НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2023 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

на тему: Навчальна система для сумісного виконання IT проєктів на основі методології CI/CD

Виконав : студент __4__ курсу, групи _ІО-92_
(шифр групи)

_____Костюк Антон Володимирович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____асистент кафедри ОТ, Гайдай А. Р._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) ст. викладач Виноградов Ю.М._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент ___ професор кафедри ІСТ, д.т.н., Богдан Корнієнко_____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2023 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“ Комп’ютерні системи та мережі”

спеціальності 123 “Комп’ютерна інженерія”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2023 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Костюка Антона Володимировича

1. Тема проєкту *Навчальна система для сумісного виконання IT проєктів на основі методології CI/CD*
керівник проєкту Гайдай Анатолій Русланович, асистент кафедри,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 31 травня 2023 року №2101-с

2. Термін здачі студентом закінченого проєкту *8 червня 2023 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Аналіз предметної області та рішень.

Розділ 2. Аналіз та вибір засобів реалізації.

Розділ 3. Проєктування та реалізація системи.

Розділ 4. Тестування та огляд системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема, функціональна схема, принципова схема.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Виноградов Ю.М.		

7. Дата видачі завдання «22» листопада 2022 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*18.12.2022-30.12.2022*	
2.	*Вивчення та аналіз завдання*	*31.12.2022-12.03.2023*	
3.	*Розробка архітектури та загальної структури системи*	*13.03.2023-26.03.2023*	
4.	*Розробка структур окремих підсистем*	*27.03.2023-02.04.2023*	
5.	*Програмна реалізація системи*	*03.04.2023-16.04.2023*	
6.	*Оформлення пояснювальної записки*	*24.04.2023-21.05.2023*	
7.	*Захист програмного продукту*	*06.06.2023*	
8.	*Передзахист*	*09.06.2023*	
9.	*Захист*	*19.06.2023*	

Студент-дипломник _____ Антон КОСТЮК
(підпис)

Керівник проєкту _____ Анатолій ГАЙДАЙ
(підпис)

АНОТАЦІЯ

Даний дипломний проект присвячений розробці навчальної системи на основі методології CI/CD. Було проаналізовано існуючі рішення, що мають подібний функціонал, враховано їх переваги та недоліки. Обрано відповідні технології для розробки системи. Робота включає етапи аналізу, проєктування, розробки і тестування системи.

Систему реалізовано у вигляді CI/CD пайплайну, використовуючи сервер автоматизації Jenkins, систему статичного аналізу коду SonarQube, базу даних PostgreSQL та сервер системи контролю версій Gitea. Створена система націлена на навчання основним практикам та підходам DevOps та CI/CD, покращення взаємодії в командах.

Ключові слова: методологія CI/CD, навчальна система, Jenkins, SonarQube, Gitea, PostgreSQL.

ANNOTATION

This Bachelor's Degree project is dedicated to the development of a training system based on the CI/CD methodology. Existing solutions with similar functionality were analyzed, their advantages and disadvantages were taken into account. Appropriate technologies were chosen for the development of the system. The work includes the stages of analysis, design, development and testing of the system.

The system is implemented as a CI/CD pipeline developed using Jenkins automation server, SonarQube static code analysis system, PostgreSQL database and Gitea version control system server. The created system is aimed at teaching basic practices and approaches to DevOps and CI/CD, improving interaction in teams.

Keywords: CI/CD methodology, traning system, Jenkins, SonarQube, Gitea, PostgreSQL.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Навчальна система для	4		
			сумісного виконання ІТ			
			проєктів на основі			
			методології CI/CD			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Навчальна система для	62		
			сумісного виконання ІТ			
			проєктів на основі			
			методології CI/CD			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Навчальна система для	1		
			сумісного виконання ІТ			
			проєктів на основі			
			методології CI/CD			
			Варіанти використання			
			системи (Структурна схема)			
	А4	***ІАЛЦ.4672008.005 Д2***	Навчальна система для	1		
			сумісного виконання ІТ			
			проєктів на основі			
			методології CI/CD			
			Взаємодія підсистем			
			(Функціональна схема)			
	А4	***ІАЛЦ.467200.006 Д3***	Навчальна система для	1		
			сумісного виконання ІТ			
			проєктів на основі			
			методології CI/CD			
			Алгоритм роботи системи			
			(Принципова схема)			
	А4	***ІАЛЦ.467200.007 Д4***	Навчальна система для	3		
			сумісного виконання ІТ			
			проєктів на основі			
			методології CI/CD			
			Текст програмного коду			

Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	

ІАЛЦ.467200.001 ОА

Розроб		Костюк А.В.			*Навчальна система для сумісного виконання ІТ проєктів на основі методології CI/CD* Опис альбому	Літ.	Аркуш 1 / Аркушів 1
Перев		Гайдай А.Р.					
						КПІ ім. Ігоря Сікорського, ФІОТ, ІО-92	

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Навчальна система для сумісного виконання IT проєктів на основі методології CI/CD*»

Київ – 2023

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку навчальної системи для сумісного виконання ІТ проєктів на основі методології CI/CD, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування цієї системи є навчання та ознайомлення учнів із сучасними технологіями та практиками CI/CD та DevOps; спрощення життєвого циклу розробки програмного забезпечення.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут імені Ігоря Сікорського».

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка навчальної системи для сумісного виконання ІТ проєктів на основі методології CI/CD, яка має на меті забезпечити ефективний та зручний спосіб навчання практикам та технологіям розробки програмного забезпечення.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

- Простий і інтуїтивно-зрозумілий інтерфейс системи.
- Автоматизація рутинних, повторюваних процесів на різних етапах розробки.
- Автоматичний статичний аналіз коду як частина процесу QA.
- Надати можливість користувачам вносити зміни до налаштувань у ході навчального процесу.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Будь-яке IDE.
- Docker Desktop.
- Веб-браузер Chrome, Firefox, Opera, Edge або Safari.

5.3. Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i3.
- ROM не менше ніж 32 ГБ.
- RAM не менше ніж 16 ГБ.

6. ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	18.12.2022-30.12.2022
Вивчення та аналіз завдання	31.12.2022-12.03.2023
Розробка архітектури та загальної структури системи	13.03.2023-26.03.2023
Розробка структур окремих частин системи	27.03.2023-02.04.2023
Програмна реалізація системи	03.04.2023-16.04.2023
Виправлення помилок	17.04.2023-23.04.2023
Оформлення пояснювальної записки	24.04.2023-21.05.2023

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Навчальна система для сумісного виконання ІТ проєктів на основі методології CI/CD*»

Київ – 2023

ЗМІСТ

ВСТУП

За останнє десятиліття, швидкий розвиток сфери інформаційних технологій та зростаюча складність архітектурних рішень створили нові виклики для команд, що створювали IT продукти. Користуючись традиційними методами, команди розробки та операцій часто працювали незалежно одна від одної, що призводило до різноманітних проблем та неефективності у життєвому циклі розробки. Ці проблеми включали довгі цикли розробки, погану співпрацю між командами, часті помилки та збої, низьку якість та надійність програмних продуктів, а також високі витрати та ризики розгортання.

Щоб подолати ці виклики, створювались нові практики, які пізніше стануть невід'ємною частину циклу розробки в багатьох командах. Зараз вони сформувались у такі методології як DevOps та CI/CD.

Мета даного проєкту – використання створеної системи для навчання, вивчення процесів розробки, їх взаємозв'язок та ознайомлення з згаданими вище методологіями.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА РІШЕНЬ

1.1 Опис та аналіз предметної області

Розробка, впровадження та тестування програмного забезпечення є трьома взаємопов'язаними процесами, які спрямовані на створення високоякісних програмних продуктів для клієнтів.

Розробка програмного забезпечення - це процес проєктування, кодування, дебагінгу та документування програмних додатків або систем відповідно до вимог і специфікацій. Наприклад, розробник програмного забезпечення може використовувати певну мову програмування, таку як Java або Python, щоб створити веб-додаток, який дозволяє користувачам забронювати рейси онлайн.

Розгортання (deployment) програмного забезпечення - це процес установки, конфігурації та запуску програмних додатків або систем на цільових платформах або середовищах. До прикладу, розробник може використовувати такі інструменти як Docker або Kubernetes, щоб розгортати веб-додаток на хмарному сервері, який може обробляти численні запити від користувачів.

Тестування програмного забезпечення - це процес перевірки та валідації програмних додатків або систем, щоб переконатися, що вони відповідають очікуванням і стандартам якості, функціональності, продуктивності та безпеки. Наприклад, тестувальник програмного забезпечення може використовувати Selenium або JUnit, для проведення автоматизованого тестування на веб-додатку, щоб перевірити, чи він працює правильно та ефективно.

Цикл розробки програмного забезпечення, розгортання та тестування може змінюватися в залежності від використовуваної методології розробки,

наприклад, Waterfall, Agile, Scrum, DevOps і т.д. Однак загальний цикл складається з наступних фаз:

- Планування: етап включає в себе визначення обсягу, цілей, досягнень і графіку програмного проекту. Вона також включає в себе виявлення зацікавлених сторін, ресурсів, ризиків та обмежень, пов'язаних з проектом.
- Аналіз: цей етап включає в себе збір і аналіз вимог і специфікацій програмного проекту. Він також містить розробку архітектури та структури програмного забезпечення або системи.
- Розробка: цей етап включає в себе впровадження програмного забезпечення або системи відповідно до дизайну та специфікацій. Проведення одиничних тестів та тестів інтеграції забезпечує коректну і очікувану роботу компонентів програмного забезпечення.
- Доставка: цей етап передбачає розгортання програмного забезпечення або системи на цільову платформу або середовище. На цьому етапі також виконується управління конфігурацією та релізами, щоб забезпечити, що версія ПЗ є послідовною та відстежуваною.
- Тестування: на даному етапі відбувається проведення різних комплексних видів тестування ПЗ або системи, таких як функціональні тести, тести продуктивності, тести зручності, випробування безпеки і т.д. Це також включає в себе звітування та вирішення будь-яких дефектів або проблем, виявлених під час тестування.
- Підтримка: цей етап передбачає надання підтримки та оновлень для програмного забезпечення або системи після розгортання. Вона також включає в себе проведення коректівного, адаптивного та профілактичного технічного обслуговування для забезпечення роботи функціоналу та надійності ПЗ або системи.

Цикл розробки, впровадження та тестування програмного забезпечення є ітеративним і безперервним процесом, який вимагає постійної комунікації та

співпраці між зацікавленими сторонами, розробниками, тестувальниками та користувачами. Цей цикл спрямований на створення програмних продуктів, які відповідають потребам і очікуванням клієнтів, забезпечуючи при цьому якість, ефективність та надійність.

Кожен з цих етапів має великий вплив на загальний хід розробки, тому їх ефективність і надійність відіграє критичну роль при створенні продуктів на сьогоднішньому ринку. Через швидкий розвиток технологій компаніям довелось нарощувати темпи розробки ПЗ, що, без імплементації сучасних методологій, несло негативні наслідки і, як результат, нижчу конкурентоспроможність на ринку.

Серед проблем традиційних методів розробки можна виділити:

- відсутність комунікації, що призводило до непорозумінь та затримок;
- ручне виконання процесів тестування та розгортання, що не лише займало цінний час, а й було схильним до помилок через різні середовища розробки;
- неефективне управління версіями спонукало до рідких і масивних змін, в яких відслідковувати дефекти було складно;
- ручне розгортання оновлень програмного забезпечення часто було складним і схильним до помилок процесом через ризик виникнення помилок у конфігурації та сумісності;
- середовища розробки, тестування, проміжні та виробничі часто були відокремлені - це ускладнювало точне відтворення та тестування реальних сценаріїв, що призводило до невідповідностей між середовищами і збільшувало шанси на помилки;
- відсутність ланцюгів зворотнього зв’язку між командами розробки та операцій означала, що втрачались цінні знання про продукт та можливості для розвитку, низькою була здатність швидко ітеризувати, покращувати якість коду та ефективно вирішувати потенційні збої.

На рисунку 1.1 можна побачити діаграму традиційної методології Waterfall, яку часто наводять у приклад як застарілу модель, основними проблемами якої є неітеративність та виключна послідовність етапів. Попри це, вона все ще має своє місце для вузького спектру задач.

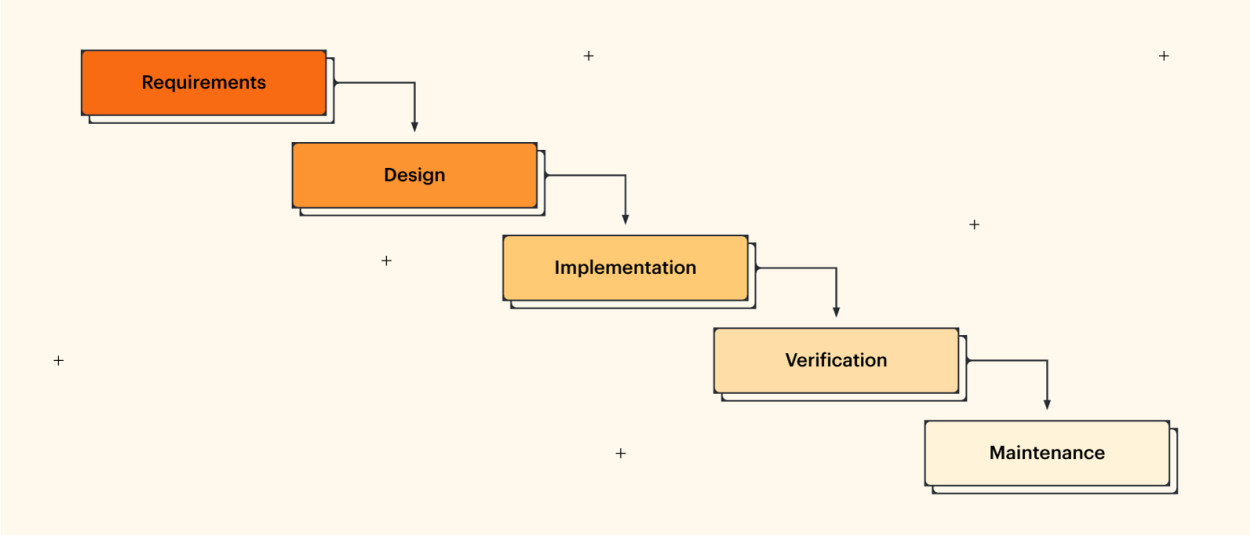


Рисунок 1.1 – методологія розробки Waterfall

Одною з відносно нових методологій, що була призвана вирішувати дані проблеми стала DevOps. Це рух, який виник із співпраці команд розробників і операцій для випуску кращого програмного забезпечення, в коротші терміни. Він був натхненний методологією Agile, яка спрямована задовольнити клієнта завдяки ранній і безперервній доставці цінного програмного забезпечення. Термін DevOps був введений Патріком Дебоєм на конференції Agile в 2008 році, а перша конференція DevOps відбулася в Бельгії в 2009 році[1]. DevOps передбачає застосування мислення та практик Agile для управління інфраструктурним процесом та автоматизації побудови, тестування, інтеграції та доставки програмного забезпечення.

DevOps, абревіатура від Development and Operations, є підходом до розробки програмного забезпечення, який спрямований на поліпшення співпраці та інтеграції між командами розробки та операцій. Методологія базується на тому, що ефективна координація між цими двома командами має вирішальне значення для створення високоякісних програмних продуктів.

DevOps підкреслює автоматизацію процесів, культурні зміни та використання сучасних інструментів і технологій для спрощення розробки, тестування, розгортання та технічного обслуговування програмного забезпечення.

CI/CD або Continuous Integration and Continuous Delivery (Deployment) - це набір практик, які підтримують та інтегруються в підхід DevOps шляхом автоматизації етапів розгортання програмного забезпечення. Continuous Integration відноситься до процесу регулярної інтеграції змін коду в спільний репозиторій. Впродовж циклу CI розробники постійно об'єднують свої зміни в коді, які потім автоматично будуються, тестуються і консолідуються з основною кодовою базою. Це допомагає визначити проблеми інтеграції на ранньому етапі і дозволяє членам команди співпрацювати ефективніше.

Continuous Delivery (Deployment) ґрунтується на CI, автоматизуючи постачання або розгортання змін коду до виробничих та середовищ розробки. На етапах CD весь процес від змін коду до остаточного розгортання автоматизований, що гарантує, що оновлення програмного забезпечення можуть бути доставлені швидко і надійно. Ця автоматизація усуває ручні помилки, скорочує час розгортання та забезпечує більш часті і послідовні випуски.

На рисунку 1.2 відображено цикл розробки у DevOps орієнтованих командах та практики що застосовуються на певних етапах розробки. При цьому варто відмітити, що впродовж всього циклу команда постійно і регулярно спілкується та отримує фідбек по задачам.

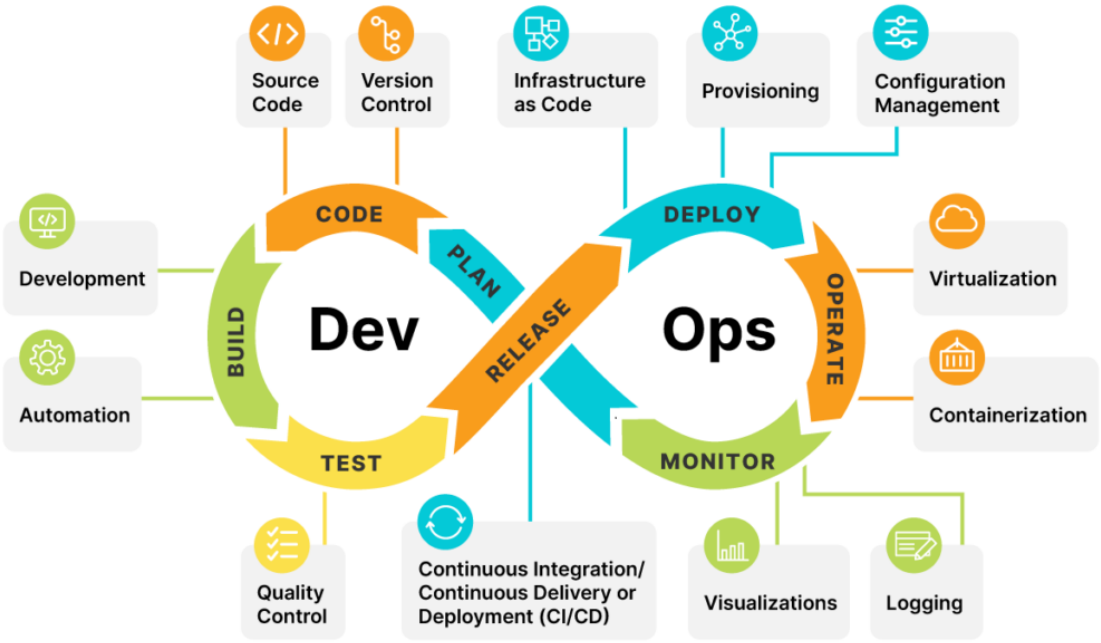


Рисунок 1.2 – цикл розробки у методології DevOps

Роль методології DevOps і CI/CD в розробці програмного забезпечення багатогранна:

- DevOps сприяє більш тісній співпраці та спілкуванню між розробниками, тестувальниками, операційними командами та іншими зацікавленими сторонами. Це руйнує бар'єри, заохочує спільну відповідальність і сприяє культурі співпраці, що призводить до більш швидкого зворотнього зв'язку і кращої відповідності вимогам протягом усього циклу розвитку.
- Автоматизуючи процеси створення, тестування та розгортання, CI/CD скорочує час, необхідний для розробки та доставки оновлень ПЗ. Це пришвидшує ітерації, реагування на вимоги ринку, а також можливість випуску нових функцій і виправлення помилок з більшою ефективністю.
- Аспекти автоматизації та безперервного тестування CI/CD допомагають виявити помилки та проблеми на початку циклу розробки. Це призводить до вищої якості коду, кращої надійності

ПЗ та систем. Здатність вчасно виявляти і вирішувати проблеми зменшує ризик критичних збоїв у виробничих середовищах.

- Методології DevOps та CI/CD спрощують масштабування процесів розробки програмного забезпечення. Автоматизуючи повторювані завдання та використовуючи хмарну інфраструктуру, команди можуть легко розширити свої можливості розробки, тестування та розгортання, щоб задовольнити мінливі вимоги та впоратися зі збільшеним робочим навантаженням.
- DevOps та CI/CD сприяють безперервному зворотньому зв'язку, що дозволяє розробникам збирати уявлення від користувачів, команд операцій та отримувати метрики продуктивності. Цей ланцюг зворотного зв'язку допомагає здійснювати ітеративні вдосконалення, дозволяючи командам реагувати на відгуки, виявляти бар'єри, оптимізувати процеси та постійно покращувати продукт.

Підсумовуючи, методики DevOps та CI/CD були розроблені для вирішення проблем сучасної розробки ПЗ. Вони заохочують до співпраці, автоматизації та безперервного розгортання для підвищення ефективності, швидкості, якості та надійності протягом усього життєвого циклу розробки ПЗ. Застосовуючи ці практики, організації можуть досягти швидшого виходу на ринок, покращеної спрацьованості та якісніших програмних продуктів.

1.2 Постановка задачі

Завданням проєкту є створення навчальної системи для сумісного виконання ІТ проєктів, яка використовуватиметься для спрощення циклу розробки програмного забезпечення, автоматизації та оптимізації рутинних процесів, що допоможе в підсумку збільшити ефективність роботи команди.

Основні можливості даної системи будуть зосереджені на впровадженні методологій та практик DevOps та CI/CD у життєвий цикл розробки програмного забезпечення, що включає в себе:

- Використання системи контролю версій, що використовується на етапі розробки.
- Автоматизація рутинних, повторюваних процесів на різних етапах розробки. Це включає в себе імплементацію циклу CI/CD і покриватиме збірку, тестування та розгортання.
- Автоматичний статичний аналіз коду як частина процесу QA для збільшення надійності коду та покращення зворотнього зв'язку.

Важливими аспектами системи є відкритість коду інструментів, що будуть задіяні, а також те, що система є безкоштовною у використанні. Це дозволить краще освоювати сучасні методи розробки програмного забезпечення та розвивати навички необхідні у сфері IT.

1.3 Огляд наявних комплексних рішень

Швидкий розвиток технологій веб-розробки і обчислювальних потужностей дозволив сильно спростити процес створення веб-сайтів. Конструктори сайтів - це програми або платформи, які дозволяють користувачам створювати веб-сайти без необхідності навичок програмування[2]. Вони використовують візуальні редактори, попередньо створені шаблони та функції перетягування і виведення, щоб зробити дизайн веб-сайту доступним і економічним для початківців. Будівельники веб-сайтів з часом розвивалися, щоб стати більш зручними для користувача і багатими на функції, дозволяючи користувачам легко створювати професійні сайти.

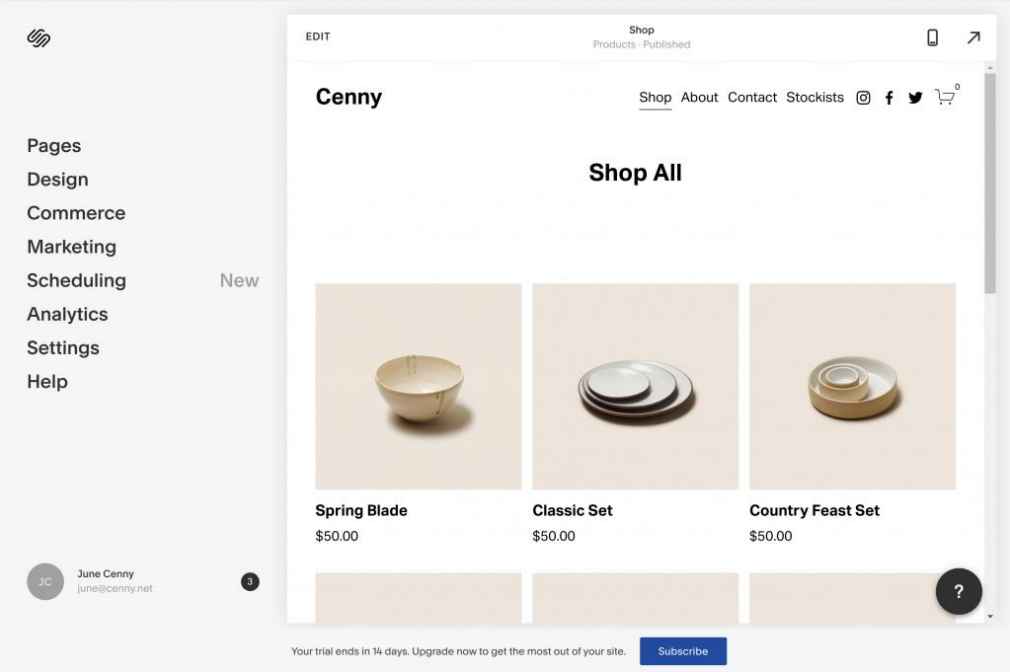


Рисунок 1.3 – інтерфейс веб-конструктора Sqarespace

Однією з головних переваг веб-конструкторів є те, що вони не вимагають ніяких навичок кодингу. Це робить їх відмінним варіантом для приватних осіб і малого бізнесу, які хочуть швидко і легко створити присутність в Інтернеті. Вони також, як правило, більш доступні, ніж наймання професійного веб-розробника, що робить їх економічно ефективним рішенням для тих, хто має обмежений бюджет.

Однак є і деякі недоліки таких інструментів. Один з головних недоліків полягає в тому, що вони можуть обмежувати користувача з точки зору дизайну і функціональності. Конструктори веб-сайтів часто мають певну кількість шаблонів та варіантів дизайну, з яких можна обирати, що може обмежити творчість та гнучкість налаштування. Крім того, вони можуть не мати вузьконаправленого функціоналу, який необхідний для проекту клієнта. Також, з вищесказаного можна зробити висновок, що такі рішення не підійдуть для розробки будь-чого, окрім, власне, веб-сайтів, а також мало придатні для сумісної роботи над проєктами. А отже, будуть хорошим варіантом лише для звичайного користувача, але не для команд розробки ПЗ.

Одним із головних рішень, що спрощує та пришвидшує розробку ПЗ, можна назвати модель PaaS. Platform-as-a-Service - це модель хмарного обчислення, що надає клієнтам повноцінну хмарну платформу для розробки, розгортання та управління додатками без витрат, складностей та нерішучості, що часто приходять із створенням та підтримкою такої платформи на місці (on premise). Рішення PaaS працюють, абстрагуючи основну інфраструктуру, дозволяючи розробникам зосередитися на створенні та розгортанні своїх додатків. Провайдери PaaS пропонують доступ до набору інструментів розробки та виконують технічне обслуговування, масштабування та інші операційні завдання, пов'язані з запуском додатка у виробництві.

Деякі з головних конкурентів на ринку PaaS включають Amazon Web Services (AWS), Google Cloud, IBM Cloud і Microsoft Azure. Популярні PaaS-рішення також доступні як проекти з відкритим кодом (наприклад, Apache Stratos, Cloud Foundry) або від постачальників програмного забезпечення (наприклад, Red Hat OpenShift і Salesforce Heroku)[3].

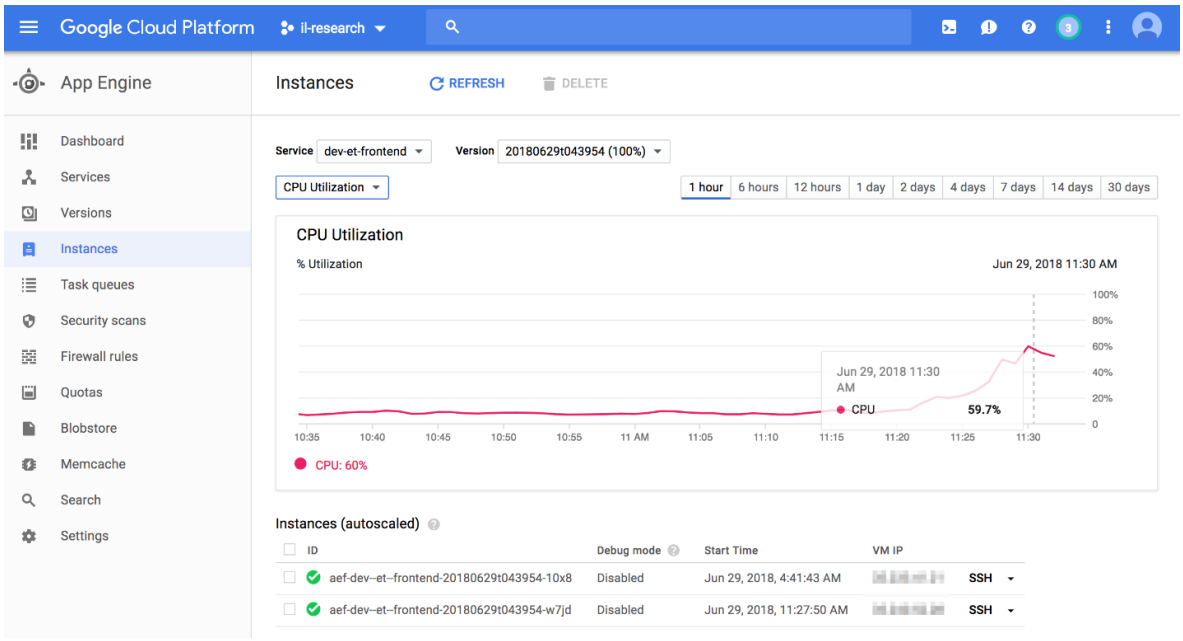


Рисунок 1.4 – панель керування PaaS від Google

PaaS пропонує кілька переваг порівняно з локальною платформою. Це включає в себе менший час виходу на ринок, доступ до більш широкого

спектру ресурсів, більшу свободу експериментувати з меншим ризиком, легку і економічно ефективну масштабованість, а також більшу гнучкість для дослідницьких груп. Однак є й деякі потенційні недоліки використання PaaS. Це може включати блокування провайдерів, обмежений контроль над основною інфраструктурою та потенційні проблеми безпеки.

Така модель є дуже привабливим варіантом для організацій, які дотримуються практик DevOps. PaaS, за рахунок хмарних обчислювальних потужностей, пришвидшує процеси кодингу, тестування та розгортання - деякі з ключових практик Agile і DevOps команд. Крім того, безпосередньо доповнює робочий процес випуску ПЗ у CI/CD і допомагає забезпечити повний цикл релізів методології DevOps. Беручи до уваги життєвий цикл розробки ПЗ, PaaS відповідає за фазу розгортання, що дозволяє розробникам зосередитися на створенні та тестуванні своїх додатків, а платформа виконує технічне обслуговування, масштабування та інші операційні завдання, пов'язані з запуском додатка у роботу. Відповідно до практик DevOps, це може допомогти спростити процес розробки і поліпшити співпрацю в команді.

В цілому, PaaS можуть бути дуже хорошим способом для розробки масштабованих веб-додатків з скромними інвестиціями. Тим не менш, для даної роботи було зроблено вибір на користь локальних платформ розробки. Причина такого рішення полягає в кількох факторах:

- PaaS-рішення за дизайном відбирають у користувача певну свободу конфігурації інфраструктури, що обмежує потенційні випадки, коли систему можна застосувати для навчання.
- Хоча розглянута модель і позиціонується як фінансово гнучка і, в певних випадках, більш вигідна ніж локальна розробка – все ж потребує постійних вкладень. Локальна навчальна система, з іншого боку, може бути розгорнута як на робочому місці, так і на локальному сервері місця навчання і не вимагає додаткових затрат.

ВИСНОВОК ДО РОЗДІЛУ 1

В даному розділі було проаналізовано предметне середовище, а саме циклу розробки ПЗ, методології та підходи. В ході огляду висвітлено ключові проблеми та недоліки традиційних методологій розробки, розглянуто переваги обраної методології DevOps, її практики та основні поняття.

Також було розглянуто існуючі рішення, що можуть вирішувати описані проблеми. Ці системи вирішують поставлені перед ними задачі і мають певні переваги, проте, також містять значні обмеження чи недоліки, через що їх застосування не завжди є однозначно вигідним рішенням.

Враховуючи розглянуті аспекти та навчальну мету проєкту, можна дійти висновку, що актуальним є дослідження та розробка власної системи, яка виконуватиме поставлені цілі та враховуватиме потенційні недоліки присутні в існуючих рішеннях. Система повинна зайняти вузьку нішу, в якій зможе допомогти ефективніше навчатись, розробляти та колаборувати над проєктами.

РОЗДІЛ 2
АНАЛІЗ ТА ВИБІР ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 Системи контролю версій

Системи управління версіями (VCS), також відомі як системи управління вихідним кодом, є інструментами, що допомагають розробникам керувати змінами в їх кодовій базі з плином часу. Вони забезпечують систематичний спосіб відстеження, організації та контролю різних версій або модифікацій файлів в рамках проекту. Системи управління версіями використовуються для поліпшення співпраці, можливості одночасної розробки, відстеження змін та загалом полегшення процесу розробки програмного забезпечення.

Серед популярних варто відмітити наступні системи:

- Git – це, певно, найрозповсюдженіша розподілена система управління версіями, яка отримала значну популярність завдяки швидкості, масштабованості та гнучкості. Вона дозволяє ефективне розгалуження і злиття коду, підтримує розподілені процеси розробки, і має велику екосистему інструментів і послуг, побудованих навколо нього[4].
- Subversion (SVN) – централізована система управління версіями, яка забезпечує більш традиційний підхід до контролю версій. Вона базується на центральному репозиторії та архітектурі клієнт-сервер[5]. Будучи не настільки популярною, як Git, Subversion все ще широко використовується в різних організаціях, особливо для проектів, які вимагають централізованого контролю над доступом до коду.
- Mercurial – система управління версіями, подібна до Git. Пропонує інтуїтивний і зручний для користувача інтерфейс і зосереджується на простоті використання. Хоча вона має меншу базу користувачів в порівнянні з Git, деякі розробники та організації віддають перевагу Mercurial за її лаконічний робочий процес[6].

Для зручності розглянемо переваги систем у табличному форматі:

Таблиця 2.1 – порівняння систем контролю версій

Характеристика	**Git**	**Subversion**	**Mercurial**
Розподіленість	Так	Ні	Так
Робота з гілками	Відмінно	Добре	Відмінно
Робота оффлайн	Так	Частково	Так
Швидкодія	Швидка	Середня	Швидка
Вирішення конфліктів	Ручне	Ручне	Автоматичне
Інтеграція з інструментами CI/CD	Відмінно	Добре	Добре
Підтримка спільноти	Велика	Велика	Середня

Через розповсюдженість, підтримку спільноти та розвинену екосистему в нашій системі використовуватиметься Git. Для сумісної роботи над проєктом також необхідний Git сервер або система управління репозиторіями. Це платформи або інструменти, які відповідають за хостинг, управління та співпрацю у Git репозиторіях. Вони надають інтерфейс для розробників та команд для зберігання, доступу та роботи з їх кодом. Ці системи значно розширюють функціонал Git, і сьогодні практично завжди використовуються разом.

GitHub – одна з найбільш широко використовуваних хостингових платформ, що пропонує набір функцій для співпраці, інтеграції та широку спільноту користувачів.

Деякі переваги використання GitHub включають:

- GitHub дозволяє розробникам легко співпрацювати над проектами, з таким функціоналом, як pull requests, перегляд коду та відстеження проблем.

- Наявність великої і активної спільноти розробників, що полегшує пошук допомоги, внески у проекти з відкритим кодом та пошук нових інструментів і бібліотек.
- Інтеграція з безліччю інших інструментів і сервісів, таких як системи CI/CD, інструменти управління проектами та редактори коду.

Серед недоліків варто виділити:

- Безкоштовний план GitHub має лише обмежену кількість приватних сховищ. Користувачі, яким потрібно більше приватних сховищ, змушені будуть перейти на платний план.
- Використовуючи GitHub, користувачі залежать від третьої сторони для розміщення свого коду. Якщо сервіс стане недоступним або виникнуть інші проблеми, це може вплинути на можливість отримати доступ до свого коду.

Bitbucket – це веб-сервіс хостингу репозиторіїв управління версіями, що належить Atlassian, має підтримку як Mercurial так і Git.

Переваги використання Bitbucket:

- Має найкращу інтеграцію з Jira та іншими продуктами Atlassian, що в комплексі може прискорювати роботу команди.
- Bitbucket Cloud надає вбудований інструментарій CI/CD, пайплайни, що дозволяє збирати, тестувати та розгортати проєкти на Bitbucket[7].

Недоліки використання Bitbucket:

- Безкоштовний план має обмеження щодо кількості користувачів і часу збірок. Користувачі, які потребують більше, повинні переходити на платний план.
- Як і у випадку з GitHub, несе в собі стандартні ризики пов’язані з залежністю від третьої сторони.

Gitea – керований спільнотою, легкий хостинг репозиторіїв, написаний на Go, опублікований під ліцензією MIT.

Переваги Gitea включають:

- Дозволяє користувачам розгортати власний сервіс Git, надаючи їм повний контроль над своїм кодом і даними.
- Має низькі мінімальні вимоги і може працювати на дешевому обладнанні, наприклад, Raspberry Pi.
- Gitea - це програмне забезпечення з відкритим кодом, що дозволяє користувачам переглядати, модифікувати і робити свій внесок у його вихідний код.

Деякі з недоліків:

- Обмежений функціонал порівняно з конкурентами.
- Власний хостинг Gitea вимагає від користувачів самостійно підтримувати власний сервер і оновлювати ПЗ.
- Gitea має меншу спільноту в порівнянні з іншими сервісами, що може ускладнити вирішення проблем та розвиток продукту.

2.2 Системи автоматизації CI/CD

Системи автоматизації CI/CD – це програмні інструменти, які допомагають автоматизувати процеси побудови, тестування та розгортання додатків. Вони призначені для спрощення життєвого циклу розробки програмного забезпечення шляхом автоматизації повторюваних завдань.

Розглянемо декілька популярних інструментів:

Jenkins – сервер автоматизації з відкритим кодом, в якому доступні сотні плагінів для підтримки побудови, розгортання та автоматизування будь-якого проекту[9]. Це автономна програма на основі Java, яка може бути встановлена через npm, Docker, або запускатися автономно на будь-якому комп'ютері з установленим Java Runtime Environment (JRE).

Можливість встановлення безлічі плагінів, в тому числі створених спільнотою, дозволяє нескінченно гнучко налаштовувати Jenkins під майже будь-які потреби. Більш того, він дозволяє налаштовувати розподілення збирання додатків на декількох машинах, що може пришвидшувати процеси

або навіть розпаралелювати збірки для, наприклад, різних платформ. Jenkins легко встановлюється і налаштовується та має функціональний веб-інтерфейс.

З іншого боку, виникає і ряд проблем пов'язаних із його сильними сторонами. В першу чергу, правильна конфігурація сервера для комплексних проектів може викликати складнощі у нових користувачів, в тому числі й через те, що відповідальність за сумісність плагінів між собою та різними версіями покладається на користувача. Також, за усе обслуговування сервера, оновлення системи і безпеку даних відповідає користувач.

Bitbucket Pipelines – інтегрований сервіс автоматизації CI/CD, вбудований в Bitbucket Cloud. Він дозволяє автоматично збирати, тестувати і розгортати свій код на основі файлу конфігурації в репозиторії[8].

Bitbucket Pipelines інтегрований в інтерфейс Bitbucket Cloud, що робить його доступним та зручним у використанні. В свою чергу, це спрощує створення пайплайнів, адже немає необхідності в менеджменті сервера, а також завдяки наявності готових шаблонів для різних мов програмування. Окрім того, згадана раніше інтеграція з Jira дозволяє динамічно отримувати інформацію про статуси пайплайнів.

Однак те, що цей інструмент вбудований в Bitbucket Cloud певною мірою обмежує його можливості, функціонал та гнучкість налаштування. Також варто відмітити, що на безкоштовному рівні певний функціонал чи потужності можуть бути обмеженими і виникне потреба в платній версії.

Таблиця 2.2 – ключові відмінності систем автоматизації

Характеристика	**Jenkins**	**Bitbucket Pipelines**
Екосистема плагінів	Обширна	Обмежена
Кастомізація	Гнучке налаштування	Обмежена
Налаштування середовища збірок	Гнучке	Обмежена
Підтримка спільноти	Обширна	Середня

Звертаючи увагу на характеристики висвітлені у таблиці, було обрано Jenkins для автоматизації процесів CI/CD.

2.3 Системи статичного аналізу коду

Системи статичного аналізу коду - це програмні інструменти, які аналізують вихідний код, не виконуючи його, з метою виявлення потенційних проблем, помилок кодування, вразливостей та дотримання стандартів програмування. Вони допомагають поліпшити якість коду, виявляти помилки на ранніх етапах процесу розробки та застосовувати найкращі практики.

SonarQube – це платформа з відкритим кодом, яка допомагає розробникам та організаціям створювати чистий код шляхом систематичного аналізу якості та безпеки коду[10]. Він може виявляти вразливості у безпеці та проблеми з продуктивністю, а також надає пропозиції щодо їх виправлення. SonarQube може бути інтегрований в пайплайн CI/CD і сумісний з широким спектром мов програмування.

Переваги використання SonarQube включають його здатність виявляти проблеми з якістю коду під час розробки, його інтеграцію з сторонніми інструментами, а також прививання стандартів кодування та збільшення покриття коду за допомогою unit тестів. Він також надає детальний опис та причини проблем і як їх виправити, привносячи цінний досвід для розробників.

Однак, є також деякі недоліки його використання. Один з головних недоліків полягає в тому, що можливі хибні спрацювання, обробка і вирішення яких займає певний час. Крім того, інтерфейс адміністрування може виглядати перегруженим для нових користувачів.

PyLint – статичний інструмент аналізу коду для Python, який перевіряє код на помилки, приводить код відповідно до стандартів і може робити пропозиції про те, як код можна переробити. Дозволяє глибоке налаштування через

модифікацію pylintrc, щоб вказати, які помилки або конвенції важливі для команди.

Однією з переваг PyLint є його здатність проводити глибокий аналіз коду і надавати детальні рекомендації з проблем що виникли. Це може допомогти

визначити потенційні проблеми та запропонувати способи поліпшення якості і стилю коду. Крім того, PyLint можна інтегрувати в більшість IDE, що робить його простим у використанні як частину процесу розробки.

Однак, один з його недоліків полягає в тому, що він може бути не настільки зручним для користувача, як деякі його конкуренти. Окрім того, хоча PyLint має безкоштовну версію, деякі функції доступні тільки в платній версії.

Таблиця 2.3 – порівняння характеристик лінтерів

Характеристика	**SonarQube**	**PyLint**
Підтримка мов	Різні мови	Python
Розпізнавання Code Smells	Так	Так
Розпізнавання вразливостей	Так	Обмежена
Аналіз технічного боргу	Так	Ні
Інтеграція з CI/CD	Так	Так
Налаштування порогів якості	Так	Ні
Інтеграція в IDE	Так	Так (обмежена)
Підтримка спільноти	Обширна	Середня

Враховуючи якості необхідні для виконання завдання проєкту, було обрано SonarQube за можливості гнучкого налаштування та підтримку багатьох мов програмування.

ВИСНОВОК ДО РОЗДІЛУ 2

В цьому розділі було розглянуто інструменти, що використовуються на різних етапах розробки ПЗ, проведено їх аналіз та порівняння.

У якості системи контролю версій, після порівняння серед лідерів ринку, обрано Git в поєднанні з сервісом Gitea. Такий вибір аргументується великою кількістю доступних матеріалів по використанню, а також тим, що Gitea є сервісом з відкритим кодом та безкоштовним. Можливість повного контролю над сервісом також забезпечить гнучкість налаштування, необхідну для навчального процесу.

Після аналізу та порівняння інструментів автоматизації CI/CD Jenkins та Bitbucket Pipelines було зроблено вибір на користь Jenkins. Дана система є універсальною, гнучкою та легко розгортається на локальному сервері. Окрім того, вона теж є проєктом з відкритим кодом та повним безкоштовним функціоналом.

Для статичного аналізу коду вирішено використовувати SonarQube, так як ця система підтримує велику кількість мов програмування, має багатий функціонал не лише моніторингу, а й вирішення проблем. А отже не буде залежати від мови написання навчальних проєктів і потенційно допоможе краще писати код. Вона також має безкоштовну версію, що має відкритий код.

РОЗДІЛ 3
ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Архітектура навчальної системи

Архітектура CI/CD пайплайну зазвичай складається з декількох взаємопов'язаних елементів, які працюють разом для автоматизації процесу доставки програмного забезпечення. В основі лежить система управління версіями, до прикладу Gitea, яка дозволяє розробникам ефективно співпрацювати та керувати кодом. Автоматизаційний сервер відіграє вирішальну роль у оркеструванні різних етапів розробки. Він полегшує безперервну інтеграцію коду шляхом автоматичного створення, тестування та упаковки додатків. Jenkins бездоганно інтегрується з іншими інструментами, такими як SonarQube, потужною платформою управління якістю коду, яка аналізує його для виявлення помилок, вразливостей та технічного боргу.

Структурну схему взаємодії елементів системи відображено в Додатку 1.

3.1.1 Система управління версіями

Gitea функціонує як спільна платформа для команд розробки програмного забезпечення, що дозволяє ефективно контролювати версії та співпрацювати з кодом.

Вона дозволяє користувачам створювати і керувати репозиторіями Git. Користувачі можуть створювати нові сховища або імпортувати існуючі з інших джерел. Кожен репозиторій має свій власний веб-інтерфейс, де користувачі можуть виконувати різні дії, такі як перегляд файлів, перегляд історії комітів та управління гілками та тегами.

Важливими є функції аутентифікації користувачів та контролю доступу. Користувачі можуть створювати окремі облікові записи, а адміністратори можуть визначати ролі користувачів та дозволи доступу. Контроль доступу

можна налаштувати на рівні репозиторію, надаючи тонкий контроль над тим, хто може переглядати, клонувати або вносити зміни до конкретних репозиторіїв.

Gitea інтегрується з CI/CD системами, такими як Jenkins, надаючи декілька варіантів їх налаштування. Через POST/GET запити Gitea повідомляє сервер автоматизації про нові події в репозиторіях, як от push, pull та інші. Ця інтеграція дозволяє розробникам запускати CI/CD пайплайни і виконувати автоматизовані тести та розгортання на основі змін коду, що відбуваються в репозиторії.

3.1.2 Сервер автоматизації

Jenkins, як сервер автоматизації, відповідає за безперервну інтеграцію та доставку програмних проєктів. Він слідує архітектурному підходу master-slave, де сервер управляє загальною системою і координує розподіл робочого навантаження між декількома вузлами. Вузли можуть бути розподілені по різних машинах, що дозволяє паралельно виконувати завдання і ефективно використовувати ресурси.

Функціонал Jenkins значно розширюється через велику екосистему плагінів. Вони надають додаткові функції, інтеграції з різними інструментами, а також підтримку різних мов програмування та інструментів побудови, що активно використовується в нашій системі для коректного налаштування їх взаємодії.

Після того, як сервер отримує запит, що сповіщає про нові дії в репозиторії, він автоматично запускатиме виконання послідовності завдань, які були запрограмовані попередньо - це і являє собою пайплайн. Він має приблизно наступний вигляд у нашому випадку:

- Підготовка середовища для роботи. Це включає в себе налаштування необхідного робочого простору, завантаження відповідної версії

вихідного коду з Git репозиторію і забезпечення наявності будь-яких залежностей або передумов.

- Виконання кроків збірки, визначених в конфігурації завдання. Ці кроки можуть включати компіляцію коду, виконання тестів, генерування артефактів та виконання будь-яких додаткових завдань. Jenkins використовує налаштовані інструменти побудови, компілятори або скрипти для виконання цих дій. В репозиторії проєкту знаходиться Jenkinsfile - текстовий файл, написаний в синтаксисі Groovy, який визначає структуру і етапи роботи. Це є ключовою частиною підходу Jenkins "Pipeline as Code", що дозволяє визначати і контролювати версії пайплайнів разом із вихідним кодом.
- Виконання зазначених тестів, такі як юніт тести, тести інтеграції або статична перевірка коду за допомогою лінтера SonarQube. Він записує результати тестів і генерує звіти, щоб забезпечити видимість результатів тестів.
- Після успішного завершення збірки та тестування, генеруються артефакти. Ці артефакти можуть бути виконуваними файлами, пакетами розгортання, документацією або будь-якими іншими вихідними файлами, визначеними в конфігурації роботи.
- Протягом всього процесу виконання Jenkins записує та оновлює статус роботи. Він генерує журнали збірок, звіти та повідомлення, щоб забезпечити наглядність прогресу, успіху або невдачі збірки. Ці звіти доступні через інтерфейс користувача або інтегруються з зовнішніми системами звітності.
- Після завершення роботи Jenkins очищає робочий простір, видаляючи тимчасові файли або артефакти, створені під виконання. Це забезпечує чисте середовище для подальших завдань і запобігає його забрудненню.

3.1.3 Система статичної перевірки коду

SonarQube інтегрується з Jenkins для проведення аналізу коду в рамках процесу CI/CD. Далі розглянуто їх взаємодію та кроки, що виконує SonarQube:

- Під час виконання завдання Jenkins викликає сканер SonarQube за допомогою плагіна, який взаємодіє з сервером SonarQube. Сканер аналізує вихідний код на основі налаштованих наборів правил, профілів якості та плагінів аналізу коду. Він вивчає шаблони коду, складність, дублювання та дотримання стандартів кодування.
- Результати аналізу надсилаються на сервер SonarQube для обробки. Передається інформація про якість коду, виявлені проблеми, охоплення тестів та інші показники.
- Сервер SonarQube отримує результати аналізу від сканера та обробляє їх, застосовуючи правила та метрики, а також генерує детальні звіти та наповнення для сторінки показників. SonarQube також оцінює відповідність коду порогу якості, налаштованому для проекту. Поріг якості визначає критерії для прийнятної якості коду на основі таких показників, як охоплення коду, дублювання коду та серйозність проблем.
- Jenkins записує результат аналізу та звіти, створені SonarQube, оновлює стан збірки і відображає отримані дані в інтерфейсі користувача Jenkins.
- На основі цих результатів, Jenkins може динамічно змінювати хід виконання роботи, блокуючи розгортання або запускаючи надсилання повідомлень, генерування звітів або позначення збірки як нестабільної або невдалої.

Для своєї роботи SonarQube вимагає також наявність реляційної бази даних та системи Elasticsearch.

Реляційна база даних - це тип БД, що організовує дані в рядки і стовпці, які колективно утворюють таблицю. Дані зазвичай структуровані в декількох таблицях, які можна об'єднати за допомогою первинного ключа або

зовнішнього ключа. Ці унікальні ідентифікатори демонструють різні відношення, які існують між таблицями, приклад можна побачити на рисунку 3.1. Система, що використовується для підтримки реляційних баз даних, є системою управління реляційними базами даних (РСУБД). В роботі з ними використовують SQL-запити для управління даними, що містять таблиці.

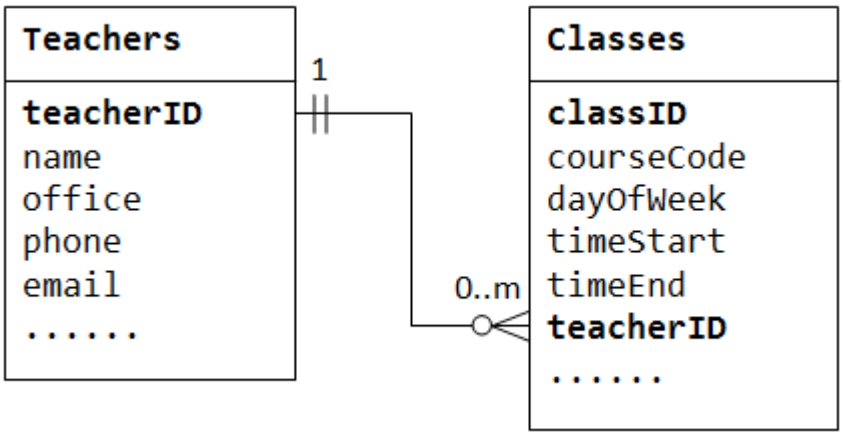


Рисунок 3.1 – відношення між таблицею вчителів та занять

В нашій системі серед доступних варіантів РСУБД було обрано PostgreSQL. Це широко використовувана БД з відкритим кодом, яка підтримує як JSON (нереляційні) так і SQL (реляційні) запити. Вона містить більш ніж тридцять років активного розвитку спільнотою, що сприяло її репутації як цілісної, надійної, стійкої та продуктивної бази даних.

SonarQube використовує РСУБД для зберігання та управління постійними даними, пов'язаними з аналізом коду. У базі зберігається інформація, така як конфігурації проектів, результати аналізу, метрики, проблеми та дані історії та пов'язані з ними метадані для подальшого пошуку та звітування.

Elasticsearch це розподілений, RESTful пошуковий та аналітичний рушій, здатний вирішувати все більшу кількість задач. Він є безкоштовним, з відкритим кодом, для всіх типів даних, включаючи текстові, чисельні, геопросторові, структуровані та неструктуризовані. Система побудована на

Apache Lucene і була вперше випущена в 2010 році компанією Elasticsearch N.V. (тепер Elastic). Рушій відомий своїм простим REST API, можливістю розподілу, швидкістю та розширюваністю.

3.2 Релізація навчальної системи

Для створення системи необхідно буде встановити сервери Jenkins, SonarQube, а також PostgreSQL з подальшим їх налаштуванням. Встановлення Gitea, в свою чергу, не буде виконуватись в межах цієї роботи. Це є можливим за рахунок вже наявного сервера, розгорнутого на серверах університету. Для цієї роботи розгортання буде виконуватись на локальній машині, але можливі також і інші варіанти, якщо цього потребує ситуація.

Для нашої системи важливо впевнитись, що ОС хоста має відповідну версію JDK та/або JRE, так як більшість компонентів побудована на Java та працює використовуючи JVM.

JDK, JRE та JVM є трьома основними пакетами, що використовуються в програмуванні на Java. JDK (Java Development Kit) – це середовище розробки програмного забезпечення, яке пропонує набір інструментів і бібліотек, необхідних для розробки Java-базованих програмних додатків і апплетів. JRE (Java Runtime Environment) – це набір програмних інструментів, відповідальних за виконання програм Java у системі. JVM (Java Virtual Machine) відповідає за завантаження, перевірку та виконання байтового коду Java.

JVM є основою, або серцем мови програмування Java, і гарантує, що вихідний код програми Java буде незалежним від платформи. JVM входить як в JDK, так і в JRE – програми Java не можуть працювати без нього.

Враховуючи вимоги та специфіку системи, було прийнято рішення скористатись платформою та інструментами Docker для оптимізації як розробки, так і подальшої експлуатації системи.

Docker - це платформа з відкритим кодом, що дозволяє розробникам автоматизувати розгортання та управління додатками. Він забезпечує ефективний спосіб упаковки додатків разом з його залежностями в легких та портативних контейнерах.

Контейнеризація – це полегшена імплементація віртуалізації, що дозволяє упаковувати та ізолювати додатки та їх залежності в самостійні одиниці, що називаються контейнерами. Кожен контейнер забезпечує ізольоване середовище, яке включає в себе додаток, його залежності, бібліотеки та конфігурації, забезпечуючи стабільність та портативність у різних обчислювальних середовищах.

Контейнери працюють на рівні оперативної системи. На відміну від традиційної віртуалізації, яка запускає певну кількість віртуальних машин на гіпервізорі, контейнери працюють з ядром операційної системи хоста. Цей підхід усуває необхідність окремої гостьової операційної системи для кожного контейнера, що призводить до збільшення швидкодії та ефективності використання ресурсів.

Гіпервізори є програмними компонентами, які забезпечують віртуалізацію шляхом абстрагування фізичних апаратних ресурсів, що й дозволяє створення віртуальных машин, які можуть працювати з різними операційними системами та додатками. Умовну модель підходів до віртуалізації через віртуальні машини та контейнеризацію можна побачити на рисунку 3.2

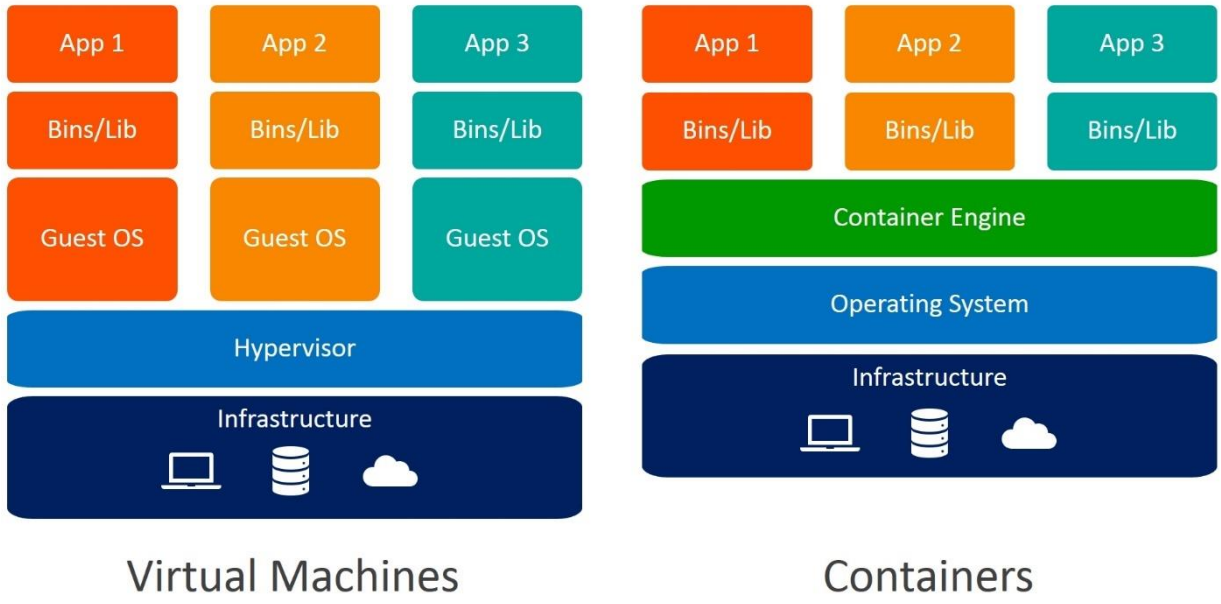


Рисунок 3.2 – моделі віртуалізації

Контейнерні зображення (образи) служать «кресленнями» для створення контейнерів. Зображення містить код програми, час запуску, бібліотеки, системні інструменти та залежності, необхідні для запуску програми. Зображення зазвичай будуються за допомогою декларативного текстового файлу під назвою Dockerfile, який визначає інструкції для створення зображення, включаючи базовий образ, установку залежностей, конфігурацію та точку входу для програми.

За рахунок контейнеризації забезпечуєтья високий рівень ізоляції між додатками, їх залежностями. Кожен контейнер працює у власному ізольованому середовищі, гарантуючи, що зміни або неполадки в одному контейнері не повпливають на інші[11]. Це допомагає запобігти конфліктам між різними додатками або версіями програмного забезпечення, що працюють на одній системі.

Docker дозволяє розробникам самостійно створювати контейнерне зображення, що містить додаток та його залежності, таким чином сприяючи відтворюваності розгортань. Обмінюючись образами, інші розробники та

системи можуть відтворити те ж саме середовище, забезпечуючи стабільну поведінку на різних етапах процесу розробки та розгортання.

Важливим, у випадку використання для кінцевих продуктів, буде легке масштабування програми горизонтально, одночасно запускаючи кілька контейнерів. Контейнерами можна оркеструвати і управляти за допомогою контейнерних оркестраційних платформ, таких як Kubernetes, які автоматизують масштабування, балансування навантаження та відновлення.

3.2.1 Docker

Перш за все, необхідно встановити Docker Desktop – застосунок, що надає зручний для користувача інтерфейс і набір інструментів для розробки, створення та запуску контейнерів на операційних системах Windows та macOS. Він включає в себе Docker Engine, який необхідний для використання можливостей докера і відповідає за створення, керування та управління контейнерами.

З офіційного сайту розробника завантажуємо відповідний до системи інсталлер та за інструкціями проводимо первинне налаштування. Запустивши додаток, можемо побачити графічний інтерфейс (рис. 3.3), що дозволяє переглядати та керувати запущеними контейнерами, моніторити використання ресурсів, конфігурувати налаштування мережі та взаємодіяти з журналами контейнерів та терміналами через GUI.

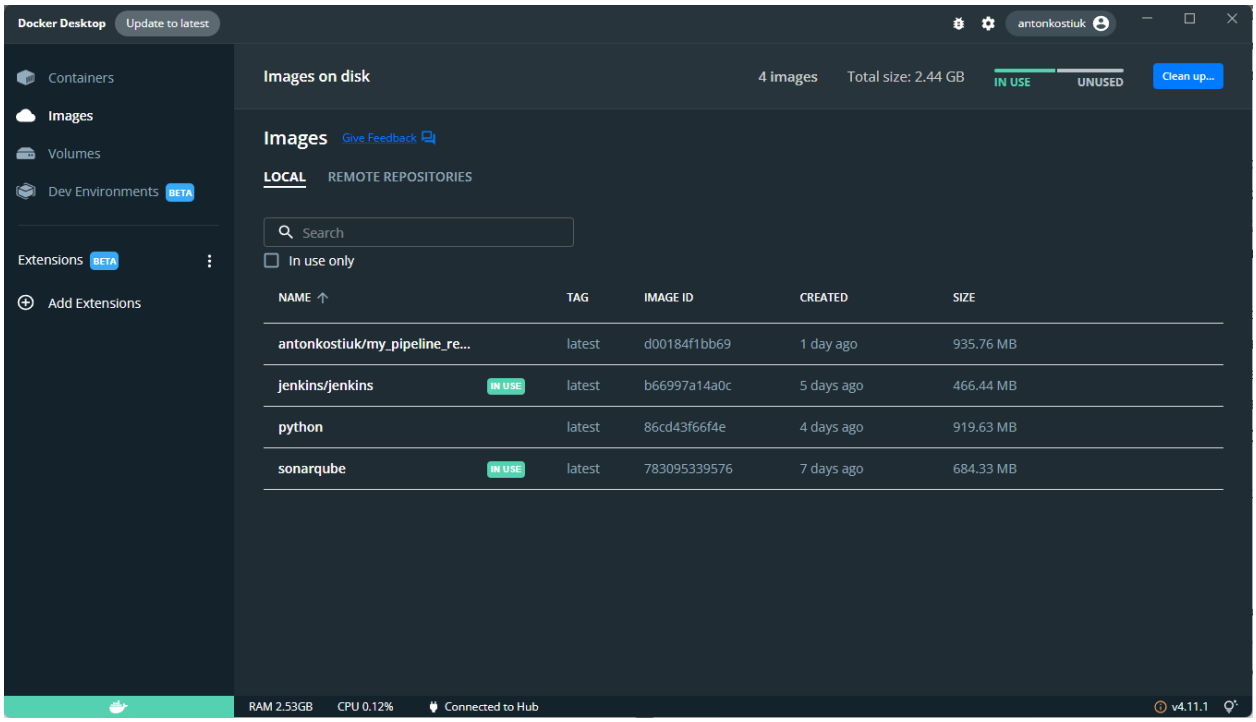


Рисунок 3.3 – інтерфейс Docker Desktop

Далі переходимо до проєктування контейнерів. Для цього скористаємось утилітою docker-compose. Це інструмент, який дозволяє визначати та керувати багатоконтейнерними додатками Docker. Він надає декларативний спосіб проєктування сервісів, мереж та локальних дисків, необхідних для додатка, що робить його дуже зручним для оркестрування та запуску декількох контейнерів як єдиного стеку додатків. Щоб почати роботу, необхідно створити файл з назвою docker-compose.yml – зробимо це у корні директорії нашого проєкту. Цей файл включає в себе конфігурації для кожного сервісу – зображення, яке потрібно використовувати, змінні середовища, мережі, локальні диски та інші налаштування. Можна визначити залежності між службами, що дозволяє їм взаємодіяти між собою.

Файл docker-compose пишеться декларативною мовою YAML – це зручний для читання людиною формат серіалізації даних, який використовується для визначення конфігурації елементів у файлі. Він використовує індентацію та відступи для представлення структури та ієрархії даних.

Синтаксис YAML дозволяє визначати пари ключових значень, списки та вкладені структури. Пара ключових значень записується як key: value, де key представляє варіант конфігурації, а value – його асоційоване значення. Списки представлені елементами, що записуються через дефіс (- item1, - item2, і т.д.), що дозволяє визначати кілька елементів одного типу.

Подальше налаштування усіх сервісів розберемо окремо для кожного з них у наступних підрозділах.

3.2.2 Docker-compose

Переходимо до створеного файлу і прописуємо конфігурацію для контейнера, що міститиме сервер Jenkins.

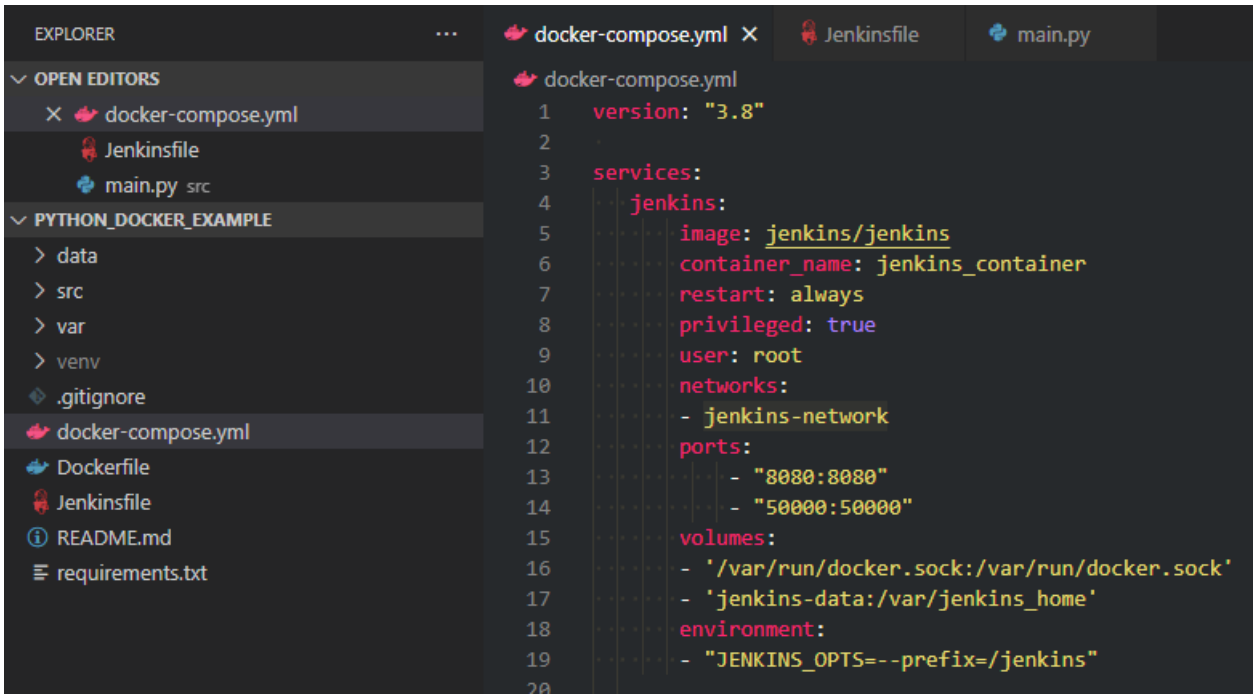


Рисунок 3.4 – конфігурація контейнера Jenkins

Спочатку необхідно вказати версію файлу, останньою на даний момент є 3.8. Далі конфігуруємо сервіс. За основу контейнера обираємо образ jenkins/jenkins, який, як і багато інших образів, можна знайти на ресурсах DockerHub. Він уже містить сервер Jenkins, тож нам залишається правильно його налаштувати.

Вказуємо ім’я контейнера на наш розсуд; виставляємо значення true для поля restart – це вказує оркестратору, що сервіс необхідно завжди перезапускати, якщо він за якихось обставин припиняє роботу. Так само робимо й для інших двох контейнерів.

Також в усіх контейнерах вказуємо мережу jenkins-network, яку створимо в кінці цього ж файлу. При запуску контейнерів docker автоматично створює спільну мережу для всіх контейнерів, проте архітектура системи вимагає вручну прописати власну мережу, так як пізніше ми налаштовуватимемо автоматичний запуск додаткового окремого контейнера, що повинен мати доступ до мережі.

Вказуємо для контейнера порти 8080:8080 та 50000:50000. Це прокидає порти з контейнера до ОС хоста, де 8080 будемо використовувати для доступу до веб-інтерфейсу сервера та комунікації між сервісами, а порт 50000 використовується Jenkins для комунікації з агентом Java Web Start (JNLP). JNLP – це агент на основі Java, який дозволяє Jenkins встановити канал зв'язку з віддаленими агентами для розподілених побудови та розгортання. Коли Jenkins запускає роботу, яка вимагає додаткових ресурсів або має бути виконана на окремій машині, вона може динамічно створювати агенти JNLP. Вони відповідають за виконання завдань роботи і звітування про результати назад до сервера.

Наступним кроком налаштовуємо мапінг локальних дисків контейнера у полі volumes. Volumes дозволяє контейнерам зберігати та обмінюватися даними. Зліва від двокрапки вказується ресурс хоста, а справа – контейнера. Для нас вони будуть корисні з декількох причин:

- Дозволяють контейнерам зберігати дані постійно, окремо від життєвого циклу контейнера. Це гарантує, що дані залишаються недоторканими і доступними навіть у разі видалення або заміни контейнера.
- Томи також можуть бути використані для інтеграції контейнерів з хостинговою системою шляхом монтажу каталогів або файлів. Це

дозволяє контейнерам отримувати доступ до файлів на хості, що робить зручним обмін кодом, файлами конфігурації або іншими ресурсами.

По-перше, змонтуємо /var/run/docker.sock. Цей файл служить інтерфейсом, через який клієнт Docker взаємодіє з демоном Docker для менеджменту та управління контейнерами, зображеннями, мережами та іншими операціями, пов'язаними з Docker.

По-друге, підключимо створений нами в файлі docker-compose том jenkins-data: до директорії /var/jenkins_home у контейнері. Цей том буде використовуватись як постійне сховище даних для сервера Jenkins. Таким чином усі подальші налаштування будуть зберігатись не лише між перезапусками, а й при видаленні самих контейнерів.

І наостанок, через ключ environment встановлюємо значення змінної середовища JENKINS_OPTS=--prefix=/jenkins. Цей крок не є обов'язковим, так як лише переносить лендінг сторінку на контекст /jenkins.

Перейдемо до конфігурації сервісу SonarQube, що можна побачити на рисунку 3.5.

docker-compose.yml | Jenkinsfile | main.py

docker-compose.yml

```
21    sonarqube:
22        image: sonarqube
23        container_name: sonar_container
24        restart: always
25        privileged: true
26        depends_on:
27          postgresdb:
28            condition: service_healthy
29        environment:
30            - SONAR_SEARCH_JAVAADDITIONALOPTS=-Dnode.store.allow_mmap=false
31            - SONAR_JDBC_URL=jdbc:postgresql://postgresdb:5432/sonarqube?currentSchema=public
32            - SONAR_JDBC_USERNAME=sonar
33            - SONAR_JDBC_PASSWORD=sonar
34        networks:
35            - jenkins-network
36        ports:
37            - "9000:9000"
38        volumes:
39            - 'sonar-data:/opt/sonarqube/data'
40            - 'sonar-data:/opt/sonarqube/extensions'
41            - 'sonar-data:/opt/sonarqube/lib/bundled-plugins'
42
```

Рисунок 3.5 – конфігурація контейнера SonarQube

За допомогою ключа environment через змінні середовища вказуємо облікові дані та веб-адресу для користувача сервера PostgreSQL. Це необхідно для можливості комунікації між серверами та доступу до бази даних. Далі перекидаємо порт 9000 – він буде використовуватись для доступу до веб-інтерфейсу та комунікації між SonarQube та Jenkins.

У томах вказуємо три директорії: /data, /extensions та /lib/bundled-plugins. Вони необхідні для збереження конфігурації.

Використаємо ключ depends_on. Він використовується для налаштування порядку запуску контейнерів. Так як SonarQube залежний від бази даних – необхідно, аби на момент запуску БД вже могла відповідати на запити. За замовчуванням, цей ключ не очікує повного розгортання сервісу і готовності його до роботи. Отже, вказуємо ключ сервісу від якого залежить запуск SonarQube – postgresdb, а під ним ключ condition зі значенням service_healthy. У такій конфігурації докер буде очікувати, поки не отримає від сервісу postgresdb позитивну відповідь на healthcheck, що ми визначимо в налаштуваннях пізніше.

Наступним кроком буде налаштування сервісу PostgreSQL, що відображено на рисунку 3.6.

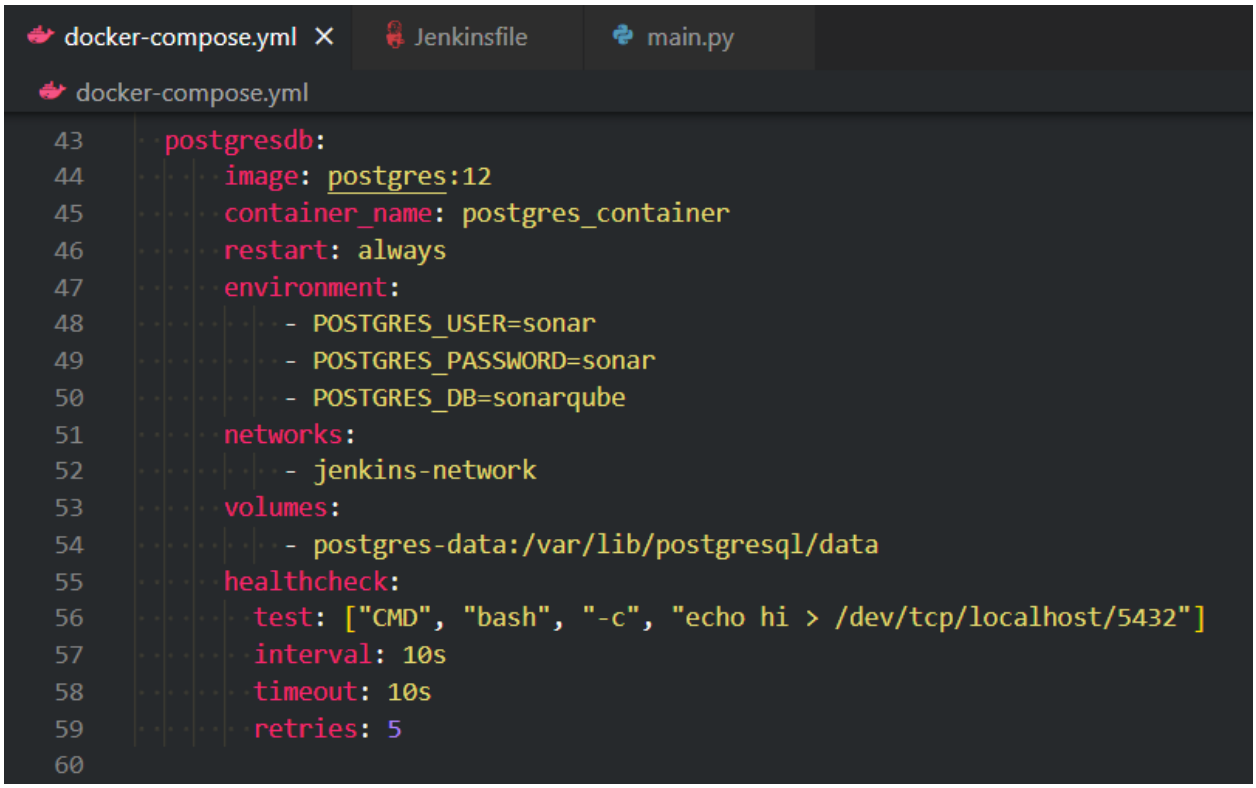


Рисунок 3.6 – конфігурація сервісу PostgreSQL

Знову скористаємось environment, щоб через змінні середовища створити обліковий запис користувача, яким буде користуватись SonarQube, а також необхідну базу даних. Як і в попередніх конфігураціях створимо том для збереження налаштувань.

Нижче визначимо перевірку здоров'я сервісу. Ключ тест відповідає за те, яким способом буде перевірятись стан – команда шелл, скрипт або додаток. Команда повинна повернути «0» у випадку успіху. Інтервал визначає час між запуском команди, а таймаут – максимальний час очікування на виконання команди. Якщо команда не виконується за цей термін, то healthcheck буде провалено.

Декларацію мереж та томів, якими ми користувались раніше продемонстровано на рисунку 3.7.

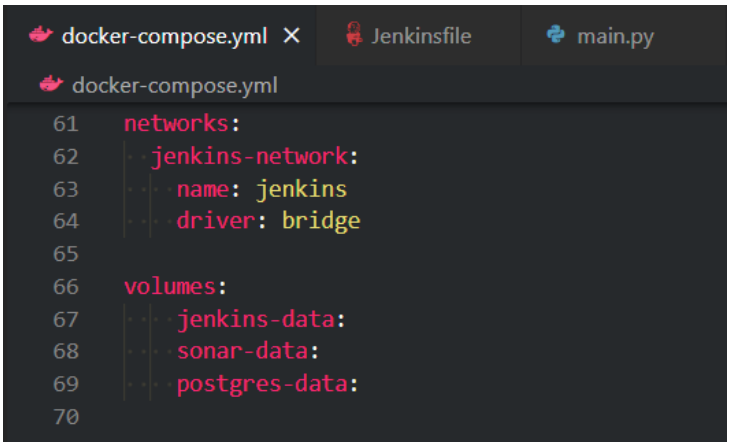


Рисунок 3.7 – конфігурація мережі та томів

На цьому конфігурацію файлу docker-compose завершено. Наступним кроком буде запуск контейнерів для подальшого налаштування серверів за допомогою веб-інтерфейсів. Для цього відкриємо термінал у директорії, що містить файл та виконаємо команду для запуску як показано на рисунку 3.8.

```
PS D:\Study\QA\python_docker_example> docker-compose up -d
Creating network "jenkins" with driver "bridge"
Creating volume "python_docker_example_sonar-data" with default driver
Creating volume "python_docker_example_postgres-data" with default driver
Pulling postgresdb (postgres:12)...
12: Pulling from library/postgres
f03b40093957: Pulling fs layer
9d674c93414d: Pulling fs layer
de781e8e259a: Pulling fs layer
5ea6efaf51f6: Pulling fs layer
b078d5f4ac82: Pulling fs layer
97f84fb2a918: Pull complete
5a6bf2f43fb8: Pull complete
f1a40e88fea4: Pull complete
ea9811dc38ae: Pull complete
515634916e47: Pull complete
81b43500849b: Pull complete
800a983fb77c: Pull complete
3ecd088e33c5: Pull complete
Digest: sha256:7db33237a29afa0a62998b7b6707bfe99766e9da02b5780f8db8f773b85c8f33
Status: Downloaded newer image for postgres:12
Creating postgres_container ... done
Creating jenkins_container  ... done
Creating sonar_container    ... done
```

Рисунок 3.8 – виконання команди запуску контейнерів

В команді вказано опцію -d – при використанні з up це вказує докеру, що контейнери треба запустити в режимі detached. В ньому контейнери

запускаються і працюють в фоновому режимі, не підключаючи вивід консолі до терміналу. Це дозволить використовувати термінал для наступних команд, поки контейнери запущені.

Відповідно до файлу конфігурації, докер спочатку створить усі необхідні томи та мережі, якщо їх ще немає, потім завантажить вказані образи контейнерів і перейде до запуску. Якщо все пройшло успішно – повинні побачити всі три контейнери запущеними, використавши команду відображену на рисунку 3.9.

```
PS D:\Study\QA\python_docker_example> docker-compose ps
       Name                     Command               State                         Ports
-------------------------------------------------------------------------------------------------------------------
jenkins_container    /usr/bin/tini -- /usr/loca ...   Up            0.0.0.0:50000->50000/tcp, 0.0.0.0:8080->8080/tcp
postgres_container   docker-entrypoint.sh postgres    Up (healthy)  5432/tcp
sonar_container      /opt/sonarqube/docker/entr ...   Up            0.0.0.0:9000->9000/tcp
PS D:\Study\QA\python_docker_example>
```

Рисунок 3.9 – список запущених контейнерів

У виводі команди docker-compose ps можемо знайти інформацію про назви контейнерів, останню виконувану команду, стан сервісу та маппінг портів контейнера.

3.2.3 SonarQube

Перш за все, переходимо за адресою http://localhost:9000/. Потрапимо на сторінку логіну, за замовчуванням, при першому запуску ім’я та пароль користувача – admin. Далі сервер запросить змінити цей пароль, тож вводимо новий на наш вибір.

Відбудеться переадресація та ми потрапимо на головну сторінку, звідки можна керувати проєктами, налаштуваннями перевірок та адмініструванням, а також побачити інформацію про перевірки що вже відбулись.

Перейдемо у вкладку Administration → System → System та перевіримо чи підключена обрана нами база даних. Значення повинні бути схожі до відображених на рисунку 3.10.

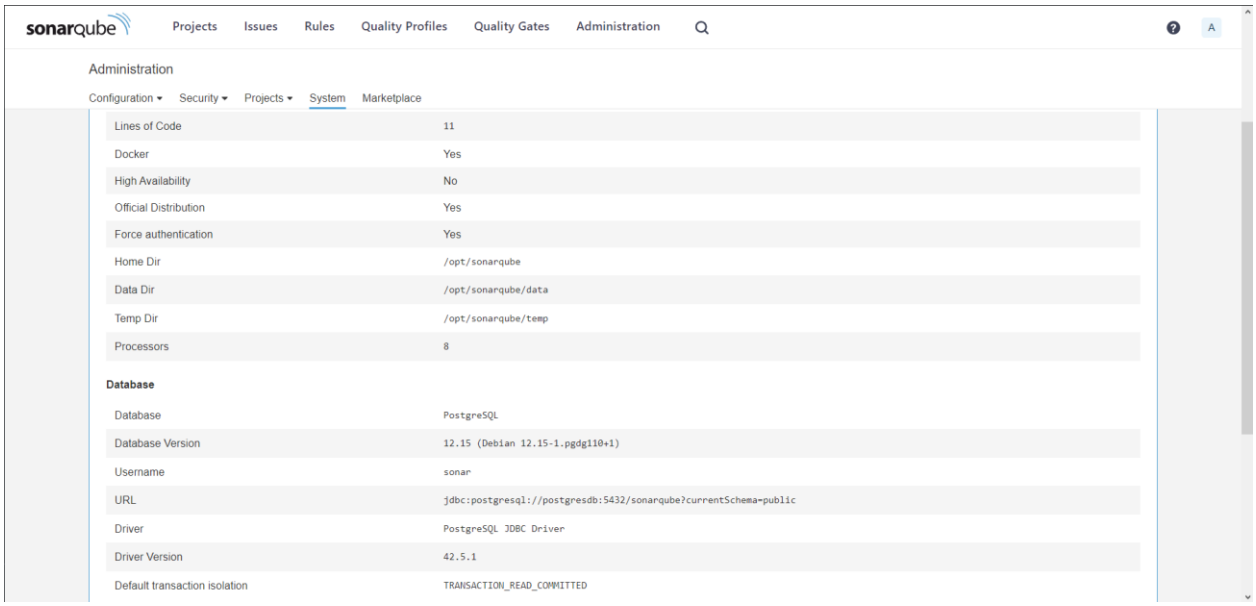


Рисунок 3.10 – інформація про підключену БД

У випадку, якщо відображається вбудована БД «H2» - необхідно перевірити конфігурацію, адже вона рекомендується для постійного використання.

На цій же сторінці відкриємо вкладку Security зліва зверху та у випадаючому списку оберемо Users. Побачимо нашого користувача, тут можна за бажанням змінити дані акаунту. Нам необхідно буде створити токен доступу до сервера SonarQube.

Для створення натискаємо на відповідне поле під Tokens і в вікні (рисунок 3.11) вводимо довільну назву, після цього натискаємо Generate і зберігаємо токен.

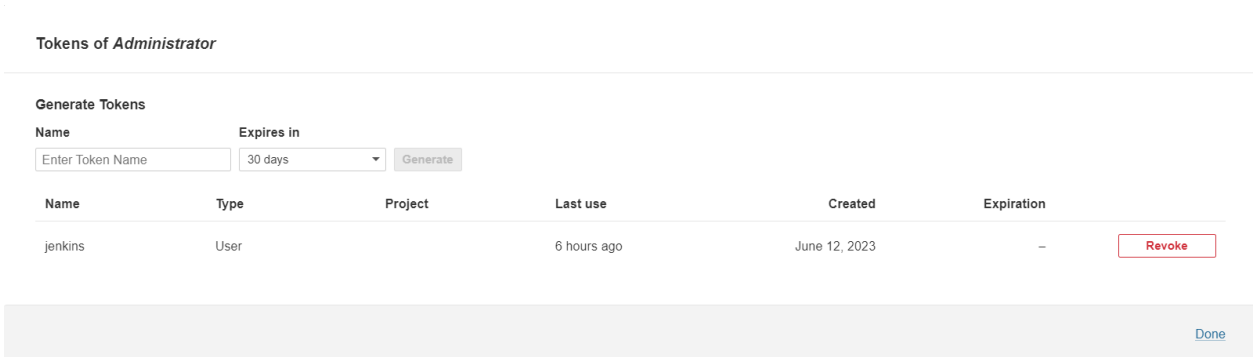


Рисунок 3.11 – створення токена автентифікації у SonarQube

Токени доступу в системах CI/CD заміняють облікові дані акаунтів та використовуються для надання дозволів і доступу до різних ресурсів в рамках CI/CD пайплайну. Вони відіграють важливу роль у забезпеченні та контролі взаємодії між різними компонентами системи. Токени гарантують, що лише авторизовані суб'єкти можуть виконувати дії, такі як завантаження артефактів та зображень або розгортання додатків. Як правило, після створення дається лише одна можливість скопіювати токен, тож варто одразу зберегти його в безпечному місці для подальшого використання.

За бажанням також можна налаштувати різноманітні варіанти перевірок коду та пороги якості, але в рамках нашого проєкту обмежимося вбудованими засобами тестування.

3.2.4 Jenkins

Переходимо до налаштування Jenkins. В нашому проєкті сервер буде доступний за адресою http://localhost:8080/jenkins. При першому логіні Jenkins запросить одноразовий пароль, що надається в процесі встановлення. Для того, щоб знайти цей пароль простіше всього скористатись наступною командою, яку введемо в тому ж терміналі, звідки запускали контейнери:

docker exec -it jenkins_container cat /var/jenkins_home/secrets/initialAdminPassword

Після введення паролю Jenkins запропонує встановити стандартні розширення. Погоджуємось та очікуємо поки процес завершиться. Далі вводимо дані облікового запису або ж пропускаємо крок і переходимо одразу на головну сторінку (рисунок 3.12).

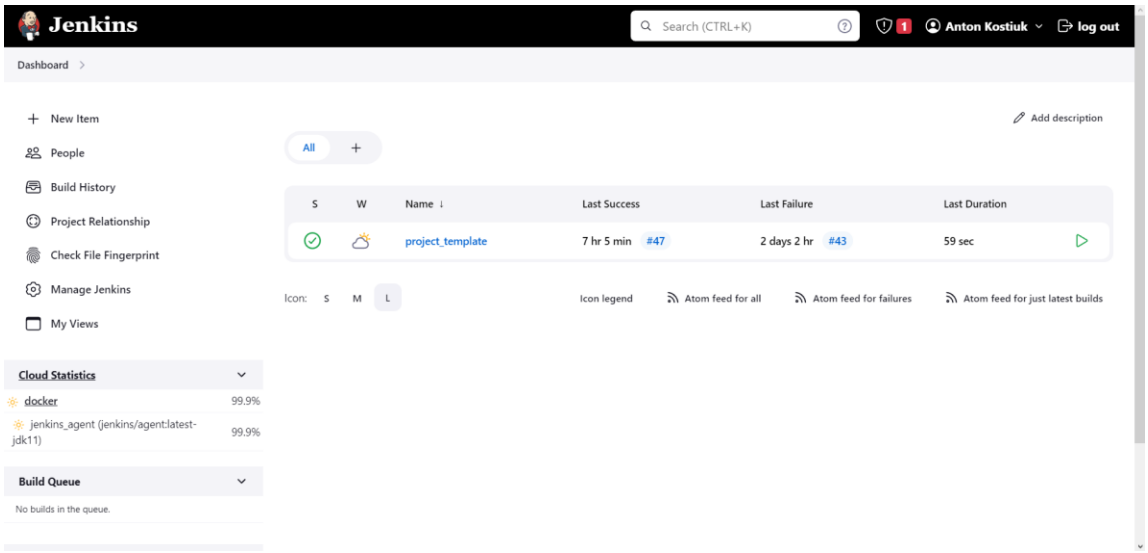


Рисунок 3.12 – головна сторінка Jenkins

Тут відображаються створені роботи, інформація про їх стан та запуски. Через меню зліва можна перейти до детальнішої інформації, історії виконання робіт або ж створити нову. Проте нам необхідна саме вкладка Manage Jenkins.

Ця сторінка містить усі налаштування сервера. Спершу, встановимо необхідні плагіни. Для цього переходимо у Plugins → Available plugins, встановлюємо Build Authorization Token Root Plugin, Docker plugin, Gitea Plugin та SonarQube Scanner for Jenkins. Перезапускаємо сервер після встановлення виставивши галочку внизу сторінки.

Перевіряємо, що всі плагіни встановились коректно перейшовши на сторінку Installed plugins. Jenkins повинен автоматично встановити додаткові плагіни-залежності для обраних нами, тож кінцевий список розширень відображений на рисунку 3.13.

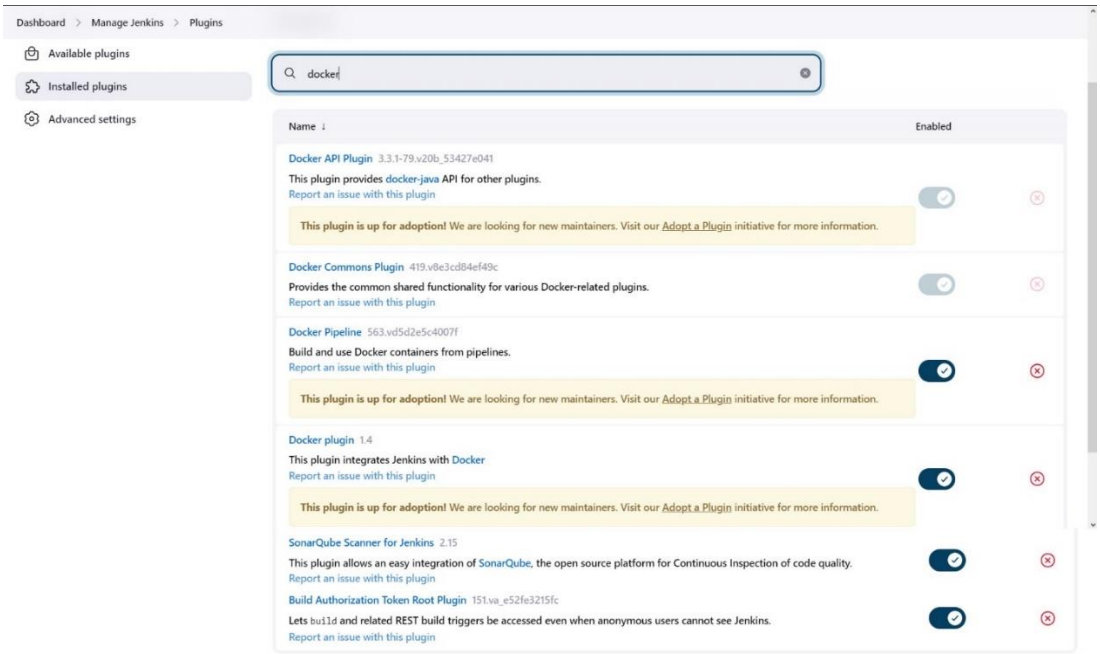


Рисунок 3.13 – інстальовані плагіни

Далі переходимо до Manage Jenkins → Tools для налаштування інструментів. Знаходимо пункт SonarQube Scanner installations (рисунок 3.14), додаємо сканер, налаштовуємо назву та обираємо варіант Install automatically. Ця опція дозволяє встановлювати інструменти по вимозі, під час виконання роботи.

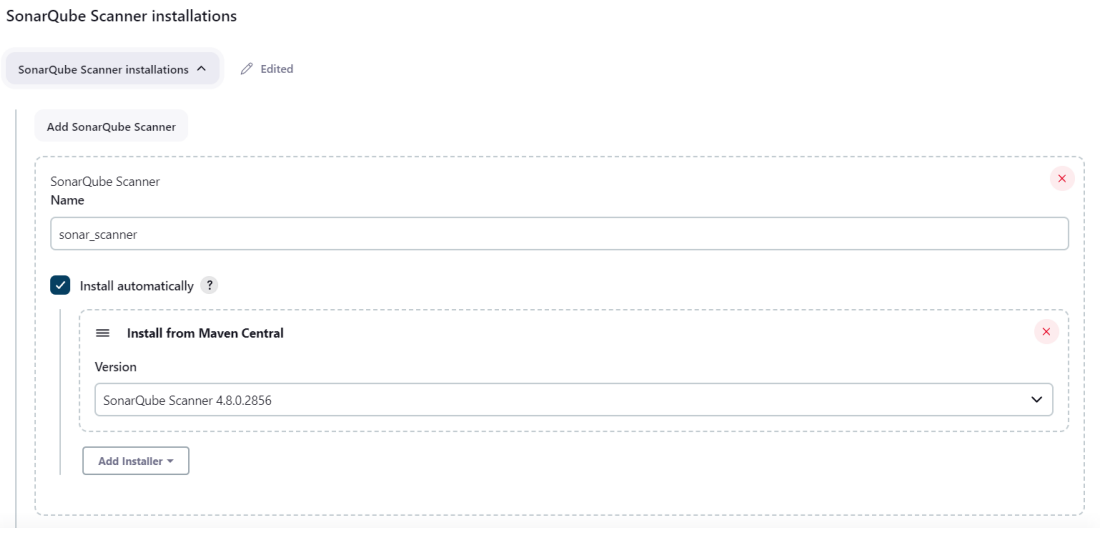


Рисунок 3.14 – налаштування сканера SonarQube

Після виклику сканера, він аналізує вихідний код на основі заздалегідь визначених правил і профілів якості, налаштованих на сервері. Сканер збирає метрики коду, виконує статичний аналіз коду і генерує звіт про якість коду.

Сервер SonarQube отримує звіт про аналіз від сканера і обробляє дані. Він застосовує конфігуровані правила, пороги та профілі якості для оцінки якості коду та створення детальних звітів. Сервер зберігає результати аналізу, що дозволяє користувачам переглядати метрики, відслідковувати проблеми та контролювати загальний стан коду проекту.

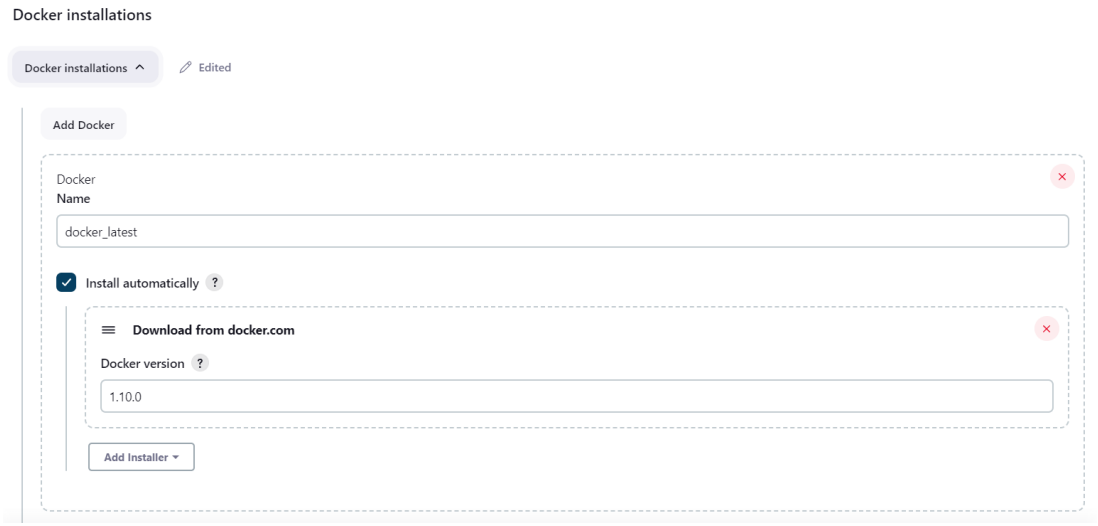


Рисунок 3.15 – налаштування клієнта Docker

Знаходимо пункт Docker installations, та по аналогії з минулим інструментом налаштовуємо встановлення. Для цього проєкту рекомендовано встановлювати саме версію 1.10.0, так як новіші версії викликають помилки при виконанні. Зберігаємо зроблені налаштування та повертаємось на минулу сторінку.

Переходимо до Manage Jenkins → Credentials → System → Global credentials. Тут конфігуруємо облікові записи та токени, які пізніше будемо використовувати в пайплайні та сервісах. Натиснемо Add Credentials та налаштуємо токен, що ми раніше створили в SonarQube (рисунок 3.16).

Рисунок 3.16 – налаштування токена SonarQube

Для токенів часто використовують тип Secret text, так як для токена не потрібен логін. Обираємо довільне зручне ім’я для ID та копіюємо наш токен в поле секрету.

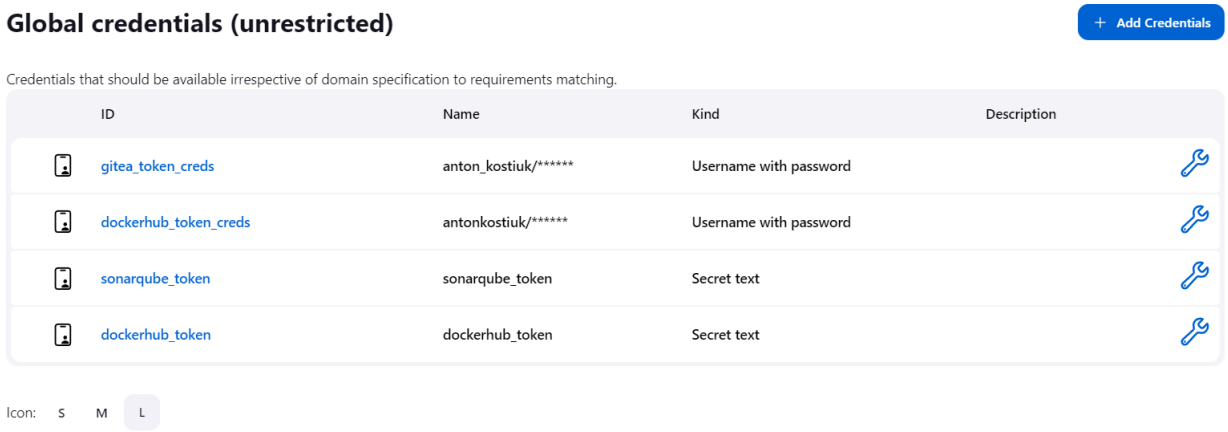


Рисунок 3.17 – необхідні облікові записи

По аналогії з токеном створимо облікові дані для Gitea та DockerHub. Відмінність у тому, що іноді все ж необхідно мати й ім’я облікового запису, тому для цих сервісів оберемо тип Username with password, а замість пароля будемо використовувати створені в сервісах токени.

Далі переходимо до налаштування системи: Manage Jenkins → System. У розділі SonarQube servers конфігуруємо з’єднання з нашим сервером SonarQube (рисунок 3.18).

Рисунок 3.18 – налаштування сервера SonarQube

Обираємо довільне ім’я та з випадаючого списку обираємо відповідний токен для автентифікації з’єднання з сервером. Для адреси сервера вказуємо http://sonarqube:9000. Ця адреса відрізняється від тої, за якою ми переходили в налаштуванні SonarQube. Це зумовлено тим, що за цією адресою сервер слухає запити всередині середовища Docker, а для нас він доступний через localhost, адже ми прокинули порт назовні.

На рисунку 3.19 відображено налаштування сервера Gitea. Воно аналогічне попередньому, за винятком опції Manage hooks та використання токена разом з ім’ям облікового запису.

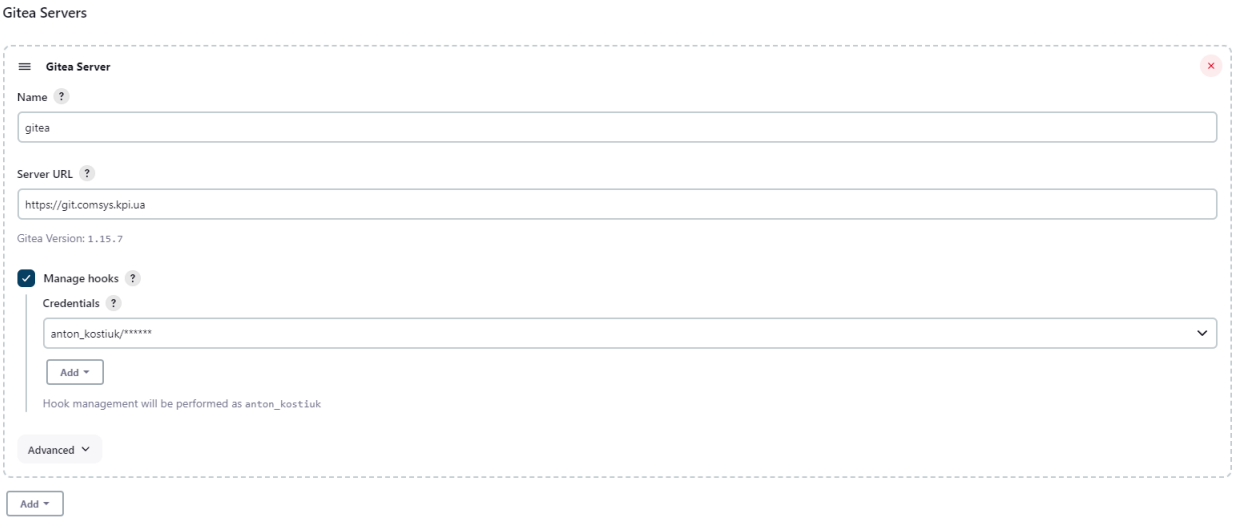


Рисунок 3.19 – налаштування сервера Gitea

Зберігаємо зміни та переходимо на головну сторінки для наступного кроку – створення, власне, пайплайну. В панелі зліва натискаємо New Item та обираємо тип проєкту Pipeline. Для Build Triggers обираємо опцію як показано на рисунку 3.20.

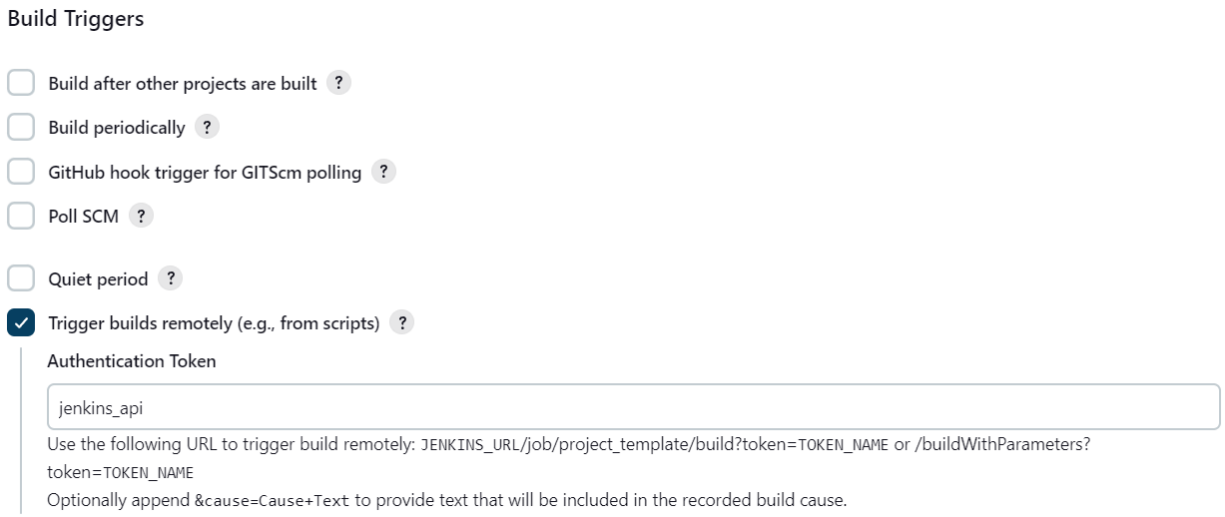


Рисунок 3.20 – налаштування тригерів пайплайну

Ці налаштування відповідають за те, яким чином буде приводитись в дію пайплайн. Найбільш ефективним буде варіант Trigger builds remotely, так як він не вимагає активних дій зі сторони Jenkins і спрацьовуватиме після запиту через REST API.

У вільному полі можна вказати довільний токен – своєрідний шар захисту для того, щоб роботу міг запустити лише той, хто має цей токен. Хоча він не є ультимативним рішенням, так як передається як чистий текст в запиті і схильний до перехоплення.

Далі переходимо до налаштування пайплайну, що відображене на рисунку 3.21. Обираємо варіант Pipeline script from SCM. Він означає, що скрипт зберігатиметься в репозиторії разом з проєктом та буде отриманий Jenkins звідти. Це корисна опція, так як є імплементацією підходу Pipeline-as-Code, що допомагає відслідковувати зміни що вносяться до пайплайну, робить простими відкати конфігурацій та інші переваги контролю версій коду.

Рисунок 3.21 – налаштування пайплайну

Обираємо систему контролю версій, в нашому випадку це Git, та додаємо дані про репозиторій Gitea та облікові дані. Нижче вказуємо гілку, яка буде відслідковуватись. Інші налаштування залишаємо за замовчуванням та зберігаємо.

Далі додамо та налаштуємо «хмарну» машину, що буде використовуватись для надання агентів Jenkins на вимогу. Для нашого проєкту, щоправда, це буде не зовсім хмарна машина. Ми скористаємось Docker, що і буде надавати нам необхідні машини що будуть використовуватись для виконання робіт там де це необхідно в пайплайні.

Перейдемо до Manage Jenkins → Clouds → New cloud. Вводимо довільне ім'я та обираємо тип Docker. Бачимо дві вкладки, що містять необхідні нам налаштування. У першій з них (рисунок 3.22) необхідно вказати шлях до змонтованого нами раніше сокета демона Jenkins та відмітити опцію Enabled нижче, щоб налаштування стали активними. Після чого перевіряємо зв'язок з хостом докера за допомогою кнопки Test Connection. У випадку успіху ми повинні побачити версію додатка та API Docker. Також нижче можна обмежити максимальну кількість контейнерів, що зможе підняти Jenkins.

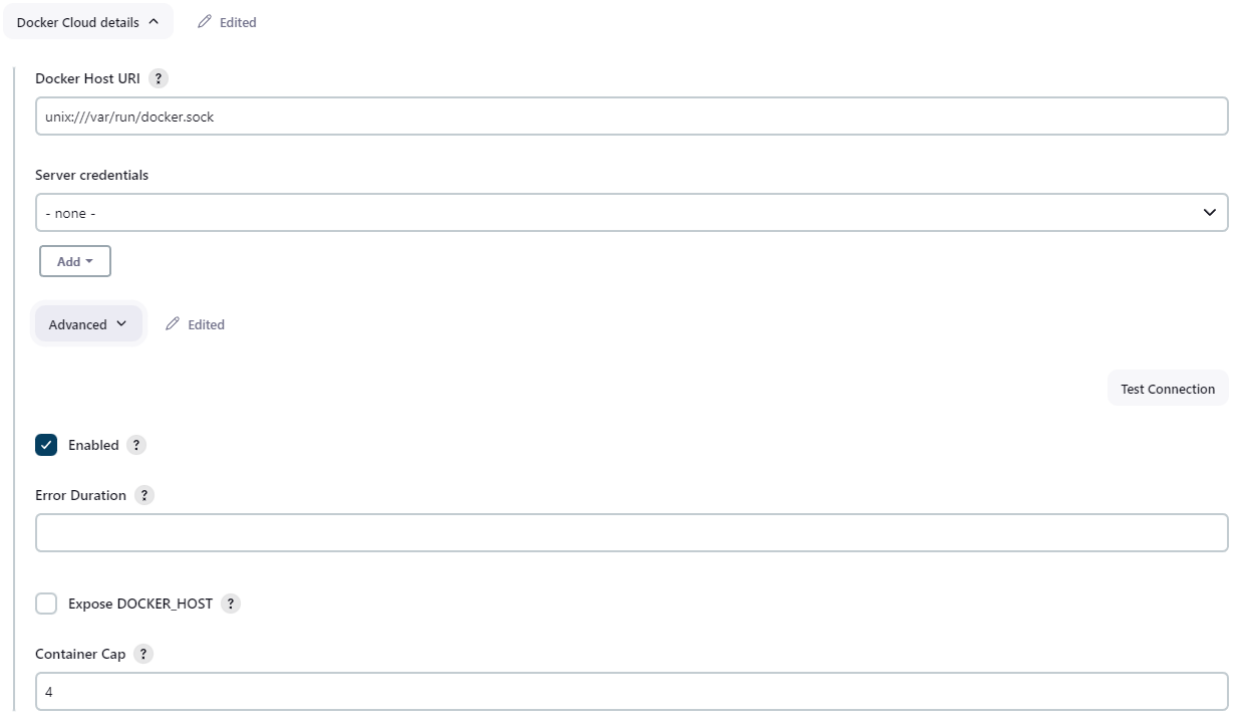


Рисунок 3.22 – налаштування Docker для хмари

Також конфігуруємо шаблон для агентів Docker. У полі Labels вказуємо назву за якою пізніше будемо викликати агента в пайплайні та активуємо налаштування. Як основу для контейнерів агентів використаємо образ

jenkins/agent:latest-jdk11. Він має необхідні залежності та додатки, щоб інтегруватись в систему Jenkins.

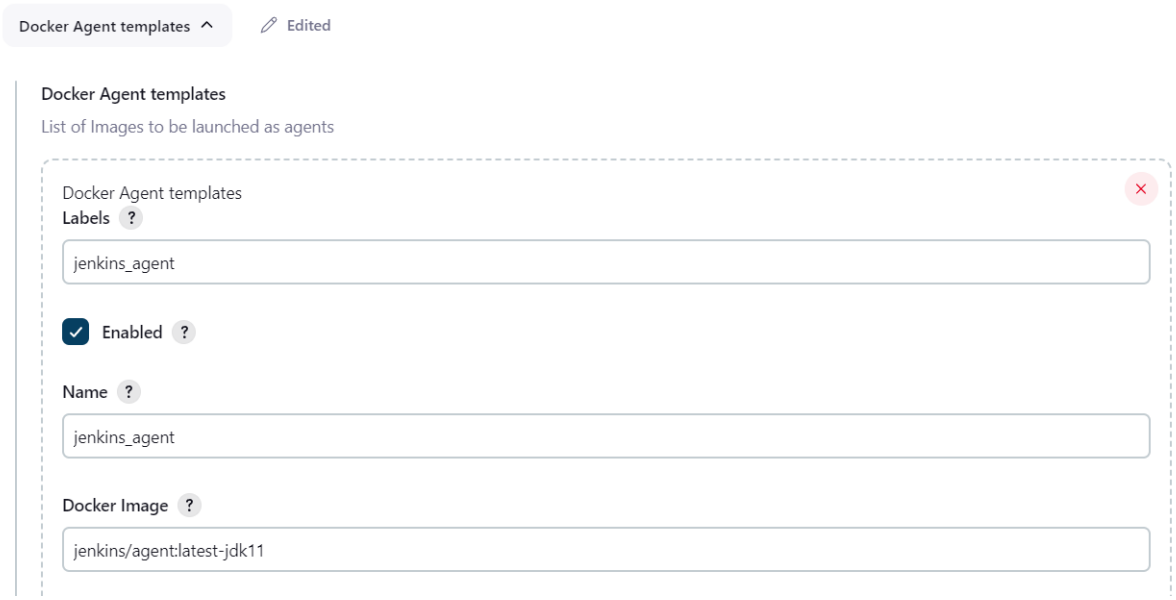


Рисунок 3.23 – налаштування шаблону агентів Docker

Як згадувалось раніше, під час налаштування мережі в docker-compose, нам необхідна була власна мережа для контейнерів. Саме її вказуємо в настройках контейнера в полі Networks. Також вказуємо наслідування томів з контейнера Jenkins в полі Volumes From для доступу до сокета.

Упевнюємось, що в налаштуванні Connect method обрано Attach Docker container та зберігаємо конфігурацію. На цьому налаштування сервера завершено.

Переходимо до написання Jenkinsfile. Groovy, скриптова мова програмування, що використовується для написання Jenkinsfile, підтримує як декларативний, так і імперативний стилі програмування. Декларативний синтаксис є більш структурованим і лаконічним способом визначення пайплайнів у Jenkins. Етапи, кроки і дії визначаються за допомогою заздалегідь визначеного набору ключових слів і синтаксису. Декларативні пайплайни спрямовані на створення вищого рівня абстракції, що полегшує визначення та управління потоком і структурою трубопроводу. З іншого боку, скриптовий синтаксис в Groovy дозволяє отримати більшу гнучкість і гранулярний

контроль над пайплайном. Кожен крок чітко визначений за допомогою скрипту Groovy. Скриптовані пайплайни надають більш просунуті можливості, що дозволяє інтегрувати складну логіку та динамічну поведінку. Виходячи з завдань, що стоять перед нашою системою, немає необхідності в написанні пайплайну в імперативному стилі, отже скористаємось стандартним декларативним синтаксисом.

Весь лістинг коду можна знайти в додатках, в той час як тут буде розглянуто ключові елементи нашого скрипту.

Перш за все у файлі визначається агент Jenkins, що виконуватиме роботу. Вказуємо лейбл агента, що ми створили раніше. Також налаштовуємо змінну середовища, де зберігатимемо облікові дані для логіна в DockerHub.

Jenkinsfile
```
pipeline {
    agent { label 'jenkins_agent' }
    environment {
        DOCKERHUB_CREDENTIALS = credentials('dockerhub_token_creds')
    }

```

Рисунок 3.24 – ініціалізація роботи

Після чекаута репозиторію визначимо запуск статичних тестів SonarQube. Для реалізації необхідно скористатись інструментом SonarQube Scanner, що ми налаштували раніше. Рядок def scannerHome = tool 'sonar_scanner'; дозволить отримати поточне місце знаходження інструмента в середовищі для подальшого виконання команди.

```
        stage('SonarQube Scan') {
            steps {
                script {
                    def scannerHome = tool 'sonar_scanner';
                    withSonarQubeEnv('sonarqube') {
                    sh "${scannerHome}/bin/sonar-scanner -Dsonar.projectKey=my_pipeline"
                    }
                }
            }
        }

```

Рисунок 3.25 – визначення сканера SonarQube

Останнім кроком пайплайну буде continuous delivery частина CI/CD підходу. Для цього контейнеризуємо наш додаток та завантажимо образ до DockerHub. Попередньо необхідно буде створити акаунт DockerHub та в настройках облікового запису, в розділі безпека створити токен доступу по аналогії з SonarQube. Цей токен тут використовується замість пароля. Також створимо репозиторій, куди будемо завантажувати наш образ.

```
34        stage('Build, Test and Push Docker Image') {
35            steps {
36                script {
37                    def dockerHome = tool 'docker_latest'
38                    env.PATH = "${dockerHome}/bin:${env.PATH}"
39                    sh 'docker --version'
40                    sh 'docker build -t my_pipeline_repo:latest .'
41                    sh 'docker login -u $DOCKERHUB_CREDENTIALS_USR -p $DOCKERHUB_CREDENTIALS_PSW -e kostia2013ua@gmail.com'
42                    sh 'docker push antonkostiuk/my_pipeline_repo:latest'
43                    sh 'docker logout'
44                }
45            }
46        }
```

Рисунок 3.26 – контейнеризація та доставка додатка

3.2.5 Gitea

Наостанок налаштуємо вебхук (webhook) та токен доступу Gitea. Як система контролю версій, вона повинна буде сповістити Jenkins про нову подію в репозиторії, що і запустить роботу.

Webhook - це механізм, який сприяє комунікації в реальному часі та взаємодії між додатками або системами через Інтернет. Це дозволяє одній програмі надсилати дані або запускати дії в іншій програмі на основі конкретних подій або оновлень. Процес використання webhook включає в себе налаштування URL-адреси в приймаючому додатку, який діє як webhook. Коли в програмі надсилання відбувається певна подія, вона генерує повідомлення. Потім надсилаюча програма надсилає запит HTTP POST до URL-адреси webhook, що містить відповідні дані про подію. Приймаюча програма, що розміщує webhook, слухає вхідні запити на наведеній кінцевій точці URL. Після отримання повідомлення, приймаюча програма може витягувати дані та виконувати різні дії на основі події, такі як оновлення даних, запуск робочих процесів або сповіщення користувачів.

Створення токену аналогічне попереднім та знаходиться у налаштуваннях користувача → додатки.

Налаштування вебхуків знаходяться в окремій вкладці у налаштуваннях репозиторію і мають наступний вигляд:

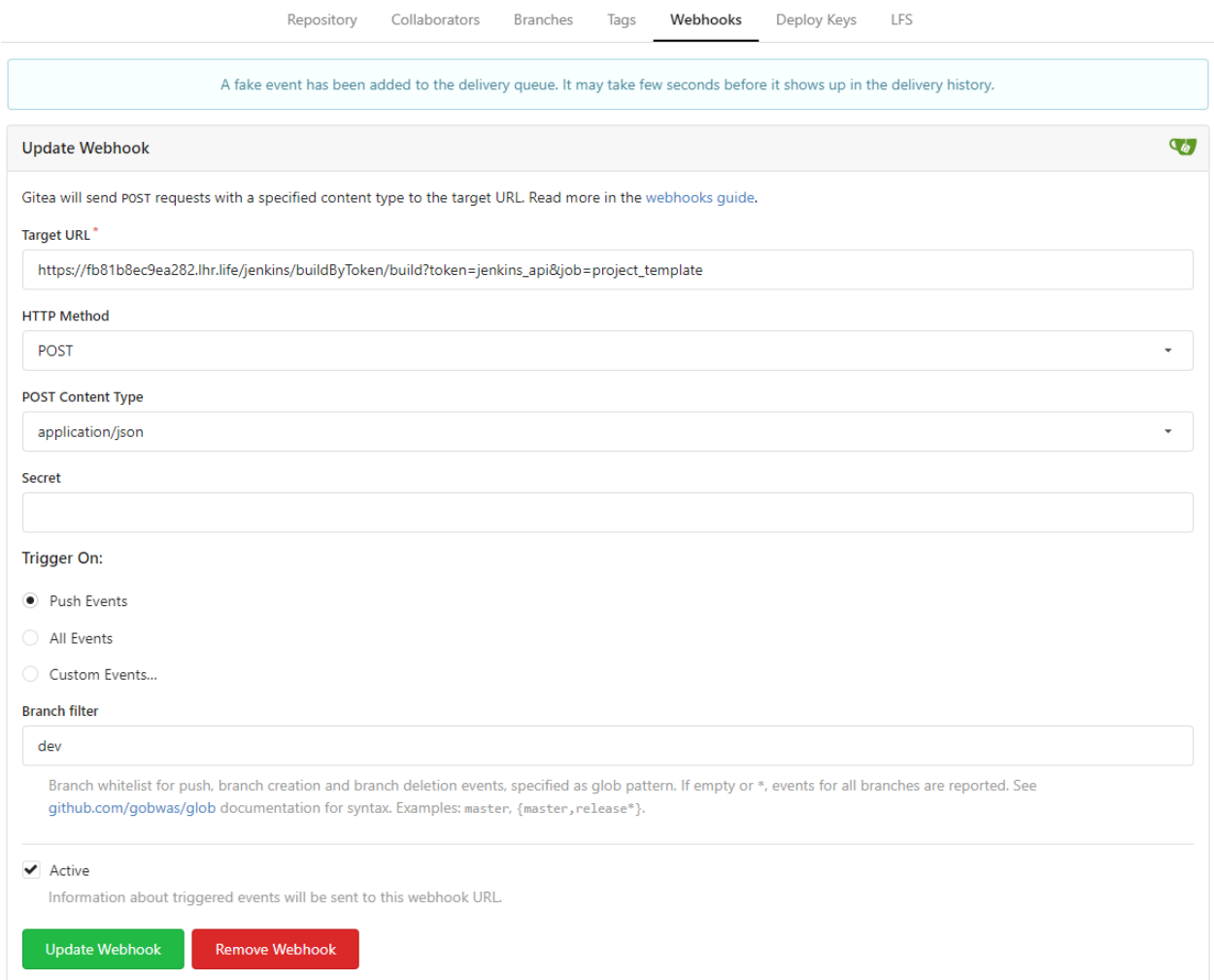


Рисунок 3.27 – налаштування webhook для Jenkins

Так як сервер Jenkins розташований на локальній машині, а сервер Gitea в зовнішній мережі, необхідно відкрити доступ до Jenkins у мережі інтернет. Як тимчасове рішення для демонстраційних цілей, було прийнято рішення скористатись безкоштовним сервісом, що створює безпечний SSH тунель між localhost:8080 та мережею інтернет. Враховуючи це, посилання буде починатись із рядка схожого на https://fb81b8ec9ea282.lhr.life, а постійною частиною URL буде:

/jenkins/buildByToken/build?token=jenkins_api&job=project_template.

Це посилання відрізняється від запропонованого Jenkins при налаштуванні пайплайну, так як створене відповідно до вимог плагіну Build Authorization Token Root Plugin. Він дозволяє обійти деякі проблеми, що виникають при авторизації вебхука при використанні стандартного посилання, проте все ще дозволяє використовувати токен, вказаний при налаштуванні пайплайну. Тим часом ключ job дозволяє вказати Jenkins якої саме роботи стосується цей вебхук.

Метод HTTP залишаємо на POST; тип контенту залишаємо за замовчуванням; налаштовуємо подію що запускатиме вебхук – в нашому випадку це Push до гілки dev.

ВИСНОВОК ДО РОЗДІЛУ 3

В даному розділі дипломної роботи була проведена реалізація CI/CD pipeline з використанням різних технологій, зокрема Jenkins, PostgreSQL, SonarQube, Docker, docker-compose та Gitea. Перш за все, була описана архітектура навчальної системи, включаючи систему управління версіями, сервер автоматизації та систему статичної перевірки коду. Далі, була детально розглянута реалізація навчальної системи, зокрема використання Docker для контейнеризації додатків, docker-compose для оркестрації контейнерів, SonarQube для статичного аналізу коду, Jenkins для автоматизації процесу CI/CD та Gitea для системи управління версіями. Кожна з цих технологій була інтегрована та налаштована з метою забезпечення ефективного та надійного процесу розробки, тестування та релізу програмного забезпечення. Реалізація CI/CD pipeline з використанням вищезгаданих технологій дозволяє здійснювати швидку та автоматизовану поставку програмних продуктів, забезпечує високу якість коду та покращує процес співпраці в команді розробників. Описані рішення та налаштування підтримують розширюваність, масштабованість та стабільність системи, що робить їх важливими компонентами в процесі розробки програмного забезпечення.

РОЗДІЛ 4
ТЕСТУВАННЯ ТА ОГЛЯД СИСТЕМИ

Розглянемо сценарій роботи з розробленою системою. Для початку необхідно розгорнути контейнери з серверами на зручному для нас хості, для прикладу локально, за допомогою команди docker-compose up -d.

Очікуємо поки усі контейнери не будуть готові до роботи. В ході роботи над проєктом, що знаходиться в репозиторії Gitea до файлів будуть внесені зміни. Для прикладу, змінимо текст у повідомленні нашого додатку і збережемо зміни. Вони відобразяться у системі контролю версій, де ми виконаємо коміт та пуш коду до репозиторію Gitea у гілку dev:

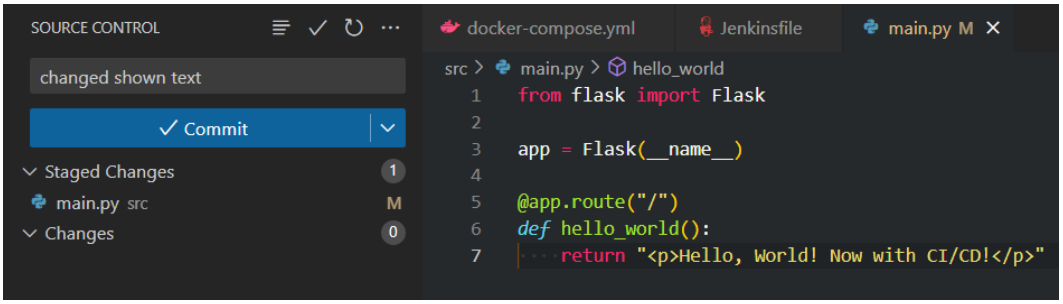


Рисунок 4.1 – внесення змін до коду

Одразу ж після виконання пушу спрацює вебхук та запуститься робота в Jenkins, що відображено на рисунку 4.2.

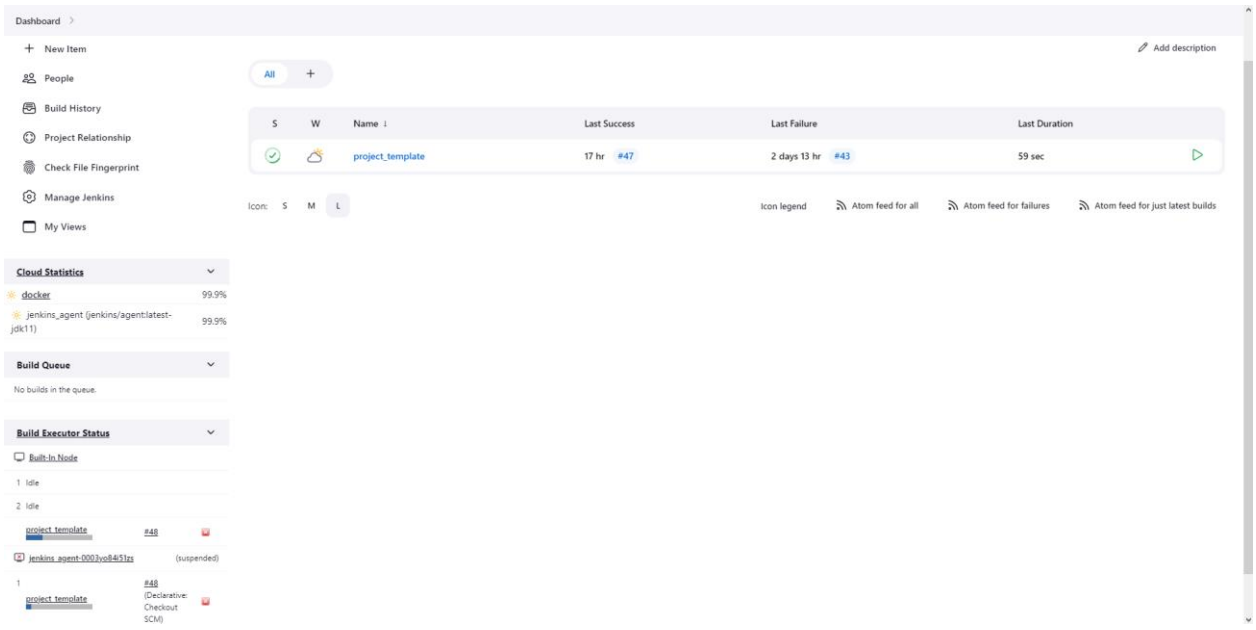


Рисунок 4.2 – автоматичний запуск роботи в Jenkins

За ходом виконання роботи можна слідкувати також перейшовши на сторінку роботи – рисунок 4.3. Тут відображатиметься виконання кожного із етапів, час що на них затрачається та успішність, тощо. Також, натиснувши на номер роботи, можна перейти до виводу терміналу з роботою.

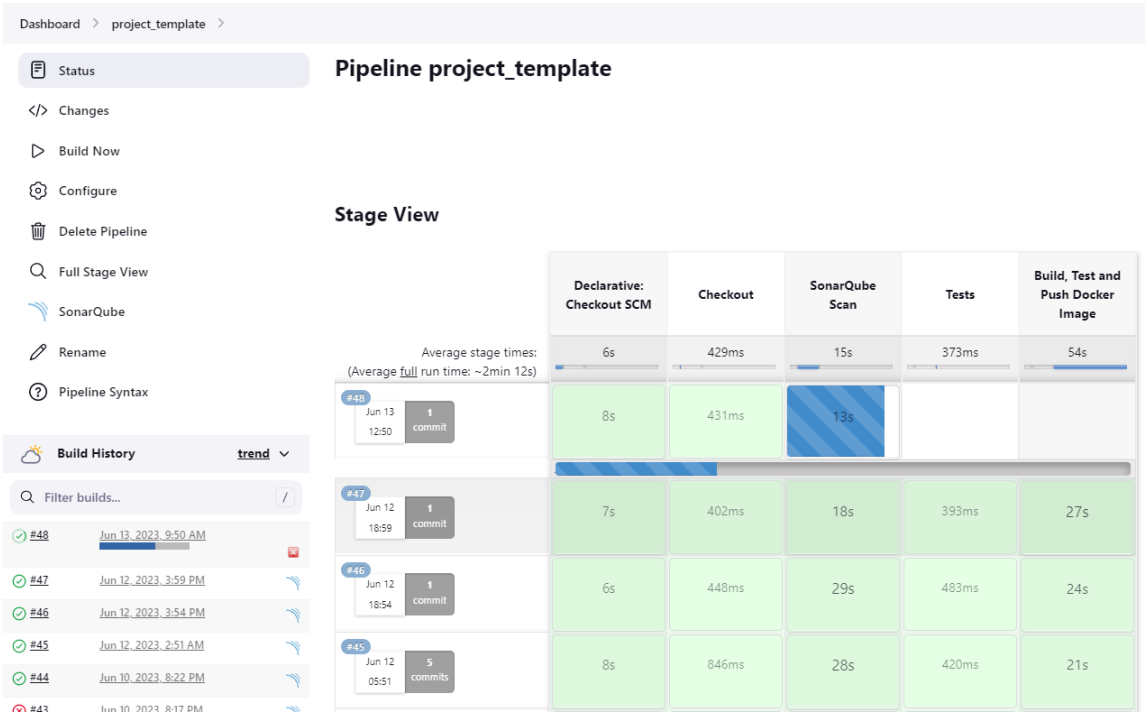


Рисунок 4.3 – сторінка огляду роботи

Після успішного завершення роботи можемо відвідати сторінку SonarQube та переглянути результати сканування проєкту (рисунок 4.4).

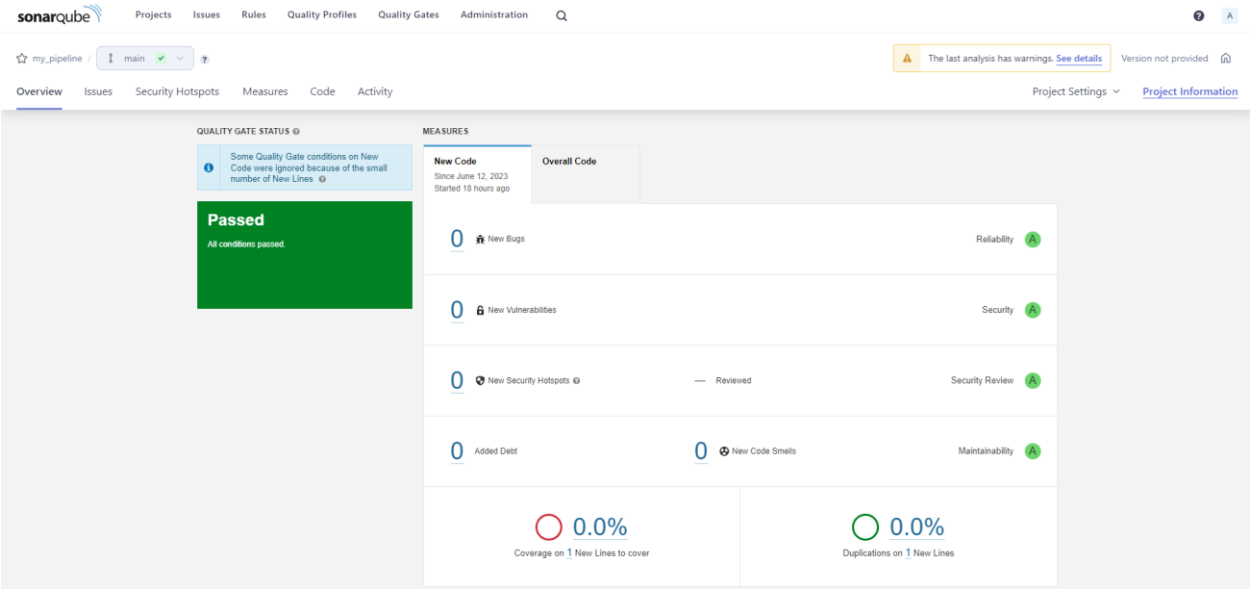


Рисунок 4.4 – результати останнього сканування SonarQube

Далі, перейшовши на сторінку репозиторію DockerHub, бачимо, що успішно відбулась доставка зібраного автоматично контейнера з нашим додатком:

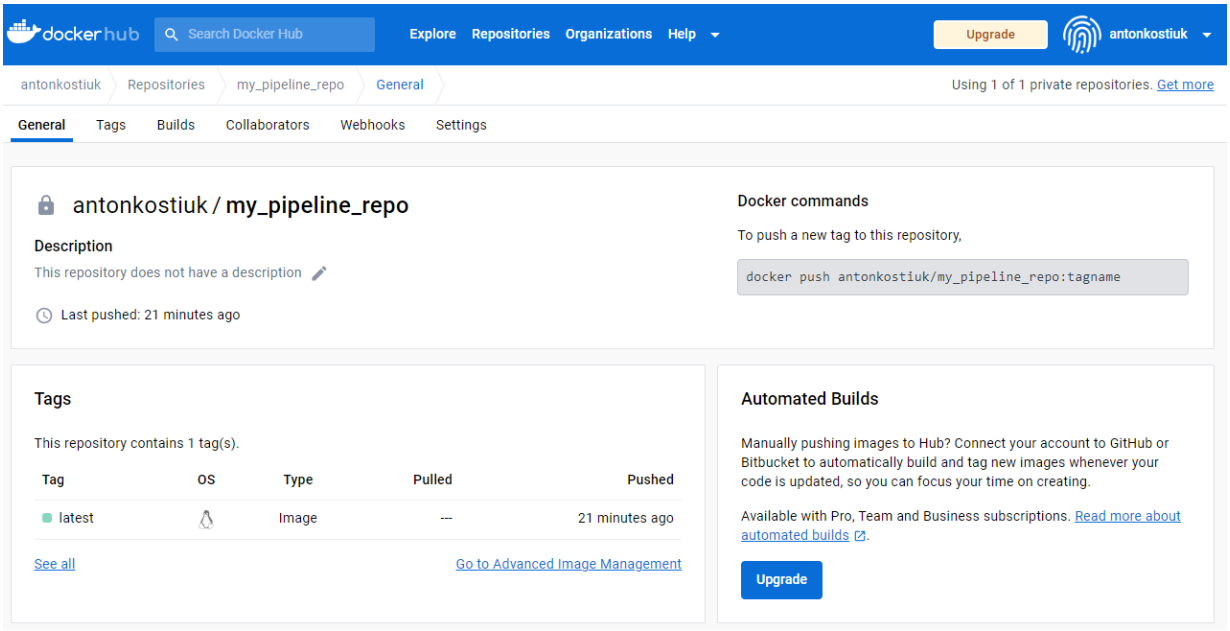


Рисунок 4.5 – сторінка репозиторію DockerHub

Таким чином перевірено коректну роботу створеної навчальної системи та її складових.

ВИСНОВОК ДО РОЗДІЛУ 4

В результаті проведеного тестування та огляду системи було встановлено, що вона працює згідно з поставленим завданням та успішно виконує всі свої функції. Тестування виявило відсутність помилок, проблем та недоліків у роботі системи, а також підтвердило її високу якість та надійність. Огляд системи також підтвердив відповідність її архітектури та компонентів вимогам проекту, а також наголосив на зручності їх використання та можливостях для подальшого розширення та підтримки. Розроблена система стане чудовою платформою для навчання, освоєння сучасних практик та методологій розробки ПЗ. Враховуючи успішне тестування та огляд системи, можна зробити висновок, що розроблена система відповідає всім вимогам, працездатна та готова до експлуатації.

ЗАГАЛЬНИЙ ВИСНОВОК

Дипломна робота присвячена розробці навчальної системи для сумісного виконання ІТ проєктів на основі методології CI/CD. Дана система дозволить оптимізувати процеси розробки ПЗ в командах, а також надасть можливість для вивчення технологій та практик, що є актуальними на даний момент, а саме практики DevOps.

В першому розділі було проведено аналіз традиційних методологій розробки ПЗ та недоліки, що супроводжують їх використання. Було оглянуто переваги методології DevOps та її практик, які дозволяють забезпечити ефективну співпрацю між командами розробників та операторів.

У другому розділі проведений детальний аналіз інструментів, які використовуються на різних етапах розробки ПЗ. Для системи контролю версій було обрано Git та сервіс Gitea. Для автоматизації CI/CD було вибрано Jenkins, а для статичного аналізу коду - SonarQube. Кожен з цих інструментів має свої переваги та відкритий код, що підтримує принципи відкритості та гнучкості.

Третій розділ описує архітектуру навчальної системи та її компоненти, такі як система управління версіями, сервер автоматизації та система статичної перевірки коду. Було розглянуто використання Docker для контейнеризації додатків, docker-compose для оркестрації контейнерів. Реалізація компонентів забезпечує ефективну співпрацю у командах, автоматизацію процесу збірки, тестування та розгортання програмного забезпечення. Розглянуто використання веб-інтерфейсу для користувачів, що дозволяє взаємодіяти з системою, переглядати історію версій, відстежувати статуси проєктів та отримувати повідомлення про стан процесу CI/CD.

В четвертому розділі було проведено тестування розробленої навчальної системи. Розглянуто сценарій використання системи, підтверджено відповідність системи поставленим вимогам.

Отже, навчальна система, розроблена в рамках дипломної роботи, є ефективним інструментом для командної роботи, що також дозволяє результативно навчатись використанню широким спектром інструментів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Buchanan I. History of DevOps [Електронний ресурс] / Ian Buchanan – Режим доступу до ресурсу: https://www.atlassian.com/devops/what-is-devops/history-of-devops.
2. Raznik J. Pros and Cons of Website Builders [Електронний ресурс] / Julia Raznik. – 2022. – Режим доступу до ресурсу: https://verpex.com/blog/website-tips/advantages-and-disadvantages-using-website-builders.
3. What is Platform-as-a-Service (PaaS)? [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/topics/paas.
4. About Git [Електронний ресурс] – Режим доступу до ресурсу: https://git-scm.com/about/branching-and-merging.
5. Apache® Subversion® [Електронний ресурс] – Режим доступу до ресурсу: https://subversion.apache.org/.
6. Mercurial [Електронний ресурс] – Режим доступу до ресурсу: https://www.mercurial-scm.org/.
7. Metla R. What is Bitbucket? [Електронний ресурс] / Roja Metla – Режим доступу до ресурсу: https://www.educba.com/what-is-bitbucket/.
8. Bitbucket Pipelines [Електронний ресурс] – Режим доступу до ресурсу: https://bitbucket.org/product/features/pipelines.
9. About Jenkins [Електронний ресурс] – Режим доступу до ресурсу: https://www.jenkins.io/.
10. SonarQube Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.sonarqube.org/latest/.
11. Docker vs Virtual Machines (VMs) : A Practical Guide to Docker Containers and VMs [Електронний ресурс] – Режим доступу до ресурсу: https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms.

ДОДАТОК 1

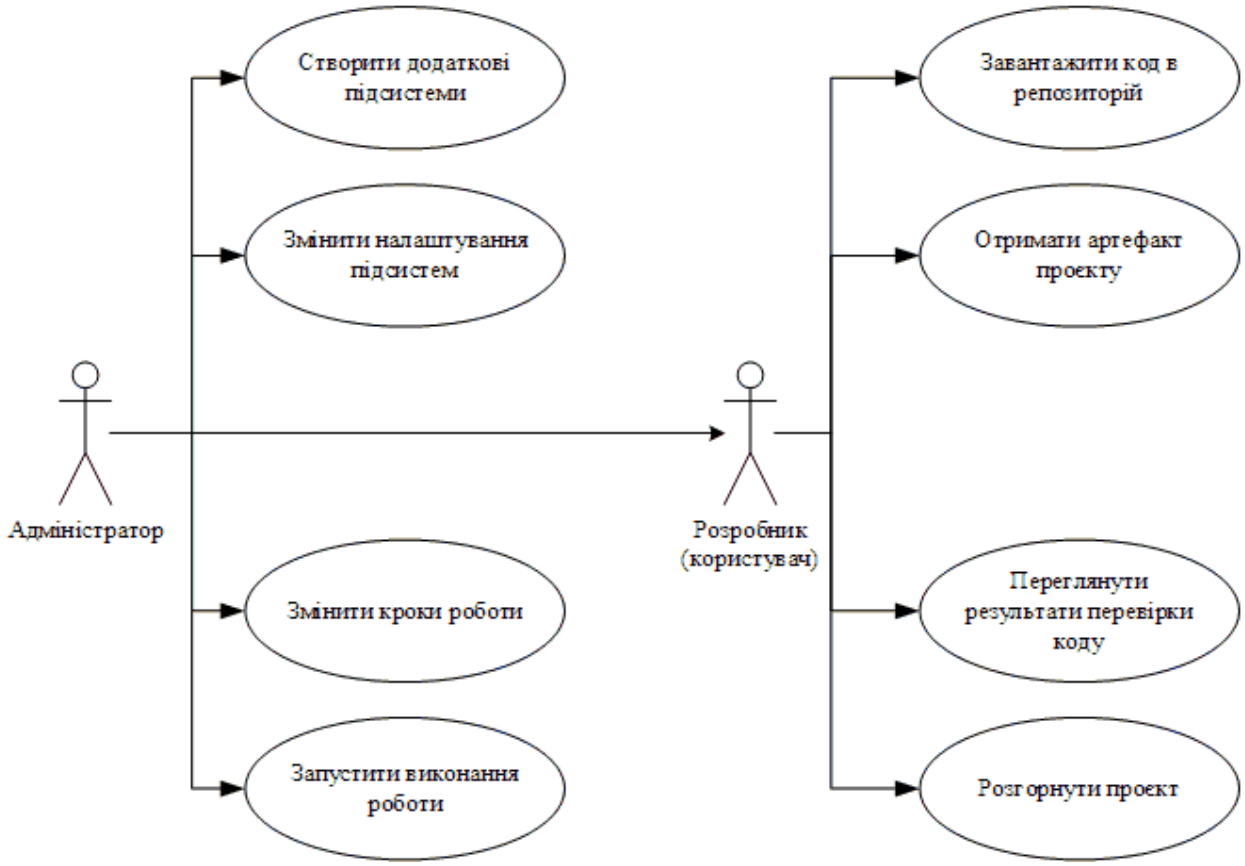
Навчальна система для сумісного виконання ІТ проєктів на основі методології CI/CD

Варіанти використання системи (структурна схема)

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.004 Д1	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		Костюк А.В.			Навчальна система для сумісного виконання ІТ проєктів на основі методології CI/CD Варіанти використання системи (Структурна схема)	Літ. / Аркуш / Аркушів: 1 / 1
Перевірив		Гайдай А.Р.				
Реценз.						КПІ ім. Ігоря Сікорського, ФІОТ, ІО-92
Н. Контр.		Виноградов Ю.М.				
Затвердив						

ДОДАТОК 2

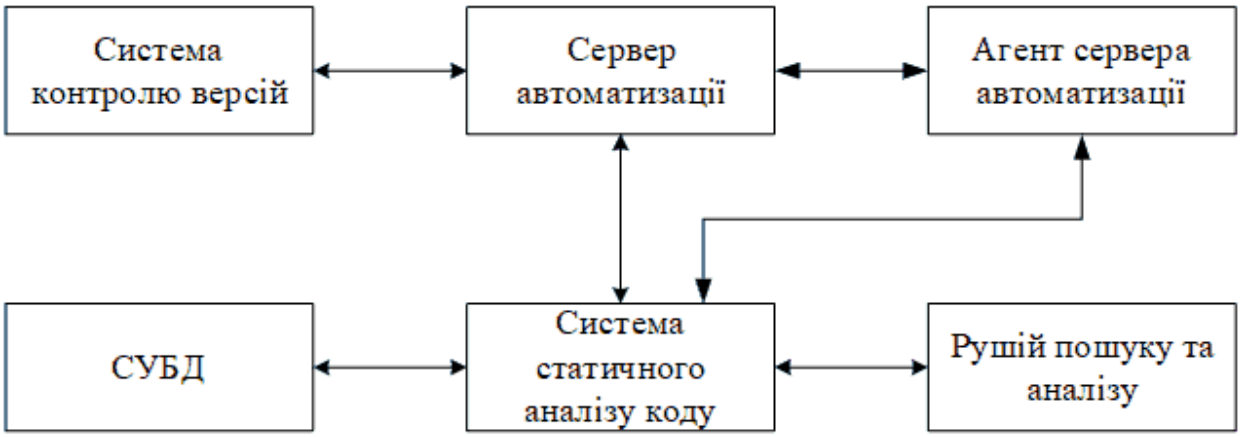
Навчальна система для сумісного виконання ІТ проєктів на основі методології CI/CD

Взаємодія підсистем (функціональна схема)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.005 Д2		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Костюк А.В.			Навчальна система для сумісного виконання ІТ проєктів на основі методології CI/CD Взаємодія підсистем (Функціональна схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Гайдай А.Р.					
Реценз.						КПІ ім. Ігоря Сікорського, ФІОТ, ІО-92	
Н. Контр.		Виноградов Ю.М.					
Затвердив							

ДОДАТОК З

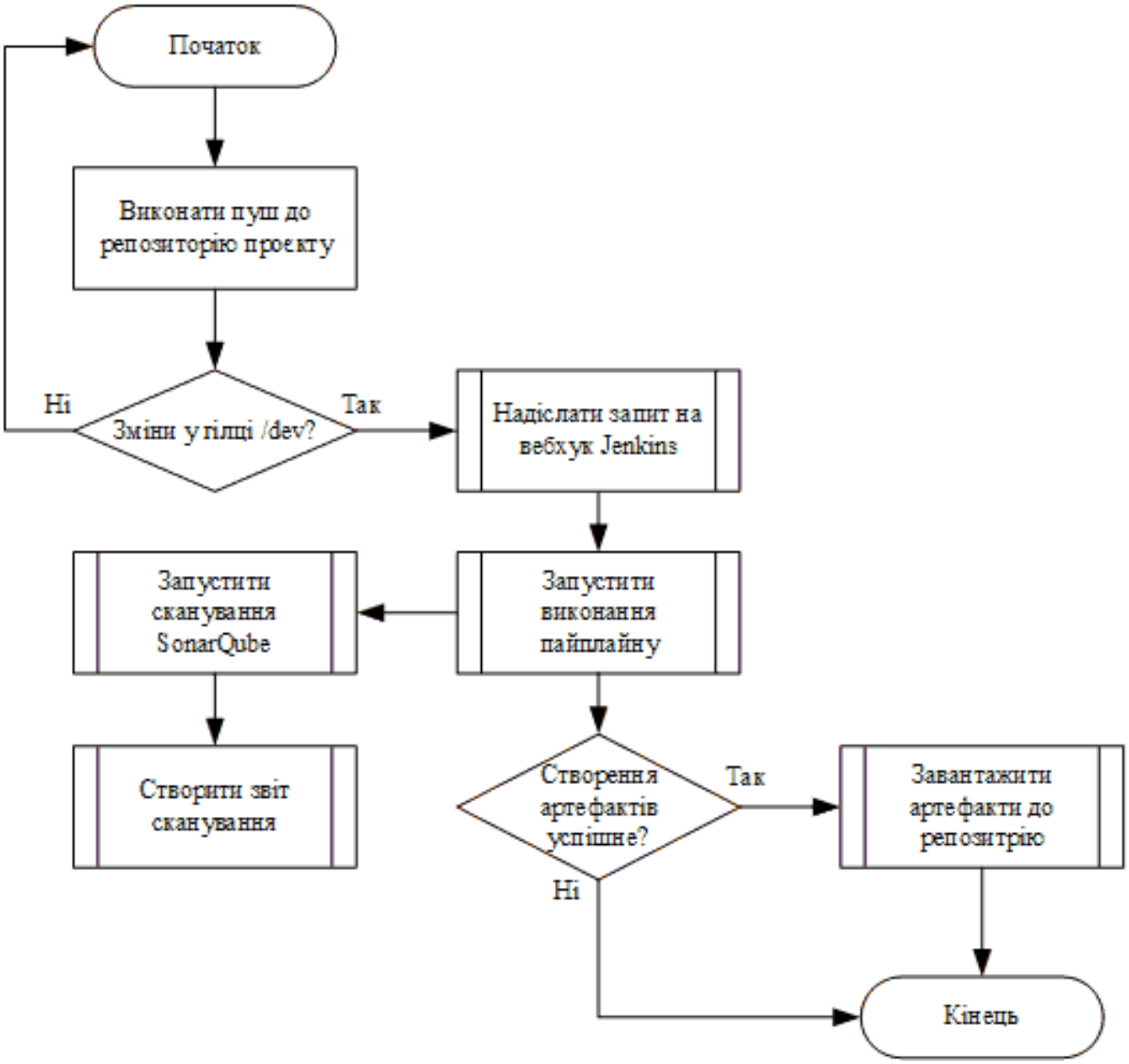
Навчальна система для сумісного виконання ІТ проєктів на основі методології CI/CD

Алгоритм роботи системи (принципова схема)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.006 Д3	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		Костюк А.В.			Навчальна система для сумісного виконання ІТ проєктів на основі методології CI/CD Алгоритм роботи системи (Принципова схема)	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Гайдай А.Р.				
Реценз.						КПІ ім. Ігоря Сікорського, ФІОТ, ІО-92
Н. Контр.		Виноградов Ю.М				
Затвердив						

ДОДАТОК 4

Навчальна система для сумісного виконання ІТ проєктів на основі методології CI/CD

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 1

Київ 2023 р

```
version: "3.8"

services:
  jenkins:
      image: jenkins/jenkins
      container_name: jenkins_container
      restart: always
      privileged: true
      user: root
      networks:
          - jenkins-network
      ports:
          - "8080:8080"
          - "50000:50000"
      volumes:
          - '/var/run/docker.sock:/var/run/docker.sock'
          - 'jenkins-data:/var/jenkins_home'
      environment:
          - "JENKINS_OPTS=--prefix=/jenkins"

  sonarqube:
      image: sonarqube
      container_name: sonar_container
      restart: always
      privileged: true
      depends_on:
        postgresdb:
          condition: service_healthy
      environment:
          - SONAR_SEARCH_JAVAADDITIONALOPTS=-Dnode.store.allow_mmap=false
          -
SONAR_JDBC_URL=jdbc:postgresql://postgresdb:5432/sonarqube?currentSchema=public
          - SONAR_JDBC_USERNAME=sonar
          - SONAR_JDBC_PASSWORD=sonar
      networks:
          - jenkins-network
      ports:
          - "9000:9000"
      volumes:
```

```
            - 'sonar-data:/opt/sonarqube/data'
            - 'sonar-data:/opt/sonarqube/extensions'
            - 'sonar-data:/opt/sonarqube/lib/bundled-plugins'

    postgresdb:
        image: postgres:12
        container_name: postgres_container
        restart: always
        environment:
            - POSTGRES_USER=sonar
            - POSTGRES_PASSWORD=sonar
            - POSTGRES_DB=sonarqube
        networks:
            - jenkins-network
        volumes:
            - postgres-data:/var/lib/postgresql/data
        healthcheck:
          test: ["CMD", "bash", "-c", "echo hi > /dev/tcp/localhost/5432"]
          interval: 10s
          timeout: 10s
          retries: 5

networks:
  jenkins-network:
    name: jenkins
    driver: bridge

volumes:
    jenkins-data:
    sonar-data:
    postgres-data:


pipeline {
    agent { label 'jenkins_agent' }
    environment {
        DOCKERHUB_CREDENTIALS = credentials('dockerhub_token_creds')
    }

    stages {
        stage('Checkout') {
            steps {
                git branch: 'dev',
```

```
                credentialsId: 'gitea_token_creds',
                url: 'https://git.comsys.kpi.ua/anton_kostiuk/my_pipeline_repo'
            }
        }

        stage('SonarQube Scan') {
            steps {
                script {
                    def scannerHome = tool 'sonar_scanner';
                    withSonarQubeEnv('sonarqube') {
                    sh "${scannerHome}/bin/sonar-scanner -Dsonar.projectKey=my_pipeline"
                    }
                }
            }
        }

        stage('Tests') {
            steps {
                // here would be the tests
                sh 'echo "test success"'
            }
        }

        stage('Build, Test and Push Docker Image') {
            steps {
                script {
                    def dockerHome = tool 'docker_latest'
                    env.PATH = "${dockerHome}/bin:${env.PATH}"
                    sh 'docker --version'
                    sh 'docker build -t my_pipeline_repo:latest .'
                    sh 'docker login -u $DOCKERHUB_CREDENTIALS_USR -p
$DOCKERHUB_CREDENTIALS_PSW -e kostia2013ua@gmail.com'
                    sh 'docker push antonkostiuk/my_pipeline_repo:latest'
                    sh 'docker logout'
                }
            }
        }
    }
}
```