time="2021-12-19T12:31:19.014Z" level=info msg="Get leases 200"
time="2021-12-19T12:31:19.029Z" level=info msg="Update leases 200"
time="2021-12-19T12:31:24.039Z" level=info msg="Get leases 200"
time="2021-12-19T12:31:24.053Z" level=info msg="Update leases 200"
time="2021-12-19T12:31:29.090Z" level=info msg="Get leases 200"
time="2021-12-19T12:31:29.104Z" level=info msg="Update leases 200"
time="2021-12-19T12:31:34.130Z" level=info msg="Get leases 200"
time="2021-12-19T12:31:34.144Z" level=info msg="Update leases 200"
time="2021-12-19T12:31:39.191Z" level=info msg="Get leases 200"
time="2021-12-19T12:31:39.208Z" level=info msg="Update leases 200"

```

Рисунок 4.5 – Логи роботи агента

Після отримання задачі буде розпочато її виконання. Під час виконання задачі буде створено CRD Workflow з описом пайплайну. Відповідно до нього агент буде послідовно виконувати кроки пайплайну. Протягом циклу виконання кроків агент буде оновлювати ресурс workflow з метою збереження прогресу виконання пайплайну на випадок несподіваного припинення його роботи й початку роботи з початку.

Переконайтеся, що ресурс було створено без помилок й агент розпочав його виконання можна за допомогою команд зображених на рисунку 4.6 та 4.7:

```

~ > k get workflow
NAME          STATUS   AGE
pipeline-1    Running  2s
~ >

```

Рисунок 4.6 – отримання ресурсу workflow у фазі Running

```

1  apiVersion: v1alpha1
2  kind: Workflow
3  metadata:
4    name: pipeline-1
5  spec:
6    entrypoint: steps
7    templates:
8      - name: steps
9        steps:
10         - - name: step-1
11             |   template: step-1
12         - - name: step-2
13             |   template: step-2
14       - name: step-1
15         container:
16           image: node:10
17           command: [node]
18           args: ["--eval", "console.log('test')"]
19           resources:
20             limits:
21               memory: 32Mi
22               cpu: 100m
23       - name: step-2
24         container:
25           image: docker/whalesay
26           command: [cowsay]
27           args: ["hello world"]
28           resources:
29             limits:
30               memory: 32Mi
31               cpu: 100m
32

```

Рисунок 4.7 – Створений ресурс Workflow

Далі для кожного кроку буде запущено Pod, в якому буде виконуватися 2 контейнера:

- контейнер крок;
- контейнер слухач логів.

Перевірити чи кроки було виконано успішно можна за допомогою команди `kubectl get po | grep pipeline-1`, результат виконання якої зображено на рисунку 4.8.

```

~ > k get pods | grep pipeline-1
pipeline-1-3893458983          0/2   Completed   0          4m50s
pipeline-1-809788873         0/2   Completed   0          5m
~ >

```

Рисунок 4.8 – Отримання списку завершених кроків (Pods)

Отримавши маніфест одного з Pods за допомогою команди “`kubectl get po/pipeline-1-809788873 -o yaml`” (рис. 4.9) можна побачити, що було успішно виконано 2 контейнера.

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    annotations:
5      kubernetes.io/psp: eks.privileged
6      steps-name: step-1
7    creationTimestamp: "2021-12-19T11:56:18Z"
8    name: pipeline-1-809788873
9    namespace: aws-local-runtime
10   resourceVersion: "814562"
11   uid: 880bc800-470d-4d51-8aa3-f668ce43a8f6
12  spec:
13   containers:
14   - command:
15     - listen
16     image: melden19/container-logger
17     name: logger
18     volumeMounts:
19     - mountPath: /var/run/docker.sock
20       name: dockersock
21   - args:
22     - --eval
23     - console.log('test')
24     command:
25     - node
26     image: node:10
27     imagePullPolicy: IfNotPresent
28     name: step-1
29     volumeMounts:
30     - mountPath: /data
31       name: data
32       subPath: step0
33   volumes:
34   - name: data
35     persistentVolumeClaim:
36       claimName: my-pvc
37   - name: dockersock
38     hostPath:
39       path: /var/run/docker.sock
40  status:
41   hostIP: 192.168.55.21
42   phase: Succeeded
43   podIP: 192.168.47.109
44   podIPs:
45   - ip: 192.168.47.109
46   qosClass: Burstable
47   startTime: "2021-12-19T11:56:18Z"

```

Рисунок 4.9 – Отримання маніфесту завершеного кроку (Pod)

Тепер повернувшись до клієнта й перейшовши на сторінку «Logs» у поле «Step name» вводимо ім'я кроку й вибираємо сховище, у цьому випадку «Mongo» (рисунок 4.10).

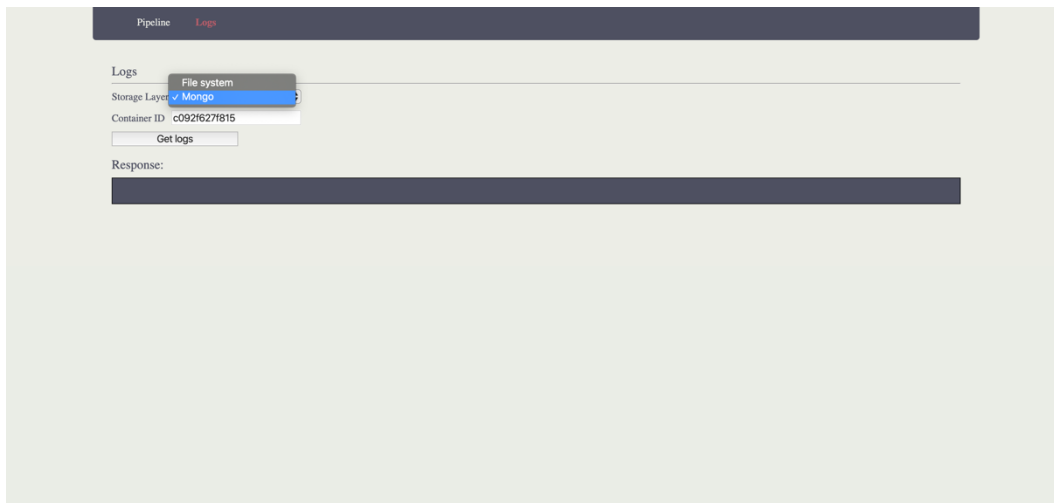


Рисунок 4.10 – Сторінка Logs

Натиснувши кнопку «Get logs» можемо отримати вихідні дані контейнері відповідно до сказаного id та сховища (рисунок 4.11).

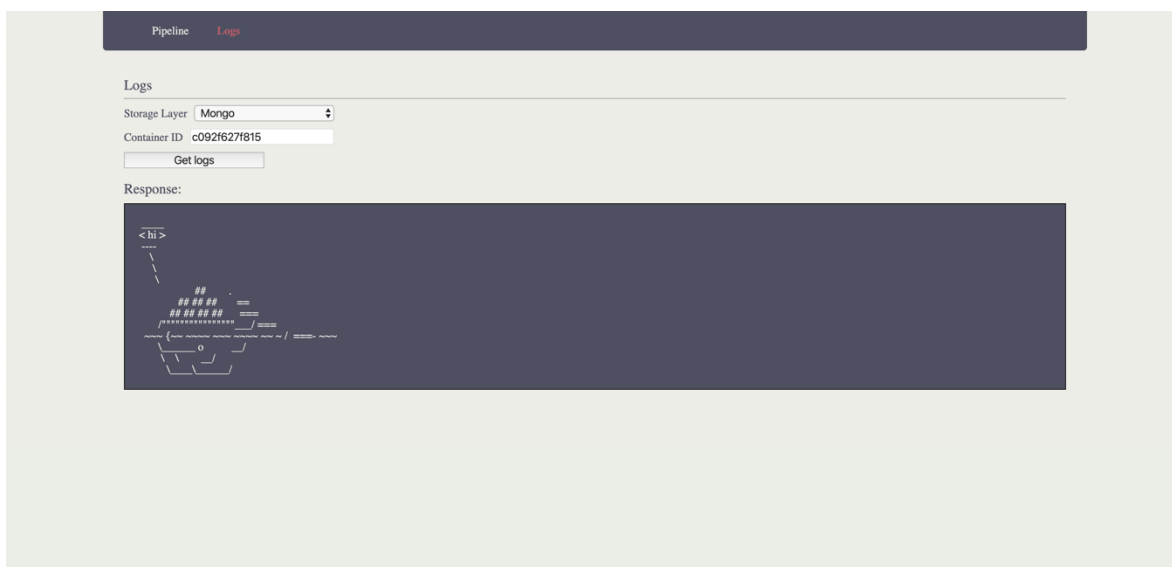


Рисунок 4.11 – Відображення вихідних даних кроку



#### 4.2 Розробка інструкції користувача

Оскільки підбір мови програмування опирався на швидкість й невимогливість в ресурсах, програмний модуль не є вимогливим потужної технічної конфігурації комп'ютера. Найбільше на швидкість роботи буде впливати частота роботи процесора. Так як Node.js є подійно-орієнтованою мовою програмування й працює лише з одним потоком, важлива саме максимальна частота ядра процесора.

Протестувавши програмну систему можна виділити таку послідовність її використання:

1. Реєстрація на платформі.
2. Встановлення агенту на власний Kubernetes кластер.
3. Опис пайплайну, вибір типу сховища.
4. Запуск пайплайну.
5. Очікування завершення пайплайну.
6. Отримання ід запущених контейнерів.
7. Введення назви кроку, вибір типу сховища.
8. Отримання логів.

У таблиці 4.1 та 4.2 наведено мінімальну та рекомендовану конфігурацію персонального комп'ютера для запуску програми.

Таблиця 4.1 – Мінімальна конфігурація:

Тип процесора	32-розрядний (x86) або <u>64-розрядний (x64)</u> процесор з тактовою частотою 1 ГГц
Об'єм оперативної пам'яті	1 ГБ для 32-розрядної системи і 2 ГБ для 64-розрядної системи
Розмір жорсткого диску	16 ГБ для 32-розрядної системи і 22 ГБ для 64-розрядної системи



Таблиця 4.2 – Рекомендована конфігурація:

Тип процесора	32-розрядний (x86) або <u>64-розрядний (x64)</u> процесор з тактовою частотою 2 ГГц
Об'єм оперативної пам'яті	2 ГБ для 32-розрядної системи і 4 ГБ для 64-розрядної системи
Розмір жорсткого диску	20 ГБ для 32-розрядної системи і 25 ГБ для 64-розрядної системи

Програма повинна працювати під управлінням сімейства операційних систем Linux та MacOS.

Для роботи з програмою потрібно встановити NodeJS та npm пакети вказані у файлі package.json.

Після цього користувач може запускати і працювати з програмним модулем.

### 4.3 Висновки

Тестування програми показало повну її працездатність та відповідність поставленому технічному завданню. Розроблено інструкцію користувача по використанню й встановленню програмного продукту.

## 5 ЕКОНОМІЧНА ЧАСТИНА

### 5.1 Оцінювання комерційного потенціалу розробки

Метою проведення комерційного та технологічного аудиту є оцінювання комерційного потенціалу впровадження програмного забезпечення у середовищі Kubernetes для інтегрування програмних продуктів.

Для проведення технологічного аудиту було залучено 3-х незалежних експертів Вінницького національного технічного університету к.т.н., доцента Кательнікова Д. І., к.т.н., доцента Черноволик Г. О. з кафедри програмного забезпечення та кафедри комп'ютерних наук к.т.н., доцента Арсенюк І. Р. Технологічний аудит проведемо з використанням таблиці 4.1 [22] де за п'ятибальною шкалою використовуючи 12 критеріїв оцінемо комерційний потенціал.

Таблиця 5.1 – Рекомендовані критерії оцінювання комерційного потенціалу розробки та їх можлива бальна оцінка

Критерії оцінювання та бали (за 5-ти бальною шкалою)					
Кри-терій	0	1	2	3	4
Технічна здійсненність концепції:					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено роботоздатність продукту в реальних умовах
Ринкові переваги (недоліки):					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів
4	Технічні та споживчі властивості продукту значно гірші, ніж в аналогів	Технічні та споживчі властивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в аналогів	Технічні та споживчі властивості продукту значно кращі, ніж в аналогів

Продовження табл. 5.1

5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
Ринкові перспективи					
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практична здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витратити значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово-промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві
11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

Таблиця 5.2 – Рівні комерційного потенціалу розробки

Середньоарифметична сума балів СБ, розрахована на основі висновків експертів	Рівень комерційного потенціалу розробки
0-10	Низький
11-20	Ниже середнього
21-30	Середній
31-40	Вище середнього
41-48	Високий

В таблиці 5.3 наведено результати оцінювання експертами комерційного потенціалу розробки.

Таблиця 5.3 – Результати оцінювання комерційного потенціалу розробки

Критерії	Прізвище, ініціали, посада експерта		
	1. Кательніков Д. І.	2. Арсенюк І. Р.	3. Черноволик Г. О.
	Бали, виставлені експертами:		
1	3	3	3
2	2	2	2
3	4	2	4
4	2	3	3
5	3	4	4
6	4	2	4
7	3	2	2
8	4	3	4
9	4	3	3
10	3	4	2
11	2	3	3
12	4	2	4
Сума балів	СБ <sub>1</sub> =39	СБ <sub>2</sub> =35	СБ <sub>3</sub> =38
Середньоарифметична сума балів $\overline{СБ}$	$\overline{СБ} = \frac{\sum_1^3 СБ_i}{3} = \frac{38 + 33 + 38}{3} = 32$		

Середньоарифметична сума балів, розрахована на основі висновків експертів склала 32 бали, що згідно таблиці 5.2 вважається, що рівень комерційного потенціалу проведених досліджень є вище середнього.

Основними клієнтами можуть виступати власники програмних продуктів, які зацікавлені у використанні CI/CD методології під час розробки власного програмного забезпечення. В якості середовища виконання буде

використовуватися кластер користувача, що зменшить мінімальну ціну програмного продукту й тим самим зробить його більш привабливішим для потенційного покупця. Для впровадження даної розробки потрібно розгорнути сервер на публічній інфраструктурі та окрему програмну компоненту у кластері на стороні клієнта, яка буде отримувати задачі від сервера та виконувати пайплайни у кластері.

В якості аналога для розробки було обрано платформу AzureDevops. Основними недоліками аналога є відсутність можливості виконання задач у ізольованому середовищі виконання, наприклад у приватній мережі.

У розробці дана проблема вирішується шляхом встановлення агента на кластер користувача, який за допомогою постійного опитування сервера на предмет нових задач, виконує їх кластері. Таким чином у випадку якщо серверна частина працює у режимі SAAS, вона не має доступ до приватної мережі клієнта, але агент встановлений в кластері клієнта може надсилати запити до сервера й таким чином отримуючи задачі для виконання у приватній мережі клієнта.

Проведемо оцінку якості і конкурентоспроможності нової розробки порівняно з аналогом. В таблиці 5.4 наведені основні техніко-економічні показники аналога і нової розробки.

Таблиця 5.4 – Основні параметри нової розробки та товару-конкурента

Показник	Варіанти		Відносний показник якості	Коефіцієнт вагомості параметра
	Базовий (товар-конкурент)	Новий (інноваційне рішення)		
1	2	3	4	5
Середній час виконання тестового пайплайну (сек.)	100	80	1,25	40%
Кількість підтримуваних видів сховищ для вихідних даних пайплайну	1	2	2	20%
Відмовостійкість, %	70	90	1,3	40%
Час завантаження сторінки, мс	1000	500	2	10%

Проведемо оцінку якості продукції, яка є найефективнішим засобом забезпечення вимог споживачів та порівняємо її з аналогом.

Визначимо відносні одиничні показники якості по кожному параметру за формулами (4.1) та (4.2) і занесемо їх у відповідну колонку табл. 4.5.

$$q_i = \frac{P_{Hi}}{P_{Bi}} \quad (5.1)$$

або

$$q_i = \frac{P_{Bi}}{P_{Hi}} \quad (5.2)$$

де  $P_{Hi}$ ,  $P_{Bi}$  – числові значення  $i$ -го параметру відповідно нового і базового виробів.

$$q_1 = \frac{100}{80} = 1,25;$$

$$q_2 = \frac{2}{1} = 2;$$

$$q_4 = \frac{90}{70} = 1,3;$$

$$q_5 = \frac{1000}{500} = 2.$$

Відносний рівень якості нової розробки визначаємо за формулою:

$$K_{я.в.} = \sum_{i=1}^n q_i \cdot \alpha_i, \quad (5.3)$$

$$K_{я.в.} = 1,25 \cdot 0,4 + 2 \cdot 0,2 + 1,3 \cdot 0,4 + 2 \cdot 0,1 = 1,64$$

Відносний коефіцієнт показника якості нової розробки більший одиниці, отже нова розробка якісніший базового товару-конкурента.

Наступним кроком є визначення конкурентоспроможності товару. Конкурентоспроможність товару є головною умовою конкурентоспроможності підприємства на ринку і важливою основою прибутковості його діяльності.

Однією із умов вибору товару споживачем є збіг основних ринкових характеристик виробу з умовними характеристиками конкретної потреби

покупця. Такими характеристиками найчастіше вважають нормативні та технічні параметри, а також ціну придбання та вартість споживання товару.

В табл. 5.5 наведено технічні та економічні показники для розрахунку конкурентоспроможності нової розробки відносно товару-аналога, технічні дані взяті з попередніх розрахунків.

Таблиця 5.5 – Нормативні, технічні та економічні параметри нової розробки і товару-виробника

Показники	Варіанти	
	Базовий (товар-конкурент)	Новий (інноваційне рішення)
1	2	3
1. Нормативно-технічні показники		
Середній час виконання тестового пайплайну (сек.)	100	80
Кількість підтримуваних видів сховищ для вихідних даних пайплайну	1	2
Відмовостійкість, %	70	90
Час завантаження сторінки, мс	1000	500
2. Економічні показники		
Ціна придбання, грн.	10000	8000

Загальний показник конкурентоспроможності інноваційного рішення ( $K$ ) з урахуванням вищезазначених груп показників можна визначити за формулою:

$$K = \frac{I_{m.n.}}{I_{e.n.}}, \quad (5.4)$$

де  $I_{m.n.}$  – індекс технічних параметрів;  $I_{e.n.}$  – індекс економічних параметрів.

Індекс технічних параметрів є відносним рівнем якості інноваційного рішення. Індекс економічних параметрів визначається за формулою (4.5)

$$I_{e.n.} = \frac{\sum_{i=1}^n P_{Hei}}{\sum_{i=1}^n P_{Bei}}, \quad (5.5)$$

де  $P_{Hei}$ ,  $P_{Bei}$  – економічні параметри (ціна придбання та споживання товару) відповідно нового та базового товарів.

$$I_{e.n.} = \frac{8000}{10000} = 0,8;$$

$$K = \frac{1,64}{0,8} = 2,05.$$

Зважаючи на розрахунки, можна зробити висновок, що нова розробка буде конкурентоспроможніше, ніж конкурентний товар.

## 5.2 Прогнозування витрат на виконання науково-дослідної роботи

Витрати, пов'язані з проведенням науково-дослідної роботи групуються за такими статтями: витрати на оплату праці, витрати на соціальні заходи, матеріали, паливо та енергія для науково-виробничих цілей, витрати на службові відрядження, програмне забезпечення для наукових робіт, інші витрати, накладні витрати.

1. Основна заробітна плата кожного із дослідників  $Z_0$ , якщо вони працюють в наукових установах бюджетної сфери визначається за формулою:

$$Z_0 = \frac{M}{T_p} * t \text{ (грн)} \quad (5.6)$$

де  $M$  – місячний посадовий оклад конкретного розробника (інженера, дослідника, науковця тощо), грн.;

$T_p$  – число робочих днів в місяці; приблизно  $T_p \approx 21...23$  дні;

$t$  – число робочих днів роботи дослідника.

Для розробки веб-платформи, що надає можливість керувати А/В експериментами на певному сайті необхідно залучити інженера програміста з



посадовим окладом 8000 грн. Кількість робочих днів у місяці складає 22, а кількість робочих днів програміста складає 20. Зведемо сумарні розрахунки до таблиця 5.6.

Таблиця 5.6 – Заробітна плата дослідника в науковій установі бюджетної сфери

Найменування посади	Місячний посадовий оклад, грн.	Оплата за робочий день, грн.	Число днів роботи	Витрати на заробітну плату грн.
Керівник	13000	590,9	5	2954
Інженер-програміст	8000	318,2	20	7272
Всього				10226

## 2. Розрахунок додаткової заробітної плати робітників

Додаткова заробітна плата  $Z_d$  всіх розробників та робітників, які приймали участь в розробці нового технічного рішення розраховується як 10 - 12 % від основної заробітної плати робітників.

На даному підприємстві додаткова заробітна плата начисляється в розмірі 10% від основної заробітної плати.

$$Z_d = (Z_o + Z_p) * \frac{N_{\text{дод}}}{100\%} \quad (5.7)$$

$$Z_d = 0,1 * 10226 = 1022(\text{грн})$$

3. Нарахування на заробітну плату  $N_{3\Pi}$  дослідників та робітників, які брали участь у виконанні даного етапу роботи, розраховуються за формулою (4.3):

$$N_{3\Pi} = (Z_o + Z_d) * \frac{\beta}{100} \quad (\text{грн}) \quad (5.8)$$

де  $Z_o$  – основна заробітна плата розробників, грн.;

$Z_d$  – додаткова заробітна плата всіх розробників та робітників, грн.;

$\beta$  – ставка єдиного внеску на загальнообов'язкове державне соціальне страхування, % .

Дана діяльність відноситься до бюджетної сфери, тому ставка єдиного внеску на загальнообов'язкове державне соціальне страхування буде складати 22%, тоді:

$$N_{зп} = (10226 + 1022) * \frac{22}{100} = 2474,5 \text{ (грн)}$$

4. Витрати на матеріали  $M$  та комплектуючі  $K$ , що були використані під час виконання даного етапу роботи, розраховуються по кожному виду матеріалів за формулою:

$$M = \sum_1^n N_i \cdot C_i \cdot K_i - \sum_1^n V_i \cdot C_v \text{ грн.}, \quad (5.9)$$

де  $N_i$  – витрати матеріалу  $i$ -го найменування, кг;

$C_i$  – вартість матеріалу  $i$ -го найменування, грн./кг.;

$K_i$  – коефіцієнт транспортних витрат,  $K_i = (1,1 \dots 1,15)$ ;

$V_i$  – маса відходів матеріалу  $i$ -го найменування, кг;

$C_v$  – ціна відходів матеріалу  $i$ -го найменування, грн/кг;

$n$  – кількість видів матеріалів.

Витрати на комплектуючі вироби, які використовують при виготовленні одиниці продукції, розраховуються, згідно їх номенклатури, за формулою:

$$K = \sum_{i=1}^n N_i \cdot C_i \cdot K_i, \quad (5.10)$$

де  $N_i$  – кількість комплектуючих  $i$ -го виду, шт.;

$C_i$  – покупна ціна комплектуючих  $i$ -го найменування, грн.;

$K_i$  – коефіцієнт транспортних витрат  $(1,1 \dots 1,15)$ .

Інформацію про використані матеріалита комплектуючі подамо у вигляді табл. 4.7.

Таблиця 5.7 – Матеріали, що використані на розробку

Найменування матеріалу	Ціна за одиницю, грн.	Витрачено	Вартість витраченого матеріалу, грн.
Папір	120	1	95
Ручка	15	1	15
CD-диск	12	1	12
Флешка	150	1	150
Всього			297
З врахуванням коефіцієнта транспортування			326,7

5. Програмне забезпечення для наукової роботи включає витрати на розробку та придбання спеціальних програмних засобів і програмного забезпечення необхідного для проведення дослідження.

Балансову вартість програмного забезпечення розраховують за формулою:

$$V_{\text{прг}} = \sum_{i=1}^k C_{i\text{прг}} \cdot C_{\text{пргі}} \cdot K_i, \quad (5.11)$$

де  $C_{i\text{прг}}$  – ціна придбання одиниці програмного засобу цього виду, грн;

$C_{\text{пргі}}$  – кількість одиниць програмного забезпечення відповідного найменування, які придбані для проведення досліджень, шт.;

$K_i$  – коефіцієнт, що враховує інсталяцію, налагодження програмного засобу тощо ( $K_i = 1, 10 \dots 1, 12$ ).

$k$  – кількість найменувань програмних засобів.

Отримані результати занесемо в таблицю 4.8.

Таблиця 5.8 – Витрати на придбання програмних засобів по кожному виду

Найменування устаткування	Кількість, шт	Ціна за одиницю, грн	Вартість
Ліцензія середовища розробки WebStrom	1	2000	2000
Microsoft Windows 10	1	3199	3199
Всього			5199

6. Амортизація обладнання, комп'ютерів та приміщень, які використовувались під час виконання даного етапу роботи

Дані відрахування розраховують по кожному виду обладнання, приміщенням тощо.

$$A = \frac{Ц \cdot T}{T_{кор} \cdot 12} \quad [грн], \quad (5.12)$$

де Ц – балансова вартість даного виду обладнання (приміщень), грн.;

$T_{кор}$  – час користування;

T – термін використання обладнання (приміщень), цілі місяці.

Згідно пункту 137.3.3 Податкового кодекса амортизація нараховується на основні засоби вартістю понад 2500 грн. В нашому випадку для написання магістерської роботи використовувався персональний комп'ютер вартістю 10000 грн.

$$A = \frac{10000 \cdot 1}{2 \cdot 12} = 416,67$$

7. До статті «Паливо та енергія для науково-виробничих цілей» відносяться витрати на всі види палива й енергії, що безпосередньо використовуються з технологічною метою на проведення досліджень.

$$B_e = \sum_{i=1}^n \frac{W_{yt} \cdot t_i \cdot C_e \cdot K_{впі}}{\eta_i} \quad (5.13)$$

де  $W_{yt}$  – встановлена потужність обладнання на певному етапі розробки, кВт;

$t_i$  – тривалість роботи обладнання на етапі дослідження, год;

$C_e$  – вартість 1 кВт-години електроенергії, грн;

$K_{впі}$  – коефіцієнт, що враховує використання потужності,  $K_{впі} < 1$ ;

$\eta_i$  – коефіцієнт корисної дії обладнання,  $\eta_i < 1$ .

Для написання магістерської роботи використовується персональний комп'ютер для якого розрахуємо витрати на електроенергію.

$$V_e = \frac{0,3 \cdot 160 \cdot 4,1 \cdot 0,5}{0,8} = 123$$

Накладні (загальновиробничі) витрати  $V_{нзв}$  охоплюють: витрати на управління організацією, оплата службових відряджень, витрати на утримання, ремонт та експлуатацію основних засобів, витрати на опалення, освітлення, водопостачання, охорону праці тощо. Накладні (загальновиробничі) витрати  $V_{нзв}$  можна прийняти як (100...150)% від суми основної заробітної плати розробників та робітників, які виконували дану МКНР, тобто:

$$V_{нзв} = (Z_o + Z_p) \cdot \frac{N_{нзв}}{100\%}, \quad (5.14)$$

де  $N_{нзв}$  – норма нарахування за статтею «Інші витрати».

$$V_{нзв} = 10226 \cdot \frac{100}{100\%} = 10226 \text{ грн}$$

Сума всіх попередніх статей витрат дає витрати, які безпосередньо стосуються даного розділу МКНР

$$B = 10226 + 1022 + 2474,5 + 326,7 + 5199 + 416,67 + 123 + 10226 \text{ грн} = 30013,87 \text{ грн.}$$

Прогнозування загальних втрат ЗВ на виконання та впровадження результатів виконаної МКНР здійснюється за формулою:

$$ЗВ = \frac{B}{\eta}, \quad (5.15)$$

де  $\eta$  – коефіцієнт, який характеризує стадію виконання даної НДР.

Оскільки, робота знаходиться на стадії науково-дослідних робіт, то коефіцієнт  $\beta = 0,9$ .

Звідси:

$$ЗВ = \frac{30013,87}{0,9} = 33\,348,7 \text{ грн.}$$

### 5.3 Розрахунок економічної ефективності науково-технічної розробки

У даному підрозділі кількісно спрогнозуємо, яку вигоду, зиск можна отримати у майбутньому від впровадження результатів виконаної наукової роботи. Розрахуємо збільшення чистого прибутку підприємства  $\Delta\Pi_i$ , для кожного із років, протягом яких очікується отримання позитивних результатів від впровадження розробки, за формулою

$$\Delta\Pi_i = \sum_1^n (\Delta\Pi_o \cdot N + \Pi_o \cdot \Delta N)_i \cdot \lambda \cdot \rho \cdot \left(1 - \frac{\nu}{100}\right) \quad (5.16)$$

де  $\Delta\Pi_o$  – покращення основного оціночного показника від впровадження результатів розробки у даному році.

$N$  – основний кількісний показник, який визначає діяльність підприємства у даному році до впровадження результатів наукової розробки;

$\Delta N$  – покращення основного кількісного показника діяльності підприємства від впровадження результатів розробки:

$\Pi_o$  – основний оціночний показник, який визначає діяльність підприємства у даному році після впровадження результатів наукової розробки;

$n$  – кількість років, протягом яких очікується отримання позитивних результатів від впровадження розробки:

$\lambda$  – коефіцієнт, який враховує сплату податку на додану вартість. Ставка податку на додану вартість дорівнює 20%, а коефіцієнт  $\lambda = 0,8333$ .

$\rho$  – коефіцієнт, який враховує рентабельність продукту.  $\rho = 0,25$ ;

$\nu$  – ставка податку на прибуток. У 2021 році – 18%.

Припустимо, що при впровадженні результатів наукової розробки покращується якість, що дозволяє підвищити ціну його реалізації на 1000 грн. Кількість одиниць реалізованої продукції також збільшиться: протягом першого року на 22 шт., протягом другого року – на 18 шт., протягом третього року на 12 шт. Реалізація продукції до впровадження розробки складала 150 шт., а її ціна

10000 грн. Розрахуємо прибуток, яке отримає підприємство протягом трьох років.

$$\begin{aligned}\Delta\Pi_1 &= [1000 \cdot 150 + (8000 + 1000) \cdot 22] \cdot 0,833 \cdot 0,25 \cdot \left(1 + \frac{18}{100}\right) \\ &= 85515 \text{ грн.}\end{aligned}$$

$$\begin{aligned}\Delta\Pi_2 &= [1000 \cdot 150 + (8000 + 1000) \cdot (22 + 18)] \cdot 0,833 \cdot 0,25 \cdot \left(1 + \frac{18}{100}\right) \\ &= 125324,85 \text{ грн.}\end{aligned}$$

$$\begin{aligned}\Delta\Pi_3 &= [1000 \cdot 150 + (8000 + 1000) \cdot (22 + 18 + 12)] \cdot 0,833 \cdot 0,25 \cdot \left(1 + \frac{18}{100}\right) \\ &= 151864,23 \text{ грн.}\end{aligned}$$

#### 4.4 Розрахунок ефективності вкладених інвестицій та періоду їх окупності

Розрахуємо основні показники, які визначають доцільність фінансування наукової розробки певним інвестором, є абсолютна і відносна ефективність вкладених інвестицій та термін їх окупності.

Розрахуємо величину початкових інвестицій  $PV$ , які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки.

$$PV = k_{\text{інв}} \cdot 3B, \quad (5.17)$$

$k_{\text{інв}}$  – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію. Це можуть бути витрати на підготовку приміщень, розробку технологій, навчання персоналу, маркетингові заходи тощо ( $k_{\text{інв}} = 2 \dots 5$ ).

$$PV = 2 \cdot 33\,348,7 = 66\,697,4$$

Розрахуємо абсолютну ефективність вкладених інвестицій  $E_{\text{абс}}$  згідно наступної формули:

$$E_{abc} = (ПП - PV) \quad (5.18)$$

де ПП – приведена вартість всіх чистих прибутків, що їх отримає підприємство від реалізації результатів наукової розробки, грн.;

$$ПП = \sum_1^T \frac{\Delta\Pi_i}{(1 + \tau)^t}, \quad (5.19)$$

де  $\Delta\Pi_i$  – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої НДДКР, грн.;

T – період часу, протягом якого виявляються результати впровадженої НДДКР, роки;

$\tau$  – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні; для України цей показник знаходиться на рівні 0,2;

t – період часу (в роках).

$$ПП = \frac{85515}{(1 + 0,2)^1} + \frac{125324,85}{(1 + 0,2)^2} + \frac{151864,23}{(1 + 0,2)^3} = 246\,178,03 \text{ грн.}$$

$$E_{abc} = (246\,178,03 - 66\,697,4) = 179\,480,63 \text{ грн.}$$

Оскільки  $E_{abc} > 0$  то вкладання коштів на виконання та впровадження результатів НДДКР може бути доцільним.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій  $E_s$ . Для цього користуються формулою:

$$E_s = T_{жс} \sqrt{1 + \frac{E_{abc}}{PV}} - 1, \quad (5.20)$$

$T_{жс}$  – життєвий цикл наукової розробки, роки.



$$E_B = \sqrt[3]{1 + \frac{246\,178,03}{66\,697,4}} - 1 = 0,67 = 67\%$$

Визначимо мінімальну ставку дисконтування, яка у загальному вигляді визначається за формулою:

$$\tau = d + f, \quad (5.21)$$

де  $d$  – середньозважена ставка за депозитними операціями в комерційних банках; в 2018 році в Україні  $d = (0,14 \dots 0,2)$ ;

$f$  – показник, що характеризує ризикованість вкладень; зазвичай, величина  $f = (0,05 \dots 0,1)$ .

$$\tau_{\min} = 0,18 + 0,05 = 0,23$$

Так як  $E_B > \tau_{\min}$  то інвестор може бути зацікавлений у фінансуванні даної наукової розробки.

Розрахуємо термін окупності вкладених у реалізацію наукового проекту інвестицій за формулою:

$$T_{ок} = \frac{1}{E_B} \quad (5.22)$$

$$T_{ок} = \frac{1}{0,67} = 1,5 \text{ роки}$$

Так як  $T_{ок} \leq 3 \dots 5$ -ти років, то фінансування даної наукової розробки в принципі є доцільним.

#### 4.5 Висновки до економічного розділу

Було проведено оцінку комерційного потенціалу розробки програмного забезпечення у середовищі Kubernetes для інтегрування програмних продуктів, яка є на вище середньому рівні. При порівнянні нової розробки з аналогом виявлено, що вона є якіснішою і конкурентоспроможнішою відносно аналога, а також краще по технічним і економічним показникам.

Прогнозування витрат на виконання науково-дослідної роботи по кожній з статей витрат складе 30013,87 грн. Загальна ж величина витрат на виконання та впровадження результатів даної НДР буде складати 33348,7 грн.

Вкладені інвестиції в даний проект окупляться через 18 місяців при прогнозованому прибутку 246 178,03 грн. за три роки.

## ВИСНОВКИ

Під час виконання магістерської дипломної роботи було метод та програмного забезпечення у середовищі Kubernetes для інтегрування програмних продуктів.

Було проаналізовано стан даної проблеми на сьогоднішній день. Розглянуто основні аналоги, визначено їх особливості та недоліки і розроблено порівняння з власною програмною системою.

Для серверної частини було обрано мову програмування Nodejs. Для агенту встановленого в Kubernetes кластер користувача було обрано мову Go.

Були вирішенні наступні задачі:

- провести аналіз існуючих рішень і засобів з автоматизації інтеграції та розгортання програмного коду;
- розробити розгалужену архітектуру програмної системи;
- розробити алгоритм взаємодії сервера й Kubernetes кластера;
- розробити алгоритм виконання пайплайну в середовищі Kubernetes;
- розробити алгоритм збереження вихідних даних кроків;
- розробити інтерфейс програмного продукту;
- розробити систему інтеграції програмного продукту в режимі реально часу.

Було розроблено архітектуру програмної системи та модель виконання пайплайну у приватному кластері користувача, які дозволили запобігти поширенню конфіденційних даних з сервером, у випадку коли пайплайн повинен мати до них доступ. Це було реалізовано за допомогою агента, який повинен бути встановленим у кластері користувача.

Тестування програмної системи довело повну її працездатність та відповідність поставленому технічному завданню. Розроблено інструкцію користувача. Магістерську кваліфікаційну роботу було оформлено відповідно до вимог [23].

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Belmont J. M. Hands-On Continuous Integration and Delivery/J. M. Belmont – New York: Packt, 2018, 416с.
2. CI/CD Pipeline. [Електронний ресурс] – Режим доступу: <https://semaphoreci.com/blog/cicd-pipeline>
3. Kubernetes. [Електронний ресурс] – Режим доступу: <https://kubernetes.io/>
4. Kubernetes wiki. [Електронний ресурс] – Режим доступу: <https://en.wikipedia.org/wiki/Kubernetes>
5. Hudson (Software). [Електронний ресурс] – Режим доступу: [https://en.wikipedia.org/wiki/Hudson\\_\(software\)](https://en.wikipedia.org/wiki/Hudson_(software))
6. Jenkins [Електронний ресурс] – Режим доступу: <https://www.jenkins.io/>
7. An Introduction to Kubernetes [Електронний ресурс] – Режим доступу: <https://github.com/Leverege/kubernetes-book/blob/master/An%20Introduction%20to%20Kubernetes%20%5BFeb%202019%5D.pdf>
8. Страндартні потоки [Електронний ресурс] – Режим доступу: [https://uk.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D1%96\\_%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D0%B8](https://uk.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D1%96_%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D0%B8)
9. Daniel P. Understanding the Linux Kernel, Second Edition/P. Daniel, C. Marco – California: O'Reilly Media, 2002. – 950 с.
10. Загальна характеристика інформаційного забезпечення [Електронний ресурс] – Режим доступу: <https://library.if.ua/book/80/5658.html>
11. Купер А. Интерфейс. Основы проектирования взаимодействия. 4-е изд./А. Купер – Санкт-Петербург: Питер, 2016. – 534 с.

12. Алгоритм [Електронний ресурс] – Режим доступу: <https://uk.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC>
13. Графічний інтерфейс користувача [Електронний ресурс] – Режим доступу: [https://en.wikipedia.org/wiki/Graphical\\_user\\_interface](https://en.wikipedia.org/wiki/Graphical_user_interface)
14. Подійно-орієнтоване програмування [Електронний ресурс] – Режим доступу: [https://en.wikipedia.org/wiki/Event-driven\\_programming](https://en.wikipedia.org/wiki/Event-driven_programming)
15. Event based VS Thread based server sys. [Електронний ресурс] – Режим доступу: <https://stackoverflow.com/questions/25280207/what-are-the-differences-between-event-driven-and-thread-based-server-system>
16. Node.js [Електронний ресурс] – <https://ru.wikipedia.org/wiki/Node.js>
17. libsigc++ [Електронний ресурс] – <https://developer.gnome.org/libsigc++/stable/>
18. Go programming language [Електронний ресурс] – [https://en.wikipedia.org/wiki/Go\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Go_(programming_language))
19. MongoDB [Електронний ресурс] – <https://github.com/night-codes/The-Little-MongoDB-Book-rus>
20. Калбертсон Р. Быстрое тестирование. / Р. Калбертсон, К. Браун, Г. Кобб. – М.: «Вильямс», 2002. – 374 с. – ISBN 5-8459-0336-X.
21. Software testing [Електронний ресурс] – [https://en.wikipedia.org/wiki/Software\\_testing](https://en.wikipedia.org/wiki/Software_testing)
22. Методичні вказівки до виконання економічної частини магістерських кваліфікаційних робіт / Уклад. : В. О. Козловський, О. Й. Лесько, В. В. Кавецький. – Вінниця : ВНТУ, 2021. – 42 с.
23. О. Н. Романюк, Р. Р. Обертюх, Т. О. Савчук, Л. П. Громова. Положення про кваліфікаційну роботу у Вінницькому національному технічному університеті. Вінниця, 2015. 27 с.

## ДОДАТКИ

Додаток А – Технічне завдання  
Міністерство освіти і науки України  
Вінницький національний технічний університет  
Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ  
д.т.н., проф. О. Н. Романюк  
" \_\_\_\_ " \_\_\_\_\_ 2021 р.

**Технічне завдання**  
**на магістерську кваліфікаційну роботу «Розробка методу та**  
**програмного забезпечення у середовищі Kubernetes для інтегрування**  
**програмних продуктів» за спеціальністю**  
**121 – Інженерія програмного забезпечення**

Керівник магістерської кваліфікаційної роботи:

\_\_\_\_\_ к.т.н., доц. Д.І.Кательніков  
" \_\_\_\_ " \_\_\_\_\_ 2021 р.

Виконав:

\_\_\_\_\_ студент гр.1ПІ-20м Д.О. Мельник  
" \_\_\_\_ " \_\_\_\_\_ 2021 р.

Вінниця – 2021 року

## **1. Найменування та галузь застосування**

Магістерська кваліфікаційна робота: «Розробка методу та програмного забезпечення у середовищі Kubernetes для інтегрування програмних продуктів».

Галузь застосування – автоматизація процесу інтеграції та розгортання.

## **2. Підстава для розробки.**

Підставою для виконання магістерської кваліфікаційної роботи (МКР) є індивідуальне завдання на МКР та наказ № ректора по ВНТУ про закріплення тем МКР.

## **3. Мета та призначення розробки.**

Метою роботи є підвищення безпеки в роботі CI/CD рішень за рахунок використання Kubernetes як середовища виконання пайплайнів..

Призначення роботи – розробка методу та програмного забезпечення у середовищі Kubernetes для підвищення безпеки інтегрування програмних продуктів.

## **3 Вихідні дані для проведення НДР**

Перелік основних літературних джерел, на основі яких буде виконуватись МКР.

1. Belmont J. M. Hands-On Continuous Integration and Delivery/J. M. Belmont – New York: Packt, 2018, 416с.
2. Daniel P. Understanding the Linux Kernel, Second Edition/P. Daniel, C. Marco – California: O'Reilly Media, 2002. – 950 с.
3. Купер А. Интерфейс. Основы проектирования взаимодействия. 4-е изд./А. Купер – Санкт-Петербург:Питер, 2016. – 534 с.

## **4. Технічні вимоги**

Можливість описувати пайплайн за допомогою yaml файлу. Виконання пайплайну відповідно до опису. Вихідні дані виконання пайплайну – результат



про виконання кроків; алгоритм виконання пайплайну у кластері користувача; алгоритм безпечної взаємодії серверу та агента; алгоритм логування вихідних даних контейнерів; запуск сервера на основі таких вхідних даних: игі бази даних MongoDB; вихідні дані – програмне забезпечення у середовищі Kubernetes для інтегрування програмних продуктів.

### **5. Конструктивні вимоги.**

Конструкція пристрою повинна відповідати естетичним та ергономічним вимогам, повинна бути зручною в обслуговуванні та керуванні.

Графічна та текстова документація повинна відповідати діючим стандартам України.

### **6. Перелік технічної документації, що пред'являється по закінченню робіт:**

- пояснювальна записка до МКР;
- технічне завдання;
- лістинги програми.

### **7. Вимоги до рівня уніфікації та стандартизації**

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

### **8. Стадії та етапи розробки:**

№ з/п	Назва етапів магістерської кваліфікаційної Роботи	Строк виконання етапів роботи
1	Аналіз існуючих рішень та виявлення їх недоліків	
2	Розробка розподіленої архітектури програмного забезпечення	
3	Розробка алгоритму безпечної взаємодії агенту та серверу	
4	Розробка алгоритму виконання пайплайнів у середовищі Kubernetes	

5	Економічна частина	
---	--------------------	--

### **9. Порядок контролю та прийняття.**

Виконання етапів магістерської кваліфікаційної роботи контролюється керівником згідно з графіком виконання роботи. Прийняття магістерської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.

Додаток Б – Протокол перевірки

## ПРОТОКОЛ ПЕРЕВІРКИ НАВЧАЛЬНОЇ (КВАЛІФІКАЦІЙНОЇ) РОБОТИ

Назва роботи: **Розробка методу та програмного забезпечення у середовищі Kubernetes для інтегрування програмних продуктів.**

Тип роботи: кваліфікаційна робота

Підрозділ : кафедра програмного забезпечення, ФІТКІ, ІПІ – 20м

Науковий керівник: к.т.н. доц. Кательніков Д.І.

Unicheck	
<b>Оригінальність</b>	<b>89,5%</b>
Схожість	10,5%

### Аналіз звіту подібності

■ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**

Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Заявляю, що ознайомена з повним звітом подібності, який був згенерований Системою щодо роботи «Розробка методу та програмного забезпечення у середовищі Kubernetes для інтегрування програмних продуктів».

Автор \_\_\_\_\_

Мельник Денис Олександрович

Опис прийнятого рішення: **допустити до захисту**

Особа, відповідальна за перевірку  
(підпис) (прізвище, ініціали)

\_\_\_\_\_ Черноволик Г. О.

Експерт  
(за потреби) \_\_\_\_\_  
(підпис)

\_\_\_\_\_  
(прізвище, ініціали, посада)

## Додаток В – Лістинг коду

## PipelineRunner.js

```

const { readFileSync } = require('fs');
const { execSync } = require('child_process');
const _ = require('lodash');
const yaml = require('js-yaml');
const uuid = require('uuid/v4');
const { docker, containerTimeout } = require('../config');
const startListener = require('./startListener');
const createStorageLayer = require('./storageLayers');
const { normalizeId, logContainer } = require('../helpers');
const Logger = require('./ContainerLogger');

const STATIC_STEP_LABEL = 'codefresh-assessment-pipeline-step';

class PipelineRunner {
  constructor(settings = {}) {
    const { path, pipeline, storageLayer } = settings;
    this.path = path || './pipeline.yml';
    this.pipeline = pipeline;
    this.storageLayer = storageLayer;

    // to prevent step name conflicts
    this.pipelineSalt = uuid();
    // label for startListener to listen for
    this.stepLabel = `${STATIC_STEP_LABEL}-${this.pipelineSalt}`;
  }

  async run() {
    const executionResult = [];

    try {
      const pipeline = this._loadPipeline();
      this._validatePipeline(pipeline);

      await this._initListener();

      for (const [stepName, stepMeta] of Object.entries(pipeline.steps)) {

        const container = await this._runStep({ name: stepName, ...stepMeta });
        normalizeId(container);
        executionResult.push({ name: stepName, ...container });

        if (container.exitCode > 0) {
          if (container.exitCode === 143) {
            throw new Error(`Step ${stepName} was terminated cause of predefined max execution timeout:
${containerTimeout / 1000} sec`);
          }
          throw new Error(`Pipeline failed on step ${stepName} with status code ${container.statusCode}`);
        }
      }
      console.log('Pipeline ran successfully');
      this._gracefulShutdown();
      return { executionResult, status: 'Pipeline ran successfully' };
    } catch (err) {
      console.error('Failed to run pipeline cause of: ', err);
      this._gracefulShutdown();
      return { executionResult, status: `Failed to run pipeline cause of: ${err}`, error: true };
    }
  }
}

```

```

async _runStep(step) {
  const { image: Image, cmd: Cmd, name } = step;

  try {
    const container = await docker.createContainer({
      name: `${this.pipelineSalt}-${name}`,
      Image,
      Cmd,
      Labels: {
        meta: this.stepLabel
      }
    });
    const output = await container.start();
    let running = true;

    if (output.Error) {
      throw output.Error;
    }

    setTimeout(() => {
      if (running) {
        container.stop().catch((err) => {
          console.error('Container timeout termination failed', err);
        });
      }
    }, containerTimeout);

    const res = await container.wait();
    running = false;

    return {
      exitCode: res.StatusCode,
      id: container.id
    };
  } catch (err) {
    throw new Error(`Failed to run step cause of: ${err}`);
  }
}

_loadPipeline() {
  try {
    if (this.pipeline) {
      return this.pipeline;
    }
    const file = readFileSync(this.path);
    return yaml.safeLoad(file);
  } catch (err) {
    throw new Error(`Failed to load pipeline cause of: ${err}`);
  }
}

_validatePipeline({ steps }) {
  if (!_.keys(steps)) {
    throw new Error('Validation failed');
  }
}

_initListener() {
  return startListener.start({ labels: [ `meta=${this.stepLabel}` ] }, async (container) => {
    normalizeId(container);
    logContainer('Attaching to container', container);
  });
}

```

```

    const storage = createStorageLayer(this.storageLayer, 'writer', container);

    const logger = new Logger(container.id, storage);
    await logger.run();
  });
}

_gracefulShutdown() {
  execSync('yes | docker container prune');
  startListener.stop();
}
}

module.exports = PipelineRunner;

```

## StartListener.js

```

const _ = require('lodash');
const { docker } = require('.././config');

class StartListener {
  start(selector, containerHandler) {
    return new Promise((resolve, reject) => {
      const filters = this._getApiFilter(selector);
      docker.getEvents({ filters }, (err, events) => {
        if (err) {
          reject(err);
        } else {
          console.log('Listening for starting containers...');
          this.events = events;
          events.on('data', (chunk) => {
            const containerData = JSON.parse(chunk.toString());
            if (this._matchLocalFilters(containerData, selector)) {
              containerHandler(containerData)
            }
          });
          resolve();
        }
      });
    });
  }

  stop() {
    if (this.events) {
      this.events.destroy();
      this.events = null;
      console.log('No longer listen to docker `start` events');
    } else {
      console.log('StartListener already stopped');
    }
  }

  _matchLocalFilters(eventData, selector) {
    const name = _.get(eventData, 'Actor.Attributes.name');
    if (selector.nameRegex && name) {
      return (new RegExp(selector.nameRegex)).test(name);
    }
    return true;
  }
}

```

```

_getApiFilter(selector) {
  const filter = {};
  _set(filter, 'event', ['start']);

  if (selector.labels) {
    _set(filter, 'label', _castArray(selector.labels));
  }

  return JSON.stringify(filter);
}
}

module.exports = new StartListener();

```

## ContainerLogger.js

```

const { promisify } = require('util');
const { logContainer } = require('../helpers');
const { docker } = require('.././config');

class ContainerLogger {
  constructor(id, storage) {
    this.id = id;
    this.storage = storage;
    this.container = docker.getContainer(this.id);
  }

  async run() {
    try {
      await this._initStorageLayer();
      await this._exec();
    } catch (err) {
      console.error('Failed to run ContainerLogger', err);
    }
  }

  _initStorageLayer() {
    return this.storage.init(this.id);
  }

  async _exec() {
    const stream = await this._attachToContainer();

    stream.on('end', () => {
      this.storage.stream.end();
      logContainer('Container finished execution', { id: this.container.id });
    });

    this.container.modem.demuxStream(stream, this.storage.stream, this.storage.stream);
  }

  _attachToContainer() {
    const attach = promisify(this.container.attach.bind(this.container));
    return attach({stream: true, stdout: true, stderr: true, logs: true});
  }
}

module.exports = ContainerLogger;

```

## App.js

```

const http = require('http');
const express = require('express');
const bodyParser = require('body-parser');
const logger = require('morgan');
const mongoose = require('mongoose');
const chalk = require('chalk');

const { mongo } = require('./config');
const index = require('./api/index');

const app = express();
const server = http.createServer(app);

/**
 * Connect to MongoDB.
 */
mongoose.set('useFindAndModify', false);
mongoose.set('useCreateIndex', true);
mongoose.set('useNewUrlParser', true);
mongoose.set('useUnifiedTopology', true);
mongoose.connect(mongo.uri)
  .then(() => console.log(`CONNECTED TO ${mongo.uri}`))
  .catch(err => {
    console.error(err);
    console.log('%s MongoDB connection error. Please make sure MongoDB is running.', chalk.red('X'));
    process.exit();
  });

/**
 * Express configuration.
 */
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*"); // update to match the domain you will make the request from
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
  next();
});

app.use('/api', index);

app.use(function(req, res, next) {
  const err = new Error('Not Found');
  err.status = 404;
  next(err);
});

app.use(function(err, req, res, next) {
  res.status(err.status || 500);
  res.json({
    message: err.message,
    error: err
  });
});

server.listen(process.env.PORT || 8080);

module.exports = app;

```



## FS.reader.js

```

const { existsSync, readFileSync } = require('fs');
const BaseStorageLayer = require('../Base');
const { fsLayerPath, notFoundMessage } = require('../constants');

class FSStorageRead extends BaseStorageLayer.Reader {
  async read() {
    const logPath = `${fsLayerPath}/${this.container.id}.log`;
    if (existsSync(logPath)) {
      return readFileSync(logPath).toString();
    } else {
      throw new Error(notFoundMessage);
    }
  }
}

module.exports = FSStorageRead;

```

## FS.writer.js

```

const fs = require('fs');

const BaseStorageLayer = require('../Base');
const { delay } = require('../helpers');
const { fsLayerPath } = require('../constants');

class FSStorageWrite extends BaseStorageLayer.Writer {
  write(log) {
    if (!this.writeStream) {
      const writePath = `${fsLayerPath}/${this.container.id}.log`;
      if (!fs.existsSync(fsLayerPath)) {
        fs.mkdirSync(fsLayerPath, { recursive: true });
      }
      this.writeStream = fs.createWriteStream(writePath);
    }

    this.writeStream.write(log);
  }
}

module.exports = FSStorageWrite;

```

## Mongo.reader.js

```

const mongoose = require('mongoose');

const BaseStorageLayer = require('../Base');
const ContainerLog = require('./Log');
const { mongo } = require('../././config');
const { notFoundMessage } = require('../constants');

class MongoStorageRead extends BaseStorageLayer.Reader {
  async init() {
    if (mongoose.connection.readyState === 0) {
      return this._initMongo();
    }
  }

  async read() {

```

```

    const { id } = this.container;
    const res = await ContainerLog.findById(id);
    if (res) {
      return res.data;
    } else {
      throw new Error(notFoundMessage);
    }
  }
}

_initMongo() {
  return mongoose.connect(mongo.uri, { useNewUrlParser: true, useUnifiedTopology: true });
}
}

module.exports = MongoStorageRead;

```

## Mongo.writer.js

```

const mongoose = require('mongoose');

const BaseStorageLayer = require('../Base');
const ContainerLog = require('./Log');
const { mongo } = require('../././config');
const { getImageAndName } = require('../././helpers');

class MongoStorageWrite extends BaseStorageLayer.Writer {
  onFinish() {
    this.containerLog.save().then(() => console.log('Saved successfully!')).catch(console.error);
  }

  async init() {
    this.containerLog = new ContainerLog({
      _id: this.container.id,
      data: "",
      ...getImageAndName(this.container)
    });

    if (mongoose.connection.readyState === 0) {
      return this._initMongo();
    }
  }

  write(log) {
    this.containerLog.data += log;
  }

  _initMongo() {
    return mongoose.connect(mongo.uri, { useNewUrlParser: true, useUnifiedTopology: true });
  }
}

module.exports = MongoStorageWrite;

```

## Pipelines.jsx

```

import React, { useContext, useCallback } from 'react';
import MonacoEditor from 'react-monaco-editor';
import BaseLayout from '../BaseLayout';

```

```

import { Store } from '../App.store';
import './style.scss';

const Pipeline = () => {
  const store = useContext(Store);
  const { editor, response, error, loading, storageLayer } = store.pipeline;

  const handlerStorageChange = useCallback(e => {
    storageLayer.set(e.target.value);
  }, [storageLayer]);

  const runPipeline = useCallback(() => {
    loading.set(true);

    const base64Pipeline = window.btoa(editor.value);

    fetch('http://localhost:8080/api/pipelines/run', {
      body: JSON.stringify({
        pipeline: base64Pipeline,
        storageLayer: storageLayer.value
      }),
      headers: {
        'Content-Type': 'application/json',
      },
      method: 'POST'
    })
    .then(res => res.json())
    .then(response.set)
    .catch(error.set)
    .finally(() => {
      loading.set(false);
    })
  }, [editor, storageLayer]);

  return (
    <BaseLayout title="Pipeline" className="pipeline-page">
      <div className="controls">
        <select value={storageLayer.value} onChange={handlerStorageChange}>
          <option value="fs">File system</option>
          <option value="mongo">Mongo</option>
        </select>

        <button onClick={runPipeline}>Run pipeline</button>
      </div>
      <div className="wrapper">
        <div style={{ width: '50%' }}>
          <p className="solid-label">Pipeline yaml:</p>
          <MonacoEditor
            width="100%"
            height="500"
            language="yaml"
            theme="vs-dark"
            value={editor.value}
            onChange={editor.set}
          />
        </div>
        <div>
          <p className="solid-label">Response:</p>
          <div className="result response">
            {loading.value
              ? 'Running...'
              : error.value
            } <div className="error">

```

```

        {error.value.toString()}
      </div>
    : <div>
      {typeof response.value === 'string' ? response.value : JSON.stringify(response.value, null, 2)}
    </div>
  }
</div>
</div>
</div>
</BaseLayout>
)
}

export default Pipeline;

```

## Logs.jsx

```

import React, { useCallback, useContext } from 'react'
import BaseLayout from '../BaseLayout'
import { Store } from '../App.store';
import './style.scss'

const Logs = () => {
  const store = useContext(Store)
  const { container, error, loading, response, storageLayer } = store.logs;

  const handlerChange = useCallback(e => {
    container.set(e.target.value);
  }, [container]);
  const handlerStorageChange = useCallback(e => {
    storageLayer.set(e.target.value);
  }, [storageLayer]);

  const handleGetLogs = useCallback(() => {
    loading.set(true);
    error.set(null);
    response.set("");
    fetch(`http://localhost:8080/api/containers/${container.value}/logs?storageLayer=${storageLayer.value}`)
      .then(res => res.json())
      .then(response.set)
      .catch((err) => {
        console.error(err);
        error.set(err);
      })
      .finally(() => loading.set(false));
  }, [container, storageLayer]);

  return (
    <BaseLayout title="Logs" className="logs-page">
      <div className="controls">
        <div>
          <label htmlFor="logs_select">Storage Layer</label>
          <select id="logs_select" value={storageLayer.value} onChange={handlerStorageChange}>
            <option value="fs">File system</option>
            <option value="mongo">Mongo</option>
          </select>
        </div>
        <div>
          <label htmlFor="logs_input">Container ID</label>
          <input id="logs_input" type="text" value={container.value} onChange={handlerChange}/>
        </div>
        <button onClick={handleGetLogs}>{loading.value ? 'Loading...' : 'Get logs'}</button>
      </div>
    </BaseLayout>
  );
};

```

```

    </div>
    <p className="solid-label">Response:</p>
    <div className="response">
      {error.value ? <div className="error">
        {error.value.toString()}
      </div> : <div>
        {typeof response.value === 'string' ? response.value : JSON.stringify(response.value, null, 2)}
      </div>}
    </div>
  </BaseLayout>
)
}

export default Logs;

```

## BaseLayout.jsx

```

import React from 'react'

const BaseLayout = ({ title, children, className }) => {
  return (
    <div className={className}>
      <h1>{title}</h1>
      <hr/>
      <section>
        {children}
      </section>
    </div>
  )
}

export default BaseLayout;

```



Додаток Г – Ілюстративна частина

## **ІЛЮСТРАТИВНА ЧАСТИНА**

**РОЗРОБКА МЕТОДУ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ  
У СЕРЕДОВИЩІ KUBERNETES ДЛЯ ІНТЕГРУВАННЯ  
ПРОГРАМНИХ ПРОДУКТІВ\**

## ВІННИЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Магістерська кваліфікаційна робота на тему:  
«Розробка методу та програмного забезпечення у середовищі  
Kubernetes для інтегрування програмних продуктів»

Виконав ст. гр. 1ПІ-20м

Мельник Д.О.

Науковий керівник: к.т.н, доцент

Кательніков Д. І.

Рисунок Г.1 – Титульна сторінка

## Актуальність

На сьогоднішній день CI / CD є найбільш широко використовуваною практикою в розробці та розгортанні програмного забезпечення. Kubernetes - сучасна технологія яка може бути адаптованою під середовище виконання для процесу CI/CD.

Це має ряд плюсів:

- Використання декларативної конфігурації та автоматизації.
- Розширюваність.
- Гнучке управління доступом.
- Стійкість до відмов.

Рисунок Г.2 – Актуальність роботи



## Мета, предмет та об'єкт дослідження

**Мета** – підвищення безпеки в роботі CI/CD рішень за рахунок використання Kubernetes як середовища виконання пайплайнів.

**Об'єкт** – захизений процес CI/CD в рамках середовища виконання Kubernetes.

**Предмет** – методи та засоби виконання CI/CD процесів у середовищі Kubernetes.

Рисунок Г.3 – Мета предмет та об'єкт дослідження

## Наукова новизна

Удосконалено метод виконання процесів безперервної інтеграції та розгортання за допомогою середовища виконання Kubernetes, що дає змогу користувачу з власним Kubernetes кластером використовувати його у якості середовища виконання, навіть якщо цей кластер знаходиться у приватній мережі, на відміну від аналогів які мають прямий доступ до середовища виконання.

Це реалізується за допомогою агента, який встановлюється у кластер користувача, особливість якого полягає у підтримці з'єднання з CI/CD платформою шляхом постійного опитування на предмет наявності нових «задач», які повинні бути виконані у кластері користувача. Це дозволяє не розголошувати секрети платформі під час виконання пайплайну, а зберігати їх у власному кластері, що значно зменшує ризики їх несанкціонованого поширення.

Рисунок Г.4 – Наукова новизна

## Огляд програм-аналогів

Для того щоб оцінити існуючі аналоги потрібно спочатку визначити критерії при оцінюванні переваг та недоліків даних програмних систем, такі як:

- Швидкість виконання процесу інтеграції
- Екосистема плагінів
- Безпека конфіденційних даних
- Вибір місця збереження зчитаних вихідних даних контейнерів під час виконання процесу інтеграції
- Можливість роботи з Docker контейнерами
- Зрозумілий формат YAML файлу

Рисунок Г.5 – Огляд аналогів

## Аналог: Hudson

Це система для реалізації безперервної інтеграцією, написана на Java, яка працює на сервлет контейнерах, таких як Apache Tomcat або Glassfish. Hudson може бути розширений за допомогою архітектури плагінів.

Основними недоліками програмної системи є відсутність підтримки Docker контейнерів.

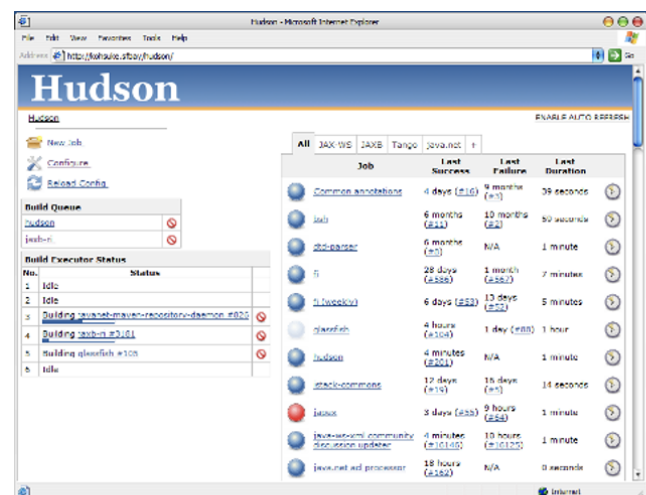


Рисунок Г.6 – Аналог Hudson

## Аналог: Jenkins

Це сервер автоматизації з відкритим кодом з неперевершеною екосистемою плагінів, який підтримує практично кожен інструмент як частину пайплайнів.

Вихідний код здебільшого Java, з кількома файлами Groovy, Ruby та Antlr.

Серед недоліків можна виділити проблеми з повідомленнями про помилки, важке початкове налаштування, незручний інтерфейс, проблеми при підтриманні великої кількості одночасно активних процесів.

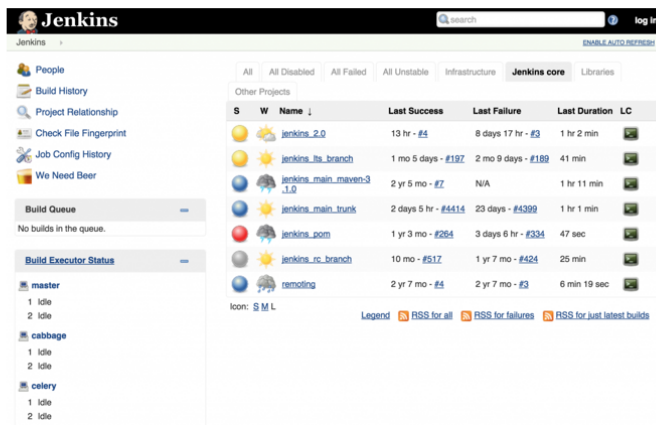


Рисунок Г.7 – Аналог Jenkins

## Архітектура додатку

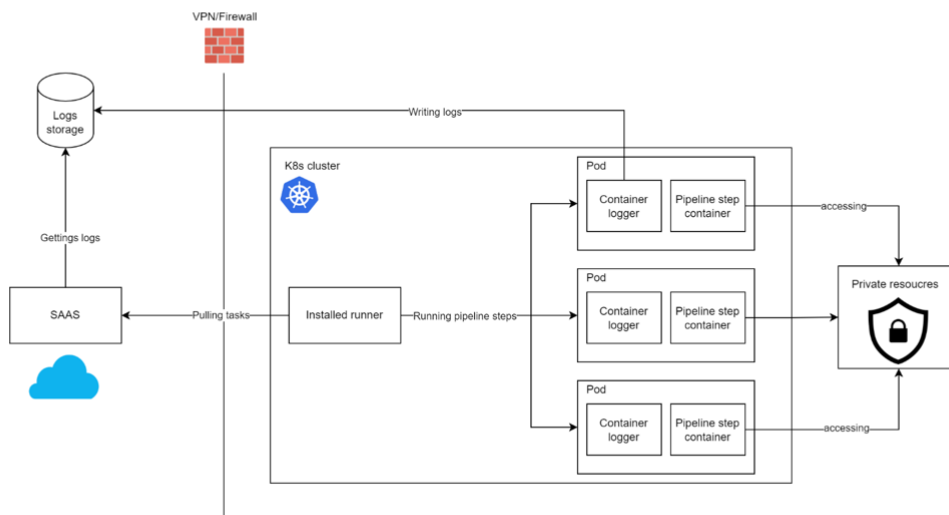


Рисунок Г.8 – Архітектура додатку

### Алгоритм взаємодії агента з сервером



Рисунок Г.9 – Блок схема 1

### Алгоритм взаємодії агента з сервером

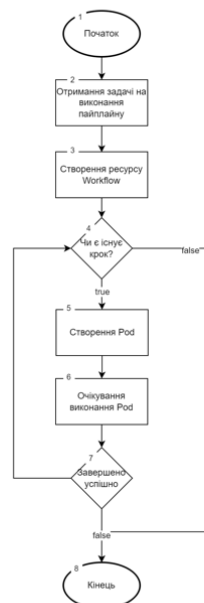


Рисунок Г.10 – Блок схема 2



Рисунок Г.11 – Блок схема 3

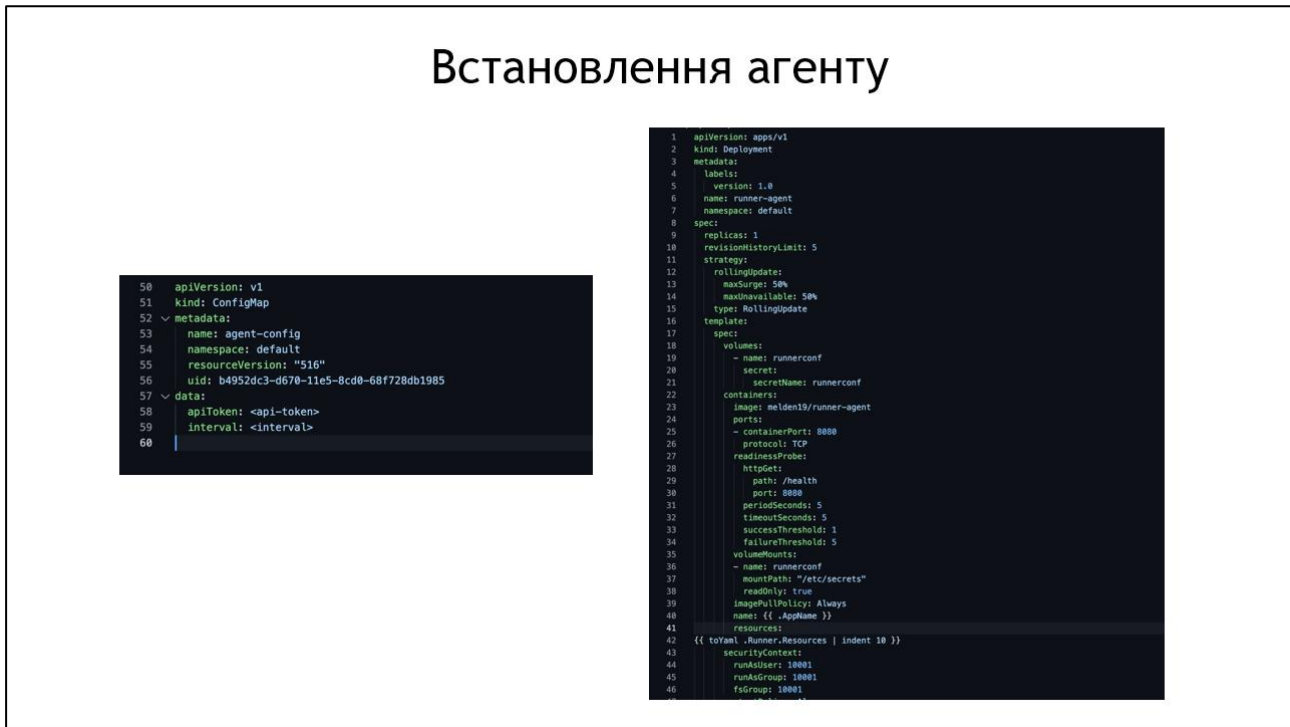


Рисунок Г.12 – Маніфести для встановлення агенту

## Результат роботи програмної системи

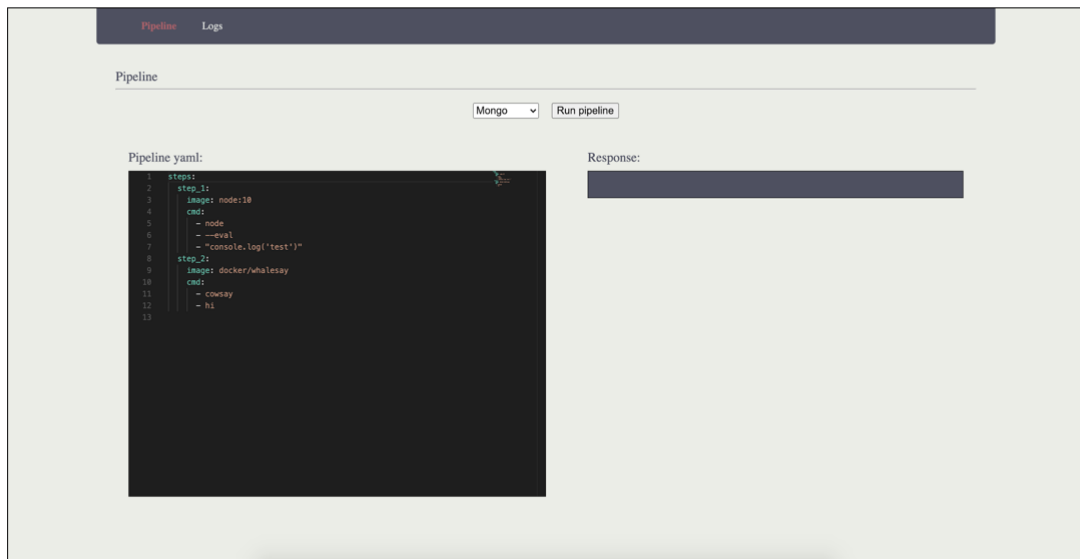


Рисунок Г.13 – Запуск пайплайну

## Результат роботи програмної системи

```
~ > k get workflow
NAME          STATUS    AGE
pipeline-1    Running  2s
~ >

~ > k get pods | grep pipeline-1
pipeline-1-3893458983    0/2    Completed    0    4m50s
pipeline-1-809788873    0/2    Completed    0    5m
~ >
```

Рисунок Г.14 – Моніторинг виконання пайплайну

## Результат роботи програмної системи

```

1  apiVersion: v1alpha1
2  kind: Workflow
3  metadata:
4    name: pipeline-1
5  spec:
6    entrypoint: steps
7    templates:
8      - name: steps
9        steps:
10         - name: step-1
11           template: step-1
12         - name: step-2
13           template: step-2
14       - name: step-1
15         container:
16           image: node:10
17           command: [node]
18           args: ["--eval", "console.log('test')"]
19           resources:
20             limits:
21               memory: 32Mi
22               cpu: 100m
23       - name: step-2
24         container:
25           image: docker/whalesay
26           command: [cowsay]
27           args: ["hello world"]
28           resources:
29             limits:
30               memory: 32Mi
31               cpu: 100m
32

```

Рисунок Г.15 – Створений ресурс Workflow

## Результат роботи програмної системи

The screenshot shows a web interface for managing a pipeline. At the top, there are tabs for "Pipeline" and "Logs". Below the tabs, there is a "Pipeline" section with a dropdown menu set to "Mongo" and a "Run pipeline" button. The main content area is divided into two panels: "Pipeline yaml:" and "Response:".

The "Pipeline yaml:" panel displays the following configuration:

```

1 steps:
2   step_1:
3     image: node:10
4     cmd:
5       - node
6       - --eval
7       - "console.log('test')"
8   step_2:
9     image: docker/whalesay
10    cmd:
11      - cowsay
12      - hi
13

```

The "Response:" panel displays the following JSON output:

```

{
  "executionResult": [
    {
      "name": "step_1",
      "exitCode": 0,
      "id": "84916d2a-9511"
    },
    {
      "name": "step_2",
      "exitCode": 1,
      "id": "662955c29994"
    }
  ],
  "status": "Pipeline ran successfully"
}

```

Рисунок Г.16 – Результат виконання пайплайну

