Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет *Інфокомунікацій*
(повна назва)

Кафедра *Інформаційно – мережна інженерія*
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Аналіз різних підходів та інструментів для безперервної інтеграції та розгортання коду.
(тема)

Виконав: студент *2* курсу, групи *ІМІм-18-1*
Кварацхелія Т.Е.
(прізвище, ініціали)

Спеціальність *172 "Телекомунікації та радіотехніка"*
(код і повна назва спеціальності)

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

Освітня програма *Інформаційно-мережн інженерія*
(повна назва освітньої програма)

Керівник *доц., к.т.н Костромицький А. І.*
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ _____
(підпис) (прізвище, ініціали)

2019 р.

Не містить відомостей, заборонених до відкритого публікування

Студент _____

Керівник_____

Харківський національний університет радіоелектроніки

Факультет *Інфокомунікацій*

Кафедра *Інформаційно – мережна інженерія*

Рівень вищої освіти другий (магістерський)

Спеціальність *172 "Телекомунікації та радіотехніка"*
(код і повна назва)

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

Освітня програма *Інформаційно-мережна інеженерія*
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2019 р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові *Кварацхелія Тимуру Елгуджайовичу*
(прізвище, ім'я, по батькові)

1. Тема роботи *Аналіз різних підходів та інструментів для безперервної інтеграції та розгортання коду.*

затверджена наказом по університету від *31 10* 2019 р. № *1609 Ст*
2. Термін подання студентом роботи до екзаменаційної комісії ____ _____ 2019 р.
3. Вихідні дані до роботи
Огляд основних підходів, методів та існуючих інструментів неперервної інтеграції коду та його розгортання, порівняльний аналіз провідних інструментів, побудова проекту на основі проведеного аналізу.

4. Перелік питань, що потрібно опрацювати в роботі
Вступ
1. Етапи та методології розробки програмного забезпечення
2. Безперервна інтеграція, розгортання та тестування
3. Аналіз існуючих інструментів побудови CI/CD/CT
4. Проектування комерційного проекту на основі обраного CI/CD/CT інструменту
Висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (тощо)
Слайди у форматі PowerPoint(вступ, завдання дипломної роботи, етапи розробки програмного забезпечення, методології розробки програмного забезпечення, безперервна інтеграція (CI), доставка, розгортання(CD) та тестування (CT), аналіз існуючих інструментів побудови CI/CD/CT, результати аналізу, практичне проектування комерційного проекту на основі аналізу, результати проектування, висновки)

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	*Ознайомлення із завданням. Уточнення ТЗ.*	*30.10.19*	
2	*Підбір літератури за темою роботи.*	*02.11–03.11.19*	
3	*Виконання розділу 1*	*03.11–10.11.19*	
4	*Виконання розділу 2*	*10.11–17.11.19*	
5	*Виконання розділу 3*	*17.11–25.11.19*	
6	*Виконання розділу 4*	*25.11–10.12.19*	
7	*Оформлення презентаційного матеріалу,*	*10.12–18.12.19*	
	підготовка до захисту у ДЕК		

Дата видачі завдання *30 жовтня 2019 р.*

Студент _____
(підпис)

Керівник роботи _____ *доц., к.т.н Костромицький А.І.*
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: – 99 с., 44 рис., 12 джерел, 1 додаток.

ІНТЕГРАЦІЯ, РОЗГОРТАННЯ, ПРОГРАМНИЙ КОД, ІНСТРУМЕНТ, UNIX, ТЕСТУВАННЯ КОДУ

Об'єкт дослідження – системна інженерія.

Мета роботи: аналіз методів та провідних інструментів безперервного інтегрування, тестування, розгортання та доставки коду, порівняння на основі обраних критеріїв та побудова працюючої інфраструктури опираючись на проведений аналіз, для поліпшення ефективності розробки коду.

Передбачувані наукові результати: порівняльний аналіз інструментів та демонстрація збільшення ефективності розробки коду шляхом застосування сучасних методів CI/CD/CT.

THE ABSTRACT

Explanatory note: – 99 p., 44 fig., 12 sources, 1 app.

INTEGRATION, DEPLOYMENT, SOFTWARE CODE, A TOOL, UNIX, CODE TESTING

Target of research – systems engineering.

Objective: analysis of methods and tools for uninterrupted integration, testing, delivery of code, selection on the basis of the criteria and the basis of infrastructures, rely on the conduct of analysis, for business.

Predictable scientific results: the next analysis and tools and demonstration of the effectiveness of working with the code in order to get the best of CI/CD/CT methods.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API (Application Programming Interface) – програмний інтерфейс додатку
CD (Continuous Deployment or Delivery) – безперервне розгортання/доставка
CI (Continuous Integration) – неперервна інтеграція
CT (Continuous Testing) – безперервне тестування
IAC (Infrastructure as a Code) – інфраструктура як код
QA – (Quality Assurance) – забезпечення якості
ОС – операційна система
ПЗ – програмне забезпечення

ВСТУП

Протягом останніх років майже всі сфери бізнесу, які так чи інакше пов’язані з розробкою програмного забезпечення зазнали значних змін. З розвитком сучасних технологій, збільшенням обсягу даних та можливостей їх швидкої обробки, компанії, які розробляють додатки, прагнуть стати більш спритними, для того щоб мати можливість впроваджувати інновації та швидше реагувати на зміни. Для цього потрібні зміни у підходах до розробки, тестування та релізу нового коду.

У зв’язку з цим, процес розробки програмного забезпечення зазнав значних змін. У зв’язку з тим, що бізнес потребує більшої гнучкості, першість у методологіях розробки посіли такі методології як Agile та DevOps, які передбачають автоматизацію та використання безперервної інтеграції, розгортання, доставки та тестування.

Безперервна інтеграція (далі за текстом CI) – це філософія кодування та набір практик, які спонукають команди розробників впроваджувати невеликі зміни та часто перевіряти код у сховищах управління версіями. Оскільки більшість сучасних додатків потребують розробки коду на різних платформах та інструментах, команді потрібен механізм інтеграції та затвердження його змін.

Технічна мета CI полягає у встановленні послідовного та автоматизованого способу складання, упаковки та тестування додатків. При послідовності в процесі інтеграції команди, швидше за все, частіше вносять зміни коду, що призводить до кращої співпраці та якості програмного забезпечення.

Так як процес випуску програмного забезпечення розвивався з часом, ручний процес переміщення коду з однієї машини на іншу і процес перевірки, чи працює він, як очікувалося, був схильним до помилок і важким процесом. Безперервне розгортання (далі за текстом CD) має призначення автоматизувати цей процес. Технічно це випуск програмного забезпечення, який використовує автоматичне тестування для перевірки, чи є зміни в кодовій базі правильними та стабільними для негайного автономного розгортання у виробничому середовищі.

Використання таких підходів у купі з безперервним тестуванням (далі за текстом CT) дозволило інженерним організаціям зосередитись на основних проблемах бізнесу замість інфраструктури та ручного тестування.

1 ЕТАПИ ТА МЕТОДОЛОГІЇ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Етапи життєвого циклу сучасного програмного забезпечення

Життєвий цикл програмного забезпечення (ПЗ) являє собою модель його створення і використання. Дана модель відображає його різні стани, починаючи з моменту виникнення потреби в даному ПЗ і закінчуючи моментом його повного виходу в ужиток для усіх користувачів. Моделі розрізняються способом взаємозв'язку етапів життєвого циклу, але кожен з цих етапів в тому чи іншому вигляді присутня в кожній моделі. Нижче я послідовно розповім про ці етапи.

1.1.1 Стратегія

Визначення стратегії припускає обстеження системи. Основне завдання обстеження – це оцінка реального обсягу проекту, його цілей і завдань, а також отримання визначень сутностей і функцій на високому рівні. На цьому етапі залучаються висококваліфіковані бізнес-аналітики, які мають постійний доступ до керівництва фірми; також передбачається тісна взаємодія з основними користувачами системи і бізнес-експертами. Основне завдання такої взаємодії – отримати якомога повнішу інформацію про систему, однозначно зрозуміти вимоги замовника і передати цю інформацію в формалізованому вигляді системним аналітикам. Як правило, інформація про систему може бути отримана на підставі ряду бесід (або семінарів) з керівництвом, експертами і користувачами.

Підсумком етапу визначення стратегії стає документ, де чітко сформульовано наступне: що саме належить замовнику, якщо він погодиться фінансувати проект; коли він зможе отримати готовий продукт (графік виконання робіт); о котрій це йому обійдеться (графік фінансування етапів робіт для великих проектів). У документі повинні бути відображені не тільки витрати, але і вигода, наприклад час окупності проекту, очікуваний економічний ефект (якщо його вдається оцінити).

Визначення стратегії – це принципова відповідь на запитання на кшталт: «Чи будемо ми робити цей проект за такі-то гроші чи ні?» Або «Чи робимо ми в принципі цей проект з даними розробником?» Іншими словами, в результаті

визначення стратегії розробник і замовник приймають рішення про продовження робіт на певних умовах з певними обов'язками сторін.

Слід виділити набір фактів, які повинні бути обов'язково відображені в заключному документі після проведення обстеження і аналізу діяльності підприємства:

- обмеження, ризики, критичні фактори, що впливають на проект;
- сукупність умов експлуатації майбутньої системи: архітектура, апаратні і програмні ресурси, зовнішні умови її функціонування; склад виконавців і робіт, що забезпечують функціонування системи;
- критичні терміни завершення етапів, форма здачі робіт, захист комерційної інформації;
- опис виконуваних системою функцій;
- інтерфейси і розподіл функцій між людиною і системою;
- вимоги до програмних і інформаційних компонентів ПЗ;
- наявність потенційного розвитку системи в майбутньому;
- те, що не буде реалізовано в рамках проекту.

Даний етап життєвого циклу ПЗ може бути представлений в моделі тільки один раз, особливо якщо модель має циклічну структуру, однак це не означає, що в циклічних моделях стратегічне планування проводиться раз і назавжди. Циклічні моделі призначені для вирішення проблем ПЗ, яке розвивається за часом; аналіз поточного стану ПЗ і підприємства, його використовує, проводиться на етапі аналізу. Таким чином, в циклічних моделях етапи визначення стратегії і аналізу як би склеюються, а ймовірність їх поділу існує лише на найпершому витку, коли керівництво підприємства приймає принципове рішення про старт проекту. В цілому етап аналізу присвячений розробці документа рівня керівництва підприємства.

1.1.2 Аналіз

Етап аналізу передбачає докладне дослідження бізнес-процесів (функцій, визначених на попередньому етапі) і інформації, необхідної для їх виконання (сутностей, їх атрибутів і зв'язків (відносин)). Цей етап дає інформаційну модель, а наступний за ним етап проектування – модель даних.

Вся інформація про систему, зібрана на етапі визначення стратегії, формалізується і уточнюється на етапі аналізу. Особливу увагу слід приділити повноті переданої інформації, аналізу інформації на несуперечність, а також пошуку невикористаної або дублюється інформації. Як правило, замовник спочатку формує вимоги не до системи в цілому, а до окремих її компонентів. І в цьому конкретному випадку циклічні моделі життєвого циклу ПЗ мають перевагу, оскільки з плином часу з великою ймовірністю буде потрібно повторний аналіз, так як у замовника часто апетит приходить під час їжі. На цьому ж етапі визначаються необхідні компоненти плану тестування.

Аналітики збирають і фіксують інформацію в двох взаємопов'язаних формах:

– функції – інформація про події та процеси, які відбуваються в бізнесі;
– сутності – інформація про речі, які мають значення для організації і про які що-небудь відомо.

Раніше двома класичними результатами аналізу були: ієрархія функцій, яка розбиває процес обробки на компоненти («що робиться і з чого це складається»), і модель «сутність-зв'язок» (Entry Relationship model, ER-модель), яка описує сутності, їх атрибути і зв'язки (відношення) між ними. Ці результати є необхідними, але не достатніми. До достатнім результатами слід віднести діаграми потоків даних і діаграми життєвих циклів сутності, які описують динаміку системи. В даний час існує спосіб формалізації проекту – Unified Modelling Language (UML), що дозволяє формально описати різні сторони життєдіяльності розроблюваного проекту. Існує досить велика кількість ПО, що реалізує UML, наприклад Rational Rose, Microsoft Visio. Про UML ми розповімо в окремій частині цієї статті.

1.1.3 Проектування

На етапі проектування формується модель даних. Проектувальники отримують вхідні дані аналізу. Кінцевим продуктом етапу проектування є схема бази даних (якщо така існує в проекті) або схема сховища даних (ER-модель) та набір специфікацій модулів системи (модель функцій).

У разі невеликого проекту одні і ті ж люди можуть виступати в ролі і аналітиків, і проектувальників, і розробників. Виникає питання, наскільки

актуальна передача результатів самому собі. В принципі, досить актуальна, оскільки часто допомагає знайти, наприклад, не описані взагалі, нечітко описані, суперечливо описані компоненти системи та інші недоліки, сприяє запобіганню потенційних помилок, а також дасть можливість ще раз подумати.

Всі специфікації повинні бути гранично точними. План тестування системи також допрацьовується на цьому етапі розробки. У багатьох проектах результати етапу проектування оформляються у вигляді єдиного документа – так званої технічної специфікації. Тут варто згадати про переваги способу UML, який дозволяє отримати одночасно як документи аналізу, які відрізняються меншою деталізацією (їх споживачі – менеджери виробництва), так і документи проектування (їх споживачі – менеджери груп розробки і тестування). Крім того, ПЗ, що реалізує UML, дозволяє здійснити генерацію коду – як мінімум ієрархію класів, а також деякі частини коду самих методів.

Завданнями проектування є:

– розгляд результатів аналізу і перевірка їх повноти;
– семінари з замовником;
– визначення критичних ділянок проекту і оцінка обмежень проекту;
– визначення архітектури системи;
– прийняття рішення про використання продуктів сторонніх розробників, а також про способи інтеграції і механізмах обміну інформацією з цими продуктами;
– проектування сховища даних: модель бази даних, бета-версія бази даних;
– проектування процесів і коду: остаточний вибір засобів розробки, визначення інтерфейсів програм, відображення функцій системи на її модулі та визначення специфікацій модулів;
– визначення вимог до процесу тестування;
– визначення вимог безпеки системи.

1.1.4 Реалізація

При реалізації проекту важливо координувати групу (групи) розробників. Всі розробники повинні підкорятися жорстким правилам контролю вихідних тестів. Група розробників, отримавши технічний проект, починає писати код

модулів. Основне їхнє завдання полягає в тому, щоб усвідомити специфікацію: проектувальник написав, що треба зробити, розробник визначає, як це зробити.

На етапі розробки здійснюється тісна взаємодія проектувальників, розробників і груп тестувальників. У разі інтенсивної розробки тестувальник буквально нерозлучний з розробником, фактично стаючи членом групи розробки.

Проектувальник на етапі розробки виконує функцію «ходячого довідника», оскільки повинен постійно відповідати на питання розробників, що стосуються технічної специфікації.

Найчастіше на етапі розробки змінюються інтерфейси користувача. Це обумовлено, крім іншого, періодичної демонстрацією модулів замовнику. Також можуть істотно змінюватися запити до даних.

Слід наголосити на необхідності існування виділеного робочого місця, де відбувається збірка всього проекту. Саме ці модулі передаються на тестування. Взаємодія тестувальника і розробника без централізованої передачі допустимо тільки в тому випадку, якщо терміново потрібно перевірити якусь правку. Дуже часто етап розробки пов'язаний з етапом тестування, і обидва процеси йдуть паралельно. Синхронізує дії тестерів і розробників система bug tracking (відстежування багів).

Крім того, повинні бути організовані сховища готових модулів проекту та бібліотек, які використовуються при складанні модулів. Це сховище постійно оновлюється. Контролювати процес оновлення повинен одна людина. Одне сховище створюється для модулів, які пройшли функціональне тестування, друге – для модулів, які пройшли тестування зв’язків. Перше – це чернетки, друге – те, з чого вже можна збирати дистрибутив системи і демонструвати його замовнику для проведення контрольних випробувань або для здачі будь-яких етапів робіт.

Слід відзначити виняткову важливість обміну інформацією між проектувальниками, розробниками, тестувальниками: помилки повинні бути класифіковані відповідно до пріоритетів; для кожного класу помилок повинна бути визначена чітка структура дій: «що робити», «як терміново», «хто відповідальний за результат»; кожна проблема має відстежуватися проектувальником / розробником / тестувальником, що відповідає за її усунення. Те ж саме стосується ситуацій, коли порушуються заплановані терміни розробки, передачі модулів на тестування. Всі проблеми при взаємодії між групами можуть коштувати досить дорого.

1.1.5 Тестування

Групи тестування можуть залучатися до співпраці вже на ранніх стадіях розробки проекту. Строго кажучи, комплексне тестування слід виділити в окремий етап розробки. Залежно від складності проекту тестування і виправлення помилок може займати третину, половину загального часу роботи над проектом і навіть більше.

Чим складніше проект, тим більше буде потреба в автоматизації системи зберігання помилок – bug tracking, яка забезпечує наступні функції:

– зберігання повідомлення про помилку (до якого компоненту системи

– відноситься помилка, хто її знайшов, як її відтворити, хто відповідає за її виправлення, коли вона повинна бути виправлена);

– система повідомлення про появу нових помилок, про зміну статусу відомих в системі помилок (повідомлення по електронній пошті);

– звіти про актуальні помилки по компонентах системи;

– інформація про помилку і її історія;

– правила доступу до помилок тих чи інших категорій;

– інтерфейс обмеженого доступу до системи bug tracking для кінцевого користувача.

Подібні системи беруть на себе безліч організаційних проблем, зокрема питання автоматичного повідомлення про помилки.

Власне тести систем можна розділити на кілька категорій:

– автономні тести модулів; вони використовуються вже на етапі розробки компонентів системи і дозволяють відстежувати помилки окремих компонентів;

– тести зв’язків компонентів системи; ці тести також використовуються і на етапі розробки, і на етапі тестування, вони дозволяють відслідковувати правильність взаємодії та обміну інформацією компонентів системи;

– системний тест; він є основним критерієм приймання системи; як правило, це група тестів, що включає і автономні тести, і тести зв’язків і моделі; даний тест повинен відтворювати роботу всіх компонентів і функцій системи; основна мета даного тесту – внутрішня приймання системи і оцінка її якості;

– приймально тест; основне його призначення – здати систему замовнику;

– тут розробники часто занижують вимоги до системи в порівнянні з системним тестом, і причини цього цілком очевидні;

– тести продуктивності і навантаження; дана група тестів входить в системний тест, але гідна окремої згадки, оскільки саме ця група тестів є основною для оцінки надійності системи.

В тести кожної групи обов'язково входять тести моделювання відмов. Тут перевіряється реакція компонента, групи компонентів, системи в цілому на відмови виду:

– відмова окремого компонента інформаційної системи;
– відмова групи компонентів інформаційної системи;
– відмова основних модулів інформаційної системи;
– відмова операційної системи;
– жорсткий збій (відмова харчування, жорстких дисків).

Ці тести дозволяють оцінити якість підсистеми відновлення коректного стану інформаційної системи і служать основним джерелом інформації для розробки стратегій запобігання негативним наслідкам збоїв при промисловій експлуатації. Як правило, цим класом тестів розробники традиційно зневажають, а потім змушені «лікувати» наслідки збоїв на промисловій системі.

Ще одним важливим аспектом програми тестування інформаційних систем є наявність генераторів тестових даних. Вони використовуються для проведення тестів функціональності системи, тестів надійності системи і тестів продуктивності системи. Завдання оцінки характеристик залежності продуктивності інформаційної системи від зростання обсягів оброблюваної інформації без генераторів даних вирішити неможливо.

1.1.6 Впровадження

Дослідна експлуатація перекриває процес тестування. Система рідко вводиться повністю. Як правило, це процес поступовий або ітераційний – в разі циклічного життєвого циклу.

Введення в експлуатацію проходить як мінімум три стадії:

– первісна завантаження інформації;
– накопичення інформації;
– вихід на проектну потужність (тобто власне перехід до етапу експлуатації).

Первісна завантаження інформації ініціює досить вузький спектр помилок: в основному мова йде про проблеми неузгодженості даних при завантаженні і про власні помилки завантажувачів. Тут потрібно застосувати методи контролю якості даних (в іншому випадку в подальшому наведені помилки обійдуться набагато дорожче). Це те, чого не було або не могло бути відстежено на тестових даних. Такі помилки повинні бути виправлені якомога швидше. Не полінуйтеся поставити налагоджену версію системи (якщо, звичайно, вам дозволять розгорнути весь комплекс супроводжуючого налагодження інформаційної системи ПЗ на місці). Якщо налагодження «на живих» даних зробити неможливо, вам доведеться моделювати ситуацію, і як можна швидше. У цьому випадку потрібні дуже кваліфіковані тестувальники.

В період накопичення інформації з інформаційної системи виявляється найбільша кількість помилок, пов'язаних з розрахованих на багато користувачів доступом. Часто на етапі тестування їм не приділяється належної уваги – через складність моделювання і дорожнечі засобів автоматизації процесу. Друга категорія виправлень пов'язана з тим, що користувача не влаштовує інтерфейс. Тут не завжди потрібно виконувати абсолютно всі побажання користувача, інакше процес введення в експлуатацію буде нескінченним. В даний момент циклічні моделі і моделі зі зворотним зв'язком етапів також дозволяють знизити витрати.

Цей етап є також найбільш серйозним тестом – тестом схвалення користувачем (customer acceptance tests). Якщо цього не було зроблено на етапі тестування, то на етапі впровадження це неодмінно станеться, і вашим тестувальником фактично буде ваш власний замовник, що не завжди приємно, особливо якщо для останнього це буде дещо несподівано.

Вихід системи на проектну потужність в хорошому варіанті – це доведення дрібних помилок і рідкісні серйозні помилки.

1.1.7 Експлуатація та технічна підтримка

Тут останнім документом, від якого залежать розробники, є документ технічного приймання. Отже, проект зданий, розробка завершена, помилки, з якими замовник погодився, описані в документації, і в ідеалі сторони задоволені один одним. Документ також визначає необхідний персонал та необхідне обладнання для підтримки працездатності системи, а також умови порушення

експлуатації продукту і відповідальність сторін. Крім цього, якщо документ укладається між сторонами, він містить умови технічної підтримки.

Умови технічної підтримки та ситуації, які підпадають під поняття «технічна підтримка», а також умови оплати визначаються, як правило, в окремому документі. Іноді в рекламних цілях менеджери розробки намагаються включити в документи зобов'язання гарантії виправлення помилок на умовах технічної підтримки за «ікс» днів. Такий підхід, можливо, і хороший з рекламною точки зору, але слід віддавати собі звіт в тому, що невиконання зобов'язань з технічних причин набагато гірше, ніж відсутність «смачною» реклами.

Чергування цих етапів, взаємодія між ними може змінюватися, виходячи з обраної вашим керівником або вами моделі процесу розробки ПЗ.

1.2 Методології розробки програмного забезпечення

Методологія розробки ПЗ – це система, яка визначає порядок виконання завдань, методи оцінки та контролю. Моделі розробки ПЗ вибирають, виходячи з напрямку проекту, його бюджету, термінів реалізації кінцевого продукту, а також увагу варто звернути й на характер і темперамент керівника проекту і його команди. Підходи розробки ПЗ відрізняються один від одного тим, як етапи життєвого циклу програмного забезпечення взаємопов'язані між собою всередині циклу розробки.

Далі я хочу розглянути популярні підходи до розробки ПЗ.

«Waterfall Model» (Водоспад)

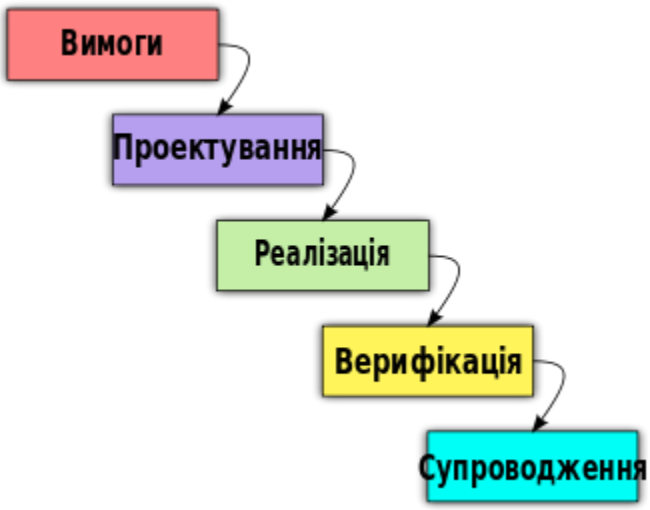


Рисунок 1.1 – Структурна схема методології «Водоспад»

Як відомо, цю модель ПЗ відносять до класики, так як вона одна з найперших моделей розробки ПЗ. Її суть полягає в послідовному виконанні етапів життєвого циклу: аналіз, дизайн, розробка, тестування, реліз і підтримка. Кожна стадія має бути завершена до початку нової. Перевагою даної моделі є те, що оцінку якості можна виконувати після кожного етапу. Але важко уявити проект, який можна було б реально виконати строго послідовно. Тому модель часто вважають застарілою. Як до плюсів, так і до мінусів можна віднести те, що при

використанні даного підходу бюджет проекту і терміни його реалізації точно визначені. Також водоспад не дає робити кроків назад, що ускладнює тестування, яке найчастіше здійснюється в кінці розробки продукту. Виходить, що наступні зміни будуть значно збільшувати бюджет і затягувати реліз проекту. Таку модель я би порадив використовувати тільки на маленьких проектах, де точно не виникне протиріч з вимогами. Каскадна модель не зовсім сумісна з розробкою ПЗ.

Як можна побачити на рис. 1.1, у водоспаді розвиток однієї фази починається лише тоді, коли попередня фаза завершена. Через цю природу кожна фаза моделі водоспаду досить точно визначена. Оскільки фази падають з вищого рівня на нижчий, як водоспад, його називають моделлю водоспаду.

Ця методологія використовується у наступних випадках:

– вимоги стабільні і не змінюються часто;
– додаток невеликий;
– немає вимоги, яка не зрозуміла або не дуже чітка;
– навколишнє середовище стабільне;
– використовувані інструменти та методи стабільні та не динамічні;
– ресурси добре навчені та доступні.

До явних плюсів використання методології «водоспад» можна віднести:

– проста і легка для розуміння та використання;
– для менших проектів модель «водоспад» працює добре і дає відповідні результати;
– оскільки фази жорсткі і точні, одна фаза робиться одна за одною, її легко підтримувати;
– критерії входу та виходу чітко визначені, тому легко та систематично підтримувати якість;
– результати добре задокументовані.

Звісно у такої застарілої методології розробки ПЗ є дуже багато мінусів, які не дозволяють використовувати її у більшості сучасних проектів:

– неможливо прийняти зміни в вимогах;
– повернутися назад до фази стає дуже важко, наприклад, якщо програма зараз перейшла на етап тестування і є вимога зміни, повернутися назад і змінити її стає важко;
– постачання кінцевого продукту запізнюється, оскільки немає прототипу, який би демонструвався проміжно;

– для великих та складних проектів ця модель не є хорошою, оскільки фактор ризику вищий;

– не підходить для проектів, де вимоги часто змінюються;

– не працює для тривалих і поточних проектів;

– оскільки тестування проводиться на більш пізньому етапі, воно не дозволяє визначити проблеми та ризики на більш ранній фазі, тому стратегію зменшення ризику важко підготувати.

«V-Model» (Крок за кроком або validation and verification)

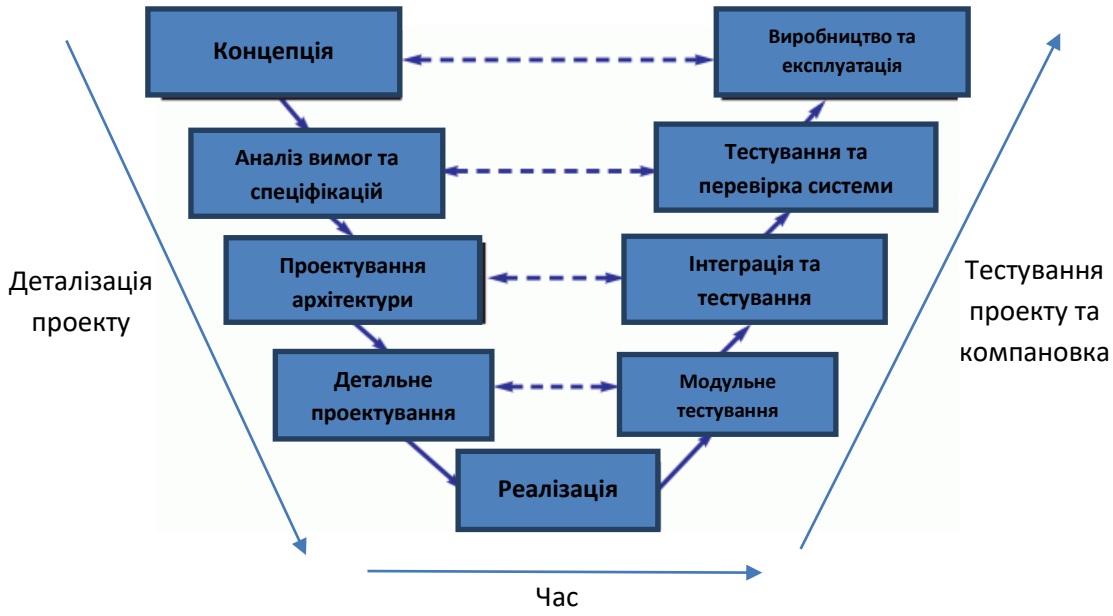


Рис 1.2 – Графічне відображення «V-Model» методології

Одним з основних недоліків моделі водоспаду було те, що дефекти були виявлені на дуже пізньому етапі процесу розробки, оскільки тестування було зроблено в кінці циклу розробки. Виправити дефекти стало дуже складно і дорого, оскільки вони були виявлені на дуже пізньому етапі. Для подолання цієї проблеми була введена нова модель розробки, яка називається «V-модель». Назву ця модель отримала від V-образної форми структурної схеми моделі(рис 1.3).

Модель розробки ПО орієнтована на те, щоб детально перевіряти і тестувати продукт на перших стадіях розробки. Вже у момент написання коду розробниками тестувальники пишуть модульні тести, тобто починають тестування паралельно з розробкою. Рекомендується дотримуватися даного підходу, якщо вам вкрай

важливо безперебійне функціонування продукту, а також відомі чіткі вимоги. V-Model відносять до практик екстремального програмування.

Щоб зрозуміти модель V, спочатку розберемося, що таке «Verification and Validation» програмного забезпечення.

Verification(від англ. верифікація) – верифікація є технікою статичного аналізу. У цій техніці тестування проводиться без виконання коду. Серед прикладів – огляди, перевірка.

Validation (від англ. валідація) – це метод динамічного аналізу, коли тестування проводиться шляхом виконання коду. Приклади включають функціональні та нефункціональні методи тестування.

У «V-моделі» контроль якості здійснюються одночасно з процесом розробки. Немає дискретної фази під назвою «тестування», скоріше тестування починається з фази вимог. Діяння з верифікації та валідації йдуть одночасно.

«Agile Model» (Гнучка модель)

Рисунок 1.3 – Графічне відображення гнучкої моделі розробки ПЗ

У таких моделях всі етапи живого циклу були виконані в ході однієї ітерації та готові до впровадження будь-яких змін. Тобто ваш проект ділиться на спринти – відрізки часу, що має принести якийсь отриманим результатом, як правило один спринт займає від однієї до чотирьох неділь. У кожному спринті є свій власний список завдань, який повинен бути виконаний на конкретній ітерації, і кожна із задач повинна мати свій рівень оцінки. Задачі та прогрес їх виконання обговорюються щоденними зустрічами, в яких команда обтягує, хто що зробив, що збирається зробити і які є важливі проблеми. Окрім цього, на початку спринту проводиться зустріч за плануванням завдання на ітерацію, а в кінці – ретроспективна (зустріч для обговорення результатів).

Методологія ідеально підходе, коли ваш проект постійно адаптується до умов ринку, має великий об'єм і довгий живий цикл. Якщо ви творчий керівник з мільйоном нових ідей, які постійно перевіряють, то цей метод саме для вас.

Agile поділяється на декілька окремих гнучких підходів:

1. Scrum (ітераційна модель розробки ПЗ або «вир»). У такому підході в команді часто з'являється Scrum-майстер, який забезпечує постійну роботу всіх команд, створюючи для всіх гарні умови: підтримку, мотивацію, організовану процедуру, проводить зустрічі. Така система добре підійде проекту до десяти людей. Цей підхід також дозволяє дуже часто реагувати на зміну вимог від клієнта та підлаштовуватись під ситуацію майже на кожному етапі розробки.

2. Kanban (з японского перекладається як «Картка»). Даний підтип розробки відрізняється своєю візуалізацією життєвого циклу. Команда орієнтується на виконання завдань, які задаються індивідуально: завдання довге переходить через всі етапи – розробка коду, перегляд коду, тестування, введення в експлуатацію (етапи можуть бути змінені в індивідуальному порядку). Ціль кожного з учасників команди – зменшити кількість завдань у першому складі. Даний наглядний підхід дозволяє зрозуміти, де саме виникла, або може виникнути проблема, а також дозволяє просто побачити організацію всього проекту.

RUP (відома як rapid application development, швидка розробка заявки, інкрементальна модель)

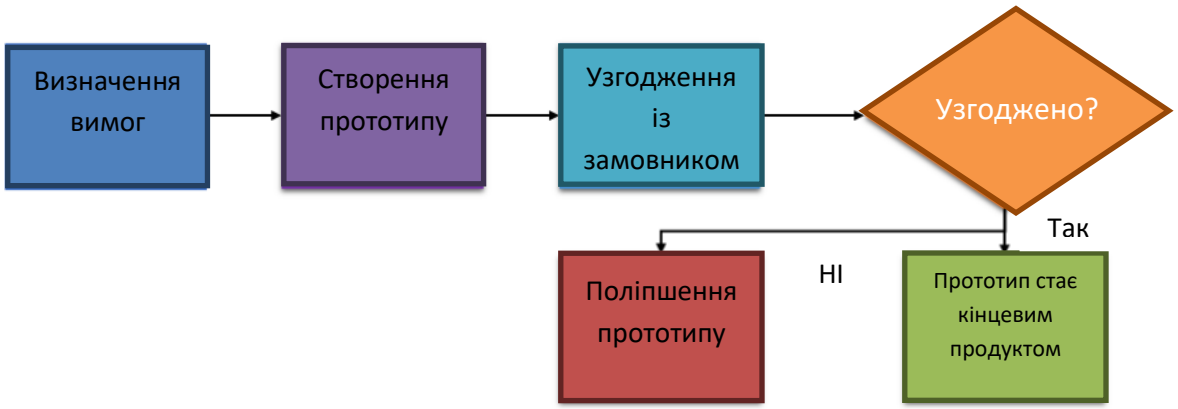


Рисунок 1.4 – Графічне відображення методології RUP

Така модель підрозділяється на кілька циклів, які складають життєвий цикл ПЗ – «мульти-водоспад». У кожному циклі є свої модулі, які проходять через етапи: спочатку визначення вимог, потім проектування, після кодування, тестування і впровадження. Виходить, що дана модель має на увазі, що на першому етапі будуть випущені основні функції, а наступні цикли дозволять додавати нові доповнення. Процес закінчиться тільки після повної імплементації системи. Використовувати Інкрементальний модель коштує тільки тоді, коли вимоги кристально ясні, одночасно можливе впровадження додаткових фіч. Кращим підходом буде саме цей в разі, якщо ви володієте ризиковими нововведеннями, і продукт потрібно максимально швидко вивести на маркет. Відзначимо й те, що завдяки поступовій інтеграції функцій, ваш проект буде дешевше, ніж міг би бути при іншій моделі, і з меншими ризиками.

Spiral model (спіральна модель)

Рисунок 1.5 – Графічне відображення спіральної методології розробки ПЗ

Дана модель спрямована на аналіз оцінки ризиків і відмінно підійде там, де немає права на помилку. Також її зручно використовувати для введення нових лінійок продукту і проведення досліджень. Виглядає ця модель, як спіраль, так як починає оцінювати ризики спочатку на локальних програмах, намагаючись запобігти ризикам, далі переходить на новий виток спіралі, перенаправлені до роботи з більш комплексними задачами.

Підхід розумніше використовувати на великих і дорогих проектах.

Спіральна модель має чотири фази:

1. Планування.
2. Аналіз ризиків.
3. Інженерна фаза.
4. Оціночна фаза.

Дії, які виконуються у фазах спіральної моделі, показані у таблиці.

Таблиця 1.1 – Фази розробки ПЗ використовуючи спіральну методологію

Фаза	Дії які виконуються	Результати
Фаза планування	- Вивчаються вимоги; - технічно-економічне обґрунтування; - перевірка для впорядкування вимог.	- Вимоги щодо розуміння документа; - доопрацьований перелік вимог.
Аналіз ризиків	- Вивчаються вимоги та проводяться сеанси мозкового штурму для виявлення потенційних ризиків; - після виявлення ризиків планується та доопрацьовується стратегія зменшення ризику.	Документ, в якому висвітлюються всі ризики та плани їх пом'якшення.
Інженерна фаза	Фактична розробка та тестування, якщо програмне забезпечення відбувається на цій фазі.	Код. Випробування та результати тестів Підсумковий звіт тесту та звіт про дефекти.
Оціночна фаза	Клієнти оцінюють програмне забезпечення та надають відгуки та схвалення.	Документ реалізованих особливостей.

Extreme Programming (екстремальне програмування, XP)

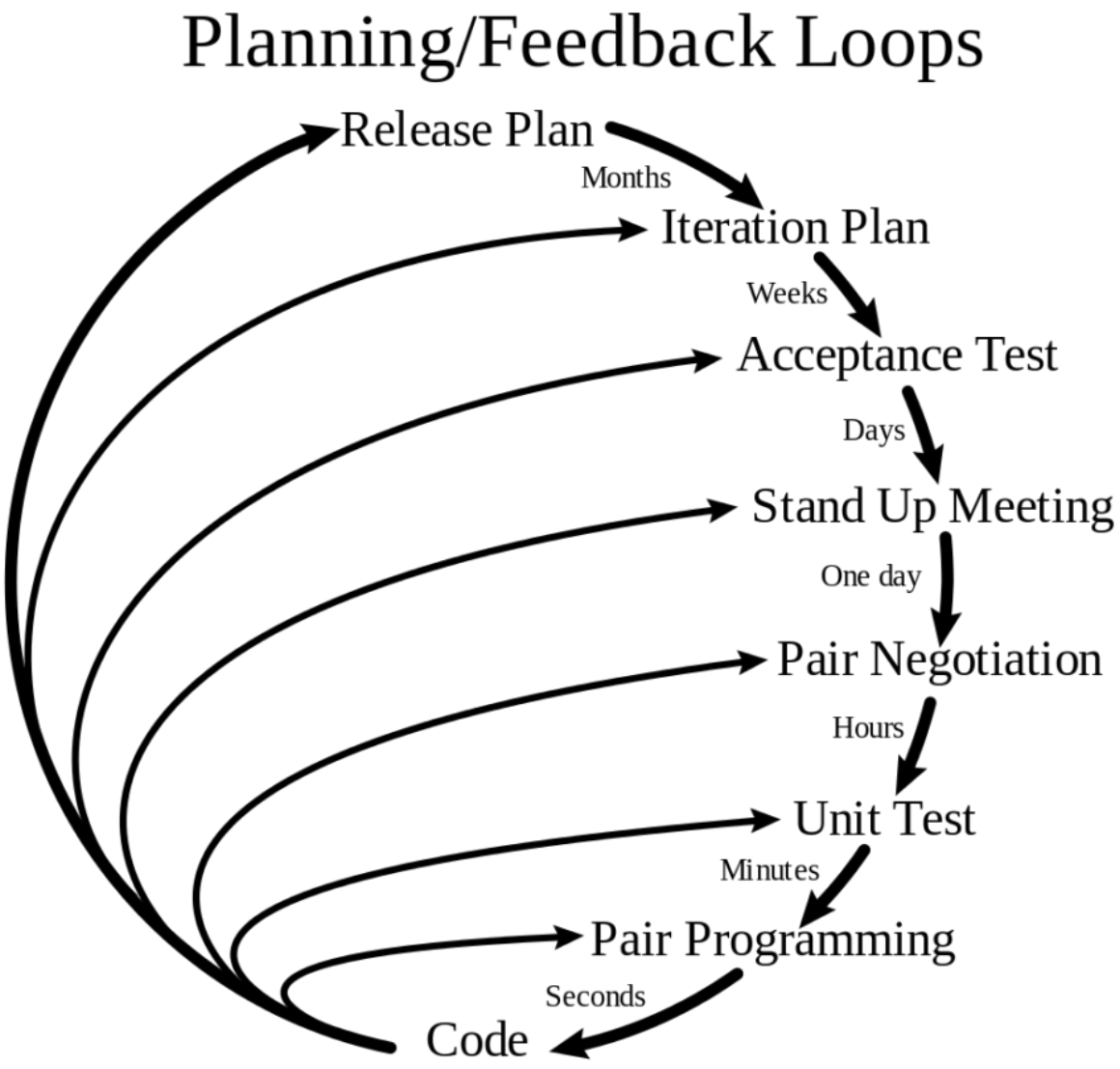


Рисунок 1.6 – Графічне відображення екстремального програмування

Екстремальне програмування вважається неформальним підходом розробки ПЗ, де кожен розробник – професіонал свого діла. Якщо ж відношення міняється, то використовування цієї методології марне. Розробка продукту ведеться короткими ітераціями. Екстремальність підходу у тому, що використовується

перше просте рішення, що створює великий ризик. У методології практично практикується парне програмування та групова розробка. Метою такої методології є створити максимально довірливі відносини с замовником і значно скоротити термін розробки продукту.

Головне в екстремальному програмі це не втратити контроль над тим що відбувається, щоб розробка продукту не перетворилася у хаос.

1.3 DevOps методологія

DevOps (від англ. Development and Operations) – це потомство гнучкої (agile) методології розробки програмного забезпечення – народжене від необхідності йти в ногу з підвищеною швидкістю програмного забезпечення та прохідними методами. Удосконалення спритної культури та методів за останнє десятиліття показало необхідність більш цілісного підходу до життєвого циклу доставки програмного забезпечення.

З Agile методологією ми вже ознайомились у підрозділі 1.2, але всеж хотілось би доповнити декількома реченнями та подивитись більш детально на цю методологію у порівнянні з DevOps. Agile Development – це паралельний термін для декількох ітеративних та поступових методологій розробки програмного забезпечення. Найпопулярніші гнучкі методології включають Scrum, Kanban, Scaled Agile Framework (SAFe), Lean Development та Extreme Programming (XP).

У той час як кожна з гнучких методологій унікальна своїм специфічним підходом, всі вони поділяють загальне бачення та основні цінності (див. Маніфест Agile). Всі вони принципово включають ітерацію та постійний зворотний зв'язок, який вона забезпечує для послідовного вдосконалення та доставки програмної системи. Усі вони передбачають постійне планування, безперервне тестування(CT), неперервну інтеграцію(CI) та інші форми безперервної еволюції як проекту, так і програмного забезпечення. Усі вони легкі, особливо порівняно з традиційними процесами у стилі водоспаду, і за своєю суттю адаптуються. І що найважливіше у гнучких методах – це те, що всі вони зосереджені на наданні можливості людям співпрацювати та приймати рішення разом швидко та ефективно.

На початку agile-команди в основному складалися з розробників. Оскільки ці agile-команди стали більш ефективними у виробництві програмного забезпечення, стало зрозуміло, що забезпечення якості (QA) та розробка коду як

окремих команд було неефективним. Agile виріс для того щоб мати можливість охоплювати QA, щоб збільшити швидкість доставки програмного забезпечення, і тепер Agile знову зростає, щоб охопити членів доставки та підтримки, щоб розширити гнучкість.

DevOps – це IT-розум, який заохочує спілкування, співпрацю, інтеграцію та автоматизацію серед розробників програмного забезпечення та IT-операцій з метою підвищення швидкості та якості доставки програмного забезпечення.

Команди DevOps зосереджуються на стандартизації середовищ розробки та автоматизації процесів доставки, щоб поліпшити передбачуваність доставки, ефективність, безпеку та ремонтопридатність. Ідеали DevOps надають розробникам більше контролю над виробничим середовищем та кращим розумінням виробничої інфраструктури. DevOps заохочує розширення можливостей команд з самостійністю будувати, перевіряти, доставляти та підтримувати додатки.

Методологія фокусується на стандартизації оточень розробки з метою сприяння швидкому випуску версій. В ідеалі, системи автоматизації збирання і випуску повинні бути доступні всім розробникам в будь-якому оточенні, і у розробників повинен бути контроль над оточенням, а інформаційно-технологічна інфраструктура повинна ставати більш сфокусованої на додатку.

Завдання DevOps – зробити процес розробки і постачання програмного забезпечення узгодженим з експлуатацією, часто ці завдання вирішуються за підтримки автоматичних засобів.

DevOps-рух виник в 2009 році і було покликане вирішити проблеми взаємодії команд розробки та експлуатації програмних продуктів, в тому ж році в Бельгії була організована серія конференцій «DevOps Days». Потім «DevOps Days» проходили в різних містах і країнах світу.

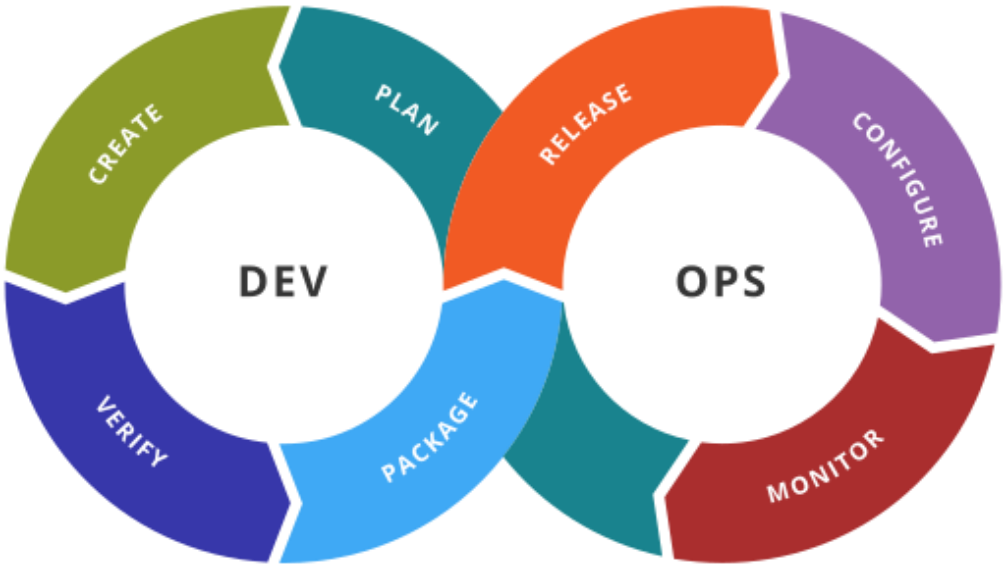


Рисунок 1.7 – Гнучкість DevOps методології

Оскільки DevOps – це командна робота (між співробітниками, що займаються розробкою, операціями і тестуванням), немає єдиного інструменту DevOps: це скоріше набір, що складається з декількох інструментів. Як правило, інструменти DevOps вписуються в одну або кілька з цих категорій, що відображає ключові аспекти розробки і доставки програмного забезпечення:

Code – розробка і аналіз коду, інструменти контролю версій, злиття коду;

Build – інструменти безперервної інтеграції, статус збірки;

Test – інструменти безперервного тестування, які забезпечують зворотний зв'язок з бізнес-ризиків;

Package – репозиторій артефактів, попередня установка додатки;

Release – управління змінами, офіційне затвердження випуску, автоматизація випуску;

Configuration – конфігурація і управління інфраструктурою, Інфраструктура як інструменти коду;

Monitoring – моніторинг продуктивності додатків, досвід роботи з кінцевим користувачем.

Незважаючи на те, що є безліч інструментів, деякі категорії з них мають особливо важливе значення в налаштуванні інструментальних засобів DevOps для використання в організації. Деякі спроби ідентифікувати ці основні інструменти можна знайти в існуючій літературі.

Такі інструменти, як Docker (контейнеризація), Jenkins (неперервна інтеграція), Puppet (інфраструктура як код) і Vagrant (платформа віртуалізації) – і багато інших – часто використовуються і часто згадуються в дискусіях по інструментах DevOps. У ході виконяння магістрської роботи я буду намагатись порівняти та висвітлити основні інструменти саме неперервної інтеграції та розгортання (CI/CD).

Agile і DevOps схожі, але Agile є зміна мислення і практики (що має привести до організаційних змін), а DevOps приділяє більше уваги впровадженню організаційних змін для досягнення своїх цілей.

Потреба в DevOps народжувалася через зростаючу популярність програмного забезпечення Agile, оскільки це призводить до збільшення кількості випущених версій.

2 БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ, РОЗГОРТАННЯ ТА ТЕУСТВАННЯ

2.1 Безперервна інтеграція коду

Безперервна інтеграція (continuous integration, CI) – це практика розробки програмного забезпечення DevOps, при якій розробники регулярно об'єднують зміни програмного коду в центральному репозиторії, після чого автоматично виконується збірка, тестування і запуск. Поняття безперервної інтеграції найчастіше застосовується до стадії складання чи інтеграції процесу випуску ПЗ і включає в себе як компонент автоматизації (наприклад, сервіс безперервної інтеграції або збірки), так і компонент культури розробки (наприклад, навчання частою інтеграцією). Головне завдання безперервної інтеграції – швидше знаходити і виправляти помилки, покращувати якість ПЗ і скорочувати тимчасові витрати на перевірку і випуск нових оновлень ПЗ.

Основна ідея такого підходу полягає в тому, щоб звести до мінімуму вартість інтеграції, зробивши це на ранньому етапі. Розробники можуть виявити конфлікти між новим та існуючим кодом на ранній стадії, коли їх ще відносно легко усунути. Після вирішення конфлікту робота може бути продовжена з упевненістю, що новий код відповідає вимогам існуючої кодової бази.

Інтеграція коду часто сама по собі не дає ніяких гарантій щодо якості або функціональності нового коду. У багатьох організаціях інтеграція коштує дорого: адже для забезпечення того, щоб код відповідав стандартам, не вносив помилок і не порушував існуючу функціональність, використовуються ручні процеси. Часта інтеграція може створювати перешкоди, коли рівень автоматизації не відповідає заходам забезпечення якості.

Щоб усунути ці перешкоди в процесі інтеграції, на практиці CI спирається на надійні набори тестів і автоматизовану систему для їх виконання. Коли розробник об'єднує код з основним сховищем, автоматизовані процеси запускають збірку нового коду. Після цього для нової збірки запускаються тестові набори, які перевіряють, чи не виникли якісь проблеми. Якщо на етапі складання або тестування відбувається збій, команда отримує попередження і може виправити збірку.

Кінцева мета безперервної інтеграції – зробити інтеграцію простим, повторюваним процесом, який є частиною повсякденного робочого процесу

розробки, знизити витрати на інтеграцію і своєчасно реагувати на дефекти коду. Робота над надійністю і швидкістю автоматизованої системи, а також розвиток командної культури, яка заохочує часті ітерації і швидке реагування на проблеми, мають основоположне значення для успіху стратегії.

При безперервній інтеграції розробники часто вносять зміни в спільно використовуваний репозиторій, використовуючи систему контролю версій, наприклад Git. Перед внесенням кожної зміни у репозиторій, розробники можуть запускати локальні модульні тести програмного коду в якості додаткового рівня перевірки перед інтеграцією. Сервіс безперервної інтеграції (рис. 2.1) автоматично виконує складання(build) та запуск модульних тестів для змін коду, що дозволяє моментально виявляти помилки.

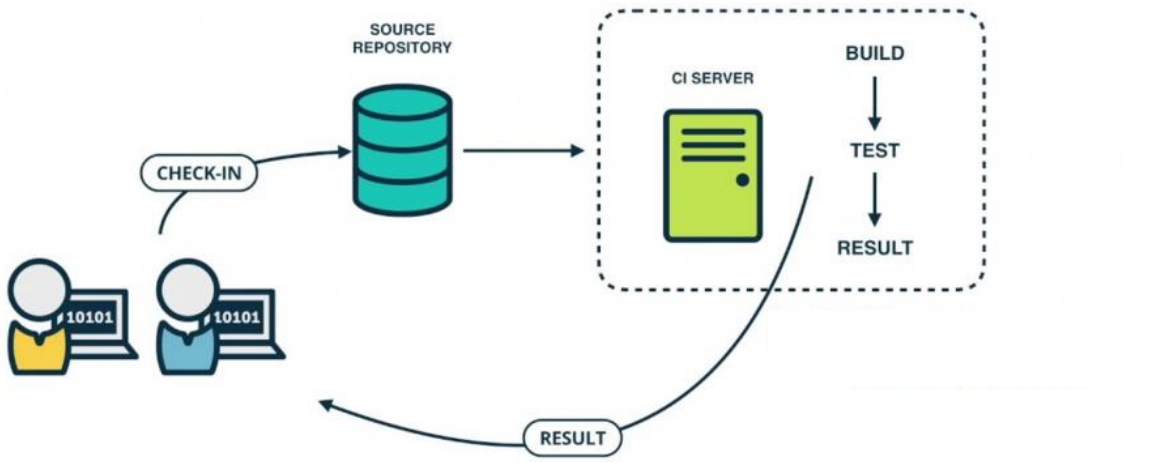


Рисунок 2.1 – Графічне відображення використання сервісу безперервної інтеграції коду

Безперервна інтеграція відноситься до стадії складання і модульного тестування процесу випуску ПЗ. Кожна підтверджена зміна коду запускає автоматичний процес складання та тестування.

За допомогою безперервної доставки зміни програмного коду автоматично проходять збірку, тестуються і готуються до запуску в робочому середовищі. Безперервна доставка розширює практику безперервної інтеграції за рахунок того, що всі зміни коду після стадії складання розгортаються в тестовому і / або в робочому середовищі.

Використання та підтримування процесу безперервної інтеграції дозволяє значно прискорити процес розробки та тестування ПЗ та збільшити ефективність використання часу розробки. Нижче наведено ряд переваг використання цієї практики:

1. Безперервна інтеграція підвищує продуктивність вашої команди за рахунок звільнення розробників від ручної роботи і стимуляції підходів, які допомагають зменшити кількість помилок і дефектів в версіях ПО для кінцевих користувачів.
2. Швидке виявлення і усунення помилок. За рахунок більш частого і всебічного тестування ваша команда зможе виявляти і усувати помилки завчасно, до того, як вони переростуть в серйозні проблеми.
3. Швидка доставка оновлень. Безперервна інтеграція дає можливість вашій команді швидше і частіше доставляти оновлення кінцевим користувачам.

2.2 Безперервна доставка та розгортання коду

Безперервна доставка (continuous delivery or deployment, CD) – це практика розробки програмного забезпечення, коли за будь-яких змінах в програмному коді виконується автоматичний збір (build), тестування і підготовка до остаточного випуску. Безперервна доставка є одним з основоположних принципів розробки сучасних додатків, оскільки розширює практику безперервної інтеграції за рахунок того, що всі зміни коду після стадії складання розгортаються в тестовій і/або в робочому середовищі. При правильному впровадженні у розробників завжди буде готовий до розгортання зібраний екземпляр ПЗ, що пройшов стандартизовану процедуру тестування.

Безперервна доставка дозволяє розробникам не тільки автоматизувати тестування на рівні модулів, але і виконувати різнопланову перевірку оновлень додатків перед тим, як розгортати їх для кінцевих користувачів. Таке тестування може включати тестування користувальницького інтерфейсу, завантаження, інтеграції, надійності API (програмний інтерфейс додатку, інтерфейс прикладного програмування) (від англ. application programming interface, API) і т.д. Все це дозволяє розробникам ретельніше перевіряти оновлення і завчасно виявляти можливі проблеми. На відміну від застарілих локальних рішень, хмарна середовище дозволяє легко і економічно автоматизувати створення і реплікацію декількох середовищ тестування.

Як показано на рис. 2.2, під час використання безперервної доставки кожна зміна програмного коду проходить збірку, тестується і потім вирушає в підготовчу (тестову або імітаційну) середу. Перед розгортанням у робочому середовищі можна використовувати кілька паралельних стадій тестування. Відмінність безперервної доставки від безперервного розгортання полягає в тому, що при безперервної доставці для розгортання оновлень в робочому середовищі потрібне підтвердження вручну. При безперервному розгортанні це відбувається автоматично без спеціального підтвердження.

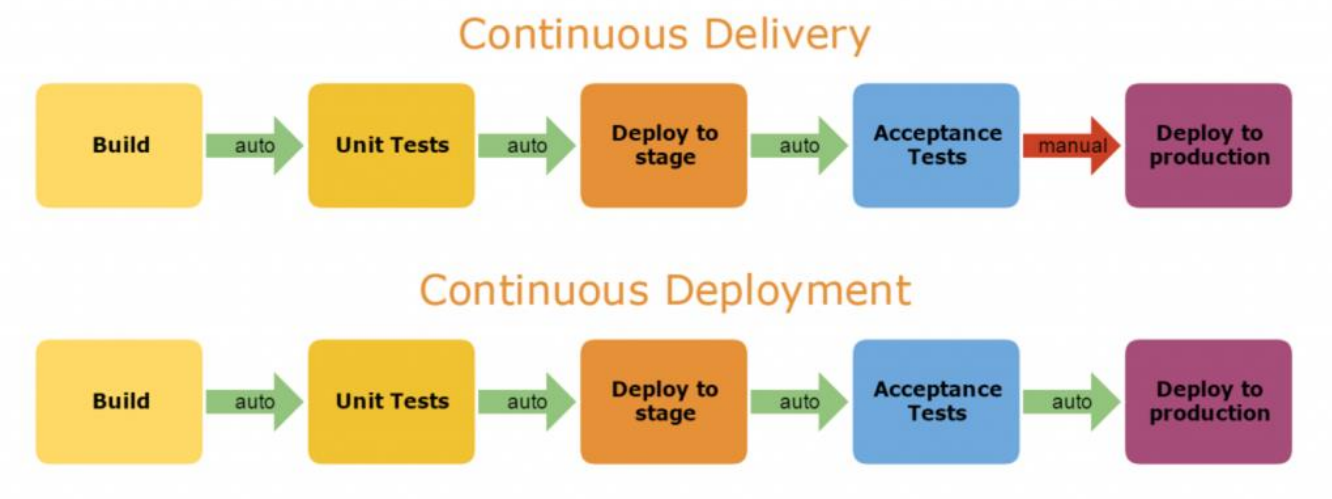


Рисунок 2.2 – Графічне відображення безперервної доставки та розгортання коду

Підтвердженням ефективності використання CD є наступні переваги:

1. Автоматизація процесу випуску ПО. Безперервна доставка дозволяє вашій команді автоматично виконувати збірку, тестувати і готувати зміни коду до запуску в робочому середовищі, що забезпечує більш ефективну і швидку доставку ПО.

2. Швидке виявлення і усунення помилок. За рахунок більш частого і повного тестування ваша команда зможе виявляти і усувати помилки заздалегідь, до того, як вони переростуть в серйозні проблеми. Безперервна доставка дозволяє спростити додаткове тестування вашого коду за рахунок автоматизації всього процесу.

3. Більш продуктивна розробка. Застосування практики безперервної доставки підвищує продуктивність вашої команди за рахунок звільнення

розробників від ручної роботи і стимуляції підходів, які допомагають зменшити кількість помилок і дефектів в розгортанні для кінцевих користувачів.

4. Швидка доставка оновлень. Безперервна доставка дає вашій команді можливість доставляти поновлення кінцевим користувачам швидше і частіше. При правильному впровадженні безперервної доставки у вас завжди буде готовий до розгортання зібраний екземпляр ПО, що пройшов стандартизовану процедуру тестування.

Отже, виходячи з вищесказаного можна зробити невеликий висновок щодо використання цих методів прискорення розробки ПЗ. Команда розробки зливає свої невеликі короткоживучі функціональні гілки (тобто частину розробленого функціоналу) з основною гілкою розробки по кілька разів на день і за рахунок безперервної інтеграції (CI), процеси побудови та тестування повністю автоматизовані, результат маємо в межах 10 хвилин. Далі за рахунок безперервної доставки автоматизується безперервна інтеграція (CI) та увесь весь процес релізу ПЗ. І нарешті, використовуючи безперервне розгортання, автоматизується повністю увесь процес розробки програмного забезпечення від написання коду до моменту релізу продукту у робоче середовище.

2.3 Безперервне тестування (CT)

Безперервне тестування – це процес виконання автоматизованих тестів у рамках конвеєра доставки програмного забезпечення для негайного зворотного зв’язку щодо бізнес-ризиків, пов'язаних з кандидатом на випуск програмного забезпечення.

Причини виникнення безперервного тестування знову таки спираються на ті ж самі проблеми які виникли перед усіма сферами бізнесу, які так чи інакше пов’язані з IT. З початком широкого використання методології DevOps та Agile, організації очікують, що команди з розробки програмного забезпечення будуть готувати все більше і більше інноваційного програмного забезпечення протягом коротших циклів доставки. Отже, організації приймають рішення використовувати постійне тестування (рис. 2.3), оскільки вони визнають, що є ряд проблеми заважають їм доставляти якісне програмне забезпечення з потрібною швидкістю. Вони визнають зростаючу важливість програмного забезпечення, а також зростаючу вартість відмови програмного забезпечення, і більше не готові робити компроміс між часом, обсягом та якістю.

Рисунок 2.3 – Графічне відображення процесу безперервного тестування

Метою постійного тестування є забезпечення швидкого та постійного зворотного зв'язку щодо рівня бізнес-ризику в останньому кандидаті на складання чи реліз. Потім ця інформація може бути використана для того, щоб визначити, чи програмне забезпечення готове просуватись через трубопровід доставки в будь-який момент часу.

Оскільки тестування починається рано і виконується постійно, ризики застосування піддаються негайному результату після їх введення. Потім команди розробників можуть запобігти переходу цих проблем на наступний етап CI/CD/CT. Це зменшує час і зусилля, які потрібно витратити на пошук та виправлення дефектів. Як результат, можна збільшити швидкість і частоту, з якою постачається якісне програмне забезпечення (програмне забезпечення, яке відповідає очікуванням прийнятного рівня ризику), а також зменшити технічну заборгованість.

Більше того, коли зусилля щодо якості програмного забезпечення та тестування узгоджуються з очікуванням бізнесу, виконання тесту створює пріоритетний перелік діючих завдань (а не потенційно переважна кількість висновків, які потребують ручного огляду). Це допомагає командам зосередити зусилля на якісних завданнях, які матимуть найбільший вплив, виходячи з цілей та пріоритетів організації.

Крім того, коли команди постійно проводять широкий набір безперервних тестів по всьому циклу розробки, вони збирають показники щодо якості процесу, а також стану програмного забезпечення. Отримані показники можна використовувати для повторного вивчення та оптимізації самого процесу,

включаючи ефективність цих тестів. Ця інформація може бути використана для встановлення циклу зворотного зв'язку, який допомагає командам поступово вдосконалювати процес. Часті вимірювання, чіткі петлі зворотного зв'язку та постійне вдосконалення є ключовими принципами DevOps.

Постійне тестування включає перевірку як функціональних, так і нефункціональних вимог.

Для тестування функціональних вимог (функціональне тестування) безперервне тестування часто передбачає тести одиниць, тестування API, тестування інтеграції та тестування системи. Для тестування нефункціональних вимог (нефункціональне тестування – щоб визначити, чи відповідає програма додаткові очікування щодо продуктивності, безпеки, відповідності тощо), вона передбачає такі практики, як аналіз статичного коду, тестування безпеки, тестування продуктивності тощо. Тести повинні бути розроблені для забезпечення якнайшвидшого виявлення (або запобігання) ризиків, які є найважливішими для бізнесу чи організації, яка випускає програмне забезпечення.

Команди часто виявляють, що для того, щоб набір тестів міг працювати постійно та ефективно оцінював рівень ризику, необхідно перевести фокус з тестування на графічний інтерфейс на тестування API.

Випробування проводяться під час або поряд із постійною інтеграцією (CI) – принаймні щодня. Для команд, які практикують безперервну доставку, тести зазвичай виконуються багато разів на день, кожного разу, коли додаток оновлюється до системи контролю версій.

В ідеалі всі тести виконуються в усіх невиробничих тестових середовищах. Для забезпечення точності та послідовності тестування слід проводити в найбільш повному, виробничому подібному середовищі. Стратегії підвищення стабільності тестового середовища включають програмне забезпечення для віртуалізації (для залежностей, якими може керувати ваша організація та зображення), віртуалізацію послуг (для залежностей, що виходять за рамки вашої сфери контролю або непридатні для зображень) та управління тестовими даними.

Головні вимоги для використання безперервного тестування:

– тестування повинно бути співпрацею з розвитку, забезпечення якості та операцій – узгоджених з пріоритетами напряму бізнесу – в рамках координованого, якісного та якісного процесу;

– тести повинні бути логічно-компонентними, поступовими та повторюваними; результати повинні бути детермінованими та значущими;

– усі випробування потрібно проводити в якийсь момент конвеєра, але не всі тести потрібно виконувати весь час;

– усуньте дані випробувань та обмеження середовища, щоб тести могли постійно та послідовно виконуватись у виробничих середовищах; щоб мінімізувати помилкові позитиви, мінімізувати обслуговування тестів та ефективніше перевірити випадки використання в сучасних системах з багатоядерними архітектурами, команди повинні робити акцент на тестуванні API.

Мета постійного тестування – застосувати «екстремальну автоматизацію» до стабільних, схожих на виробництво тестових середовищ. Автоматизація є важливою для безперервного тестування. Але автоматизоване тестування не те саме, що безперервне тестування.

Автоматизоване тестування передбачає автоматизоване виконання CI будь-якого набору тестів, який накопичила команда. Перехід від автоматизованого тестування до безперервного тестування передбачає виконання набору тестів, спеціально розроблених для оцінки ділових ризиків, пов'язаних з кандидатом на випуск та регулярно виконувати ці випробування в умовах стабільних виробничих тестових середовищ. Деякі відмінності між автоматизованим та безперервним тестуванням:

– при автоматизованому тестуванні помилка тесту може вказувати на що завгодно, від критичної проблеми до порушення тривіального стандарту іменування. При постійному тестуванні невдача тесту завжди вказує на критичний бізнес-ризик;

– при постійному тестуванні тест усувається з допомогою чіткого робочого процесу для визначення пріоритетності дефектів перед бізнес-ризиками та вирішення найважливіших;

– при постійному тестуванні щоразу, коли виявляється ризик, відбувається процес виявлення всіх подібних дефектів, які, можливо, вже були введені, а також відбувається запобігання повторенню цієї ж проблеми в майбутньому.

2.4 Переваги взаємної інтеграції CI/CD/CT

Постійна інтеграція та безперервне розгортання має багато переваг. Нижче наведено деякі з них:

– знижується ризик. Коли ви частіше розгортаєте код, це зменшує ризик провалу проекту, над яким ви працюєте, оскільки ви маєте можливість легко виявляти помилки. Це прискорює виробництво;

– біль продуктивна комунікація. Коли ви маєте добре налаштований процес CI/CD/CT, обмін кодом стає більш легким і помітним. Це покращує співпрацю між членами команди а, отже, кращу комунікацію між командами;

– швидші ітерації. Коли реліз коду відбувається часто, розрив між кодом у завершеним кодом, який вже використовується та кодом який ще у розробці стає значно меншим. Це дозволяє краще орієнтуватись у змінах, які роблять розробники при кожній ітерації;

– зменшена ручна праця. Оскільки весь процес автоматизований за допомогою CI/CD/CT, ручне зусилля для інтеграції інструментів та тестування зменшується, а отже, забезпечується плавний потік коду до різних середовищ;

– автоматизоване тестування. За рахунок автоматизацій, зменшується час на тестування програмного коду, адже безперервне тестування передбачає саме автоматизації тестів; також зменшується ризик людської помилки під час тестувань;

– швидший зворотній зв'язок. Настроювання CI/CD/CT не тільки допомагає розробникам, але й допомагає керівництву приймати рішення та діяти на виробничих змінах. Оскільки процес випуску стає набагато швидшим, менеджеру набагато легше бачити зміни та приймати швидкі бізнес-рішення.

Тепер ми зрозуміли різницю між постійною інтеграцією, безперервним розгортанням та постійною доставкою. Перейдемо до іншого терміна, який називається Pipeline (з англ. трубопровід). Рис. 2.4 дуже добре дозволяє зрозуміти значнення терміну.

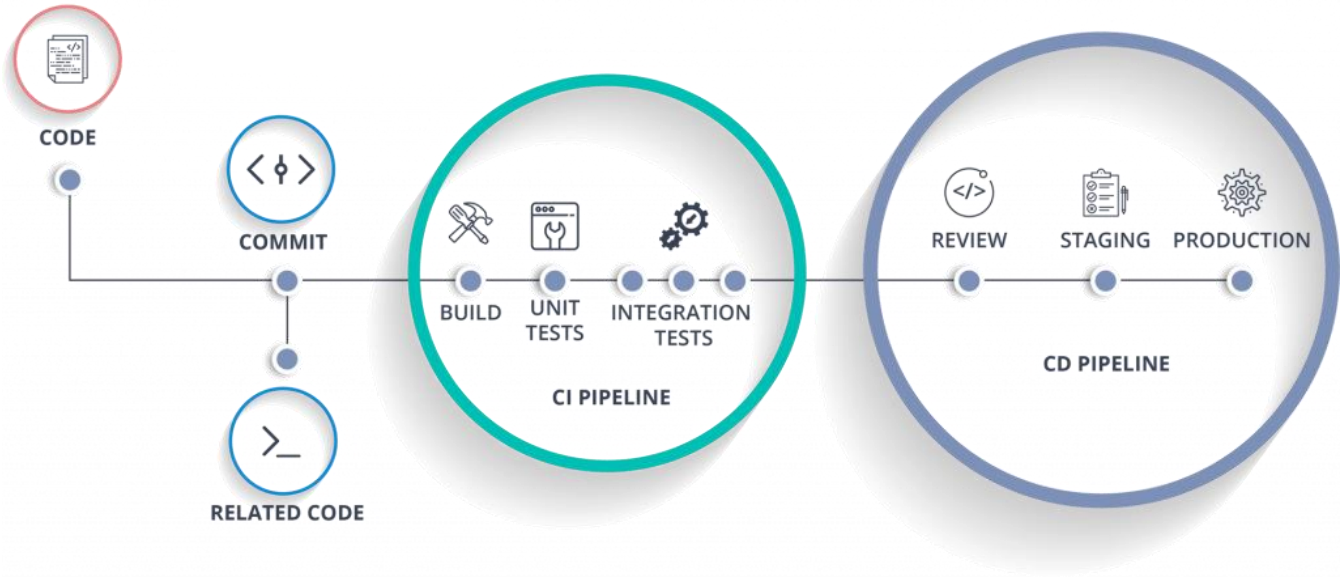


Рисунок 2.4 – Структурна схема «трубопроводу» розробки сучаного ПЗ

Pipeline – дуже важливий термін у системі CI-CD. Це послідовність кроків, які виконуються при внесенні змін до проекту. Етапами можуть бути вихідний код, збірка, база даних, артефакти, оточення, розгортання або конфігурація.

Коли код вводиться, CI Pipeline гарантує, що запускається збірка та тестування (Постійне тестування). Після того, як він передається з CI Pipeline, він переміщується до CD Pipeline, де його переглядають та передають в інші середовища та нарешті на виробництво.

3 АНАЛІЗ ІСНУЮЧИХ ІНСТРУМЕНТІВ ПОБУДОВИ CI/CD/CT

3.1 Визначення критеріїв аналізу

Перехід на методи Agile та DevOps дозволив скоротити цикли програмування до тижнів та запровадив стабільну інтервал доставки. Сьогоднішня практика безперервної інтеграції (CI) впроваджує оновлення програми ще швидше, протягом днів або годин. Це результат частого надсилання коду до спільного сховища, щоб розробники могли легко відслідковувати дефекти за допомогою автоматизованих тестів, а потім виправити їх якнайшвидше. У цьому розділі я детально поясню переваги та недоліки використання різних інструментів впровадження CI/CD/CT у процес розробки, як підходити до вибору та які виклики очікувати на цьому шляху.

Перед усім хочу відмітити, що розробники інструментів такого типу завжди намагаються зробити продукт, який буде включати в себе саме можливість об’єднати конфігурацію безперервної інтеграції, доставки, розгортання, та тестування в одному місці. Тому слід розуміти, що всі інструменти розглянуті у даному розділі можна вважати як набір готових рішень для настройки усього pipeline проекту.

Спираючись на досвід використання багатьох інструментів такого роду та широкий обсяг вивченої теоретичної інформації, я прийняв рішення використовувати наступні критерії для порівняльного аналізу:

– можливості хостингу. Програмні засоби відрізняються своїм управліннямінфраструктурою. Хмарні інструменти розміщуються на стороні постачальника, вимагають мінімальної конфігурації та можуть бути налаштовані за потребою залежно від ваших потреб. Є також власні рішення. Відповідальність за їх розгортання та підтримку покладається виключно на ваші плечі, а точніше на вашу внутрішню команду DevOps. Хоча локальні послуги виграють гнучкість процесу нарощування процесів, хостингові рішення позбавляють від труднощів із налаштуваннями, пропонуючи більшу масштабованість.

– зручність використання. Деякі інструменти можуть зробити процес збирання набагато простішим, ніж інші, з огляду на їх чіткі та зрозумілі GUI та UX. Добре розроблений інтерфейс може заощадити ваш час на етапі вбудування.

– інтеграція та підтримка програмних забезпечень. Визначення можливості інструменту CI інтегруватися з зовнішніми сервісами та іншими інструментами. Приклади інтеграції можуть включати програмне забезпечення для управління проектами (наприклад, Jira), інструменти заповнення інцидентів (наприклад, PagerDuty та Bugzilla), інструменти статичного аналізу, інструменти дотримання законодавства тощо. Інструмент CI повинен бути досить гнучким для підтримки різних типів інструментів побудови (Make, Shell Scripts, Ant, Maven, Gradle) та програмне забезпечення для управління версіями або VCS (Subversion, Perforce, Git) тощо.

– підтримка контейнерізації. Маючи плагін або конфігурацію для розгортання контейнерних інструментів, таких як Kubernetes і Docker, полегшує підключення інструмента CI до цільового середовища програми.

– бібліотека коду багаторазового використання. Бажано, коли інструмент має рішення з різноманітним загальнодоступним сховищем плагінів, з відкритим вихідним кодом та можливістю власноруч доповнювати власними напряцьованими рішеннями.

Далі, на основі досліджень ринку та незалежного опитування користувачів, було обрано декілька найбільш популярних інструментів, які будуть послідовно проаналізовані та порівняні на основі вище наведених критеріїв: Jenkins, TeamCity, Bamboo, Travis CI, Circle CI, GitLab CI, GoCD.

3.2 Огляд визначених інструментів

Jenkins: найпопулярніше та найбільш адоптоване рішення.

Jenkins – проект з відкритим кодом, написаний на Java, який працює на Windows, macOS та інших операційних системах, схожих на Unix. Це безкоштовний, що підтримується спільнотою, і може бути вашим інструментом першого вибору для постійної інтеграції. Переважно розгорнута локально, Jenkins також може працювати на хмарних серверах. Її інтеграція з Docker та Kubernetes використовує переваги контейнерів та виконує ще більш часті випуски. Jenkins можна встановити за допомогою нативних системних пакетів, Docker або навіть запустити автономно на будь-якій машині із встановленим середовищем виконання Java.

Вимоги до системи.

– сервер з встановленою операційною системою Windows, macOS або Unix;
– 256 Мб оперативної пам’яті, хоча рекомендується більше 512 МБ;
– 10 Гб дискового простору;
– встановлена Java 8.

Основні переваги:

– ніяких витрат не потрібно. Дженкінс – це безкоштовний інструмент CI, який може заощадити гроші на проекті;

– безмежні інтеграції. Jenkins може інтегруватися практично з будь-якою зовнішньою програмою, що використовується для розробки програм. Це дозволяє використовувати контейнерні технології, такі як Docker і Kubernetes нестандартно. Рецензенти G2 Crowd заявляють: «Не існує кращого інструменту для інтеграції ваших сховищ та баз коду з інфраструктурою розгортання»;

– Jenkins має доступну багату бібліотеку плагінів: Git, Gradle, Subversion, Slack, Jira, Redmine, Selenium, Pipeline. Плагіни Jenkins охоплюють п'ять областей: платформи, інтерфейс користувача, адміністрування, управління вихідним кодом і, найчастіше, управління складанням. Хоча інші інструменти CI надають подібні функції, їм не вистачає всебічної інтеграції плагінів, яку має Jenkins. Більше того, спільнота Jenkins заохочує своїх користувачів розширювати функціональність новими можливостями, надаючи навчальні ресурси;

– активна спільнота. Спільнота Jenkins проводить екскурсію для ознайомлення з основами та вдосконаленими навчальними посібниками для більш досконалого використання інструменту. Вони також проводять щорічну конференцію «DevOps World | Світ Jenkins»;

– розподіл складових і випробувальних навантажень на декількох машинах. Jenkins використовує архітектуру Master-Slave, де майстер є головним сервером, який відстежує рабів – віддалені машини, що використовуються для розповсюдження складання програмного забезпечення та тестових навантажень.

Головні недоліки:

– документація не завжди є достатньою. Наприклад, йому не вистачає інформації про створення трубопроводів. Це додає трудомістких завдань до списку, оскільки інженерам доводиться працювати над ними;

– поганий інтерфейс користувача. Його інтерфейс здається трохи застарілим, оскільки він не відповідає сучасним принципам дизайну. Відсутність пробілів робить погляди переповненими та заплутаними. Багато функцій та

значків прогресу є дуже неякісними(пікселізованими) та не оновлюються автоматично, коли завдання закінчуються;

– потрібно вручну відстежувати сервер Jenkins та його рабів, щоб зрозуміти взаємозалежності між плагінами та час від часу їх оновлювати;

– загалом, Jenkins служить найкращим чином для великих проектів, де вам потрібно багато налаштувань, які можна виконати за допомогою різних плагінів. Тут ви можете змінити майже все, але цей процес може зайняти деякий час. Однак якщо ви плануєте якнайшвидше розпочати роботу із системою CI, розгляньте різні варіанти.

TeamCity: ще один великий гравець CI.

TeamCity – це інструмент безперервної інтеграції, який допомагає створювати та розгортати різні типи проектів. TeamCity працює в середовищі Java та інтегрується з Visual Studio та IDE. Інструмент можна встановлювати як на серверах Windows, так і на Linux, підтримує .NET та проекти з відкритим стеком.

TeamCity 2019.1 забезпечує новий інтегральний інтерфейс та вбудовану інтеграцію GitLab. Він також підтримує запити на потягування сервера GitLab та Bitbucket. Випуск включає аутентифікацію на основі токенів, виявлення, звітування про тести Go та запити на AWS Spot Fleet.

Команди часто вибирають TeamCity для великої кількості аутентифікацій, розгортання та тестування функцій нестандартно, плюс підтримка Docker. Це крос-платформа, підтримує всі останні версії Windows, Linux та macOS і працює з Solaris, FreeBSD, IBM z/OS та HP-UX. TeamCity працює відразу після встановлення, додаткові налаштування чи налаштування не потрібні. Він може похвалитися рядом унікальних функцій, таких як докладні звіти про історію, миттєвий зворотній зв'язок про тест-помилки та повторне використання налаштувань, тому вам не доведеться дублювати код [1].

Цінові моделі. TeamCity пропонує безкоштовну версію з повним доступом до всіх функцій продукту, але він обмежений 100 конфігураціями побудови та трьома агентами побудови. Додавання ще одного агента побудови та 10 конфігурацій збірки наразі коштує 299 доларів США. TeamCity пропонує 60-денну пробну хмару, яка обійде локальну установку.

Є також платне корпоративне видання. Його ціна змінюється в залежності від кількості агентів, що входять. TeamCity дає 50 відсотків знижки на стартапи та безкоштовні ліцензії на проекти з відкритим кодом.

Основні переваги:

– підтримка .NET. TeamCity інтегрується з .NET інструментами краще, ніж будь-який інший інструмент CI тут. У TeamCity входить багато важливих інструментів .NET, таких як аналіз покриття коду, кілька фреймворків тестування .NET та аналіз статичного коду;

– широка підтримка VSC. TeamCity дозволяє створювати проекти лише з URL-адреси сховища VCS. Інструмент підтримує всі популярні VCS: AccuRev, ClearCase, CVS, Git, Gnu bazaar, Mercurial, Perforce, Borland StarTeam, Subversion, Team Foundation Server, SourceGear Vault, Visual SourceSafe та IBM Rational ClearCase. На рис. 3.1 схематично зображено актуальну інформацію щодо зовнішніх інтеграцій TeamCity.

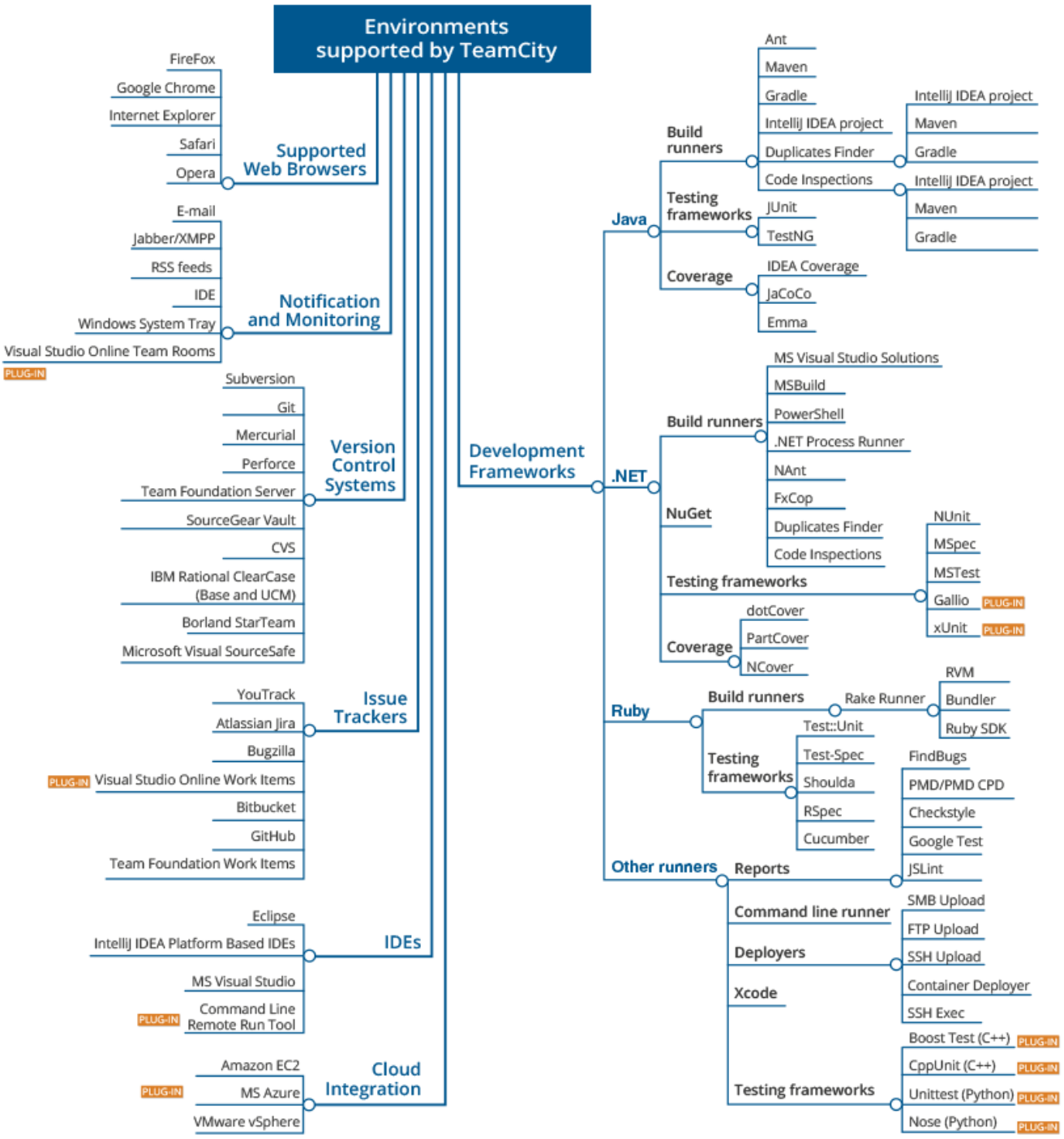


Рисунок 3.1 – Схема зовнішніх інтеграцій TeamCity

– компетентні документи. Загальну функціональність можна легко зрозуміти, переглянувши надане керівництво користувача, яке є ретельним та обширним;

– простота налаштування. TeamCity легко налаштувати та готовий працювати відразу після встановлення. Він надає хороший набір функцій, що не випускаються для створення проекту;

– вбудовані функції. TeamCity надає гарний набір функцій, що не існують, для створення проекту: детальні звіти про збірки, збої та будь-які додаткові зміни, контроль джерел та інструменти побудови ланцюга. Однією з найкращих особливостей TeamCity є «Опублікувати артефакти», яка дозволяє розгортати код або навіть будувати безпосередньо в будь-якому середовищі. Він показує хід складання на кожному кроці та кількість тестів, що залишилися до завершення збірки. Це також дозволяє повторно виконувати будь-які невдалі тести відразу після виконання ночі, тому вам не доведеться витрачати час на це наступного ранку.

Недоліки TeamCity:

– складна крива навчання. Хоча TeamCity добре відомий своїм візуально-естетичним інтерфейсом, він все ще може бути трохи складним і непосильним для новачків, пропонуючи при цьому широкий спектр параметрів конфігурації. Можливо, розробникам може знадобитися серйозне дослідження, перш ніж вони будуть готові використовувати інструмент у виробництві;

– процес оновлення вручну. Перехід від однієї основної версії до іншої – це довгий процес, який потрібно виконати вручну на вашому сервері.

Bamboo: найкраща інтеграція з продуктами Atlassian.

Окрім сприяння інтеграції, Bamboo має функції розгортання та управління випусками. У той час як Bamboo має менше варіантів, які не випускаються «з коробки», він інтегрується в оригінали з іншими продуктами Atlassian: Bitbucket, Jira і Confluence. Така ж інтеграція потребує просто величезної кількості плагінів Jenkins. Як і попередні два інструменти, Bamboo працює на Windows, Linux, Solaris та macOS. Для тих, хто працює з Bamboo на Linux, офіційна документація наполягє на створенні спеціального користувача, щоб запобігти будь-якому потенційному зловживанню.

Bamboo можна спробувати протягом 30 днів. Після цього його варіанти користувальницького рівня включають необмежену кількість локальних агентів та

10 завдань та масштабують до 1000 віддалених агентів, ціна яких становить від 10 до 126500 доларів США відповідно. Включений 12-місячний термін обслуговування Цей період можна збільшити вдвічі або втричі за більше грошей. Програмне забезпечення Atlassian безкоштовне для будь-якого проекту з відкритим кодом, який відповідає їх визначеним критеріям.

Головні переваги:

– Bitbucket Pipelines. Після того, як Atlassian відмінив Бамбукову хмару в 2016 році, інструмент став доступний лише на місцях. Однак Atlassian випустив Bitbucket Pipelines – хмарну альтернативу, вбудовану в Bitbucket – рішення для управління сховищами Git. Трубопроводи можуть бути повністю інтегровані з Bamboo. Як спеціально налаштована система, Bitbucket Pipelines забезпечує автоматичну побудову, тестування та розгортання коду на основі файлу конфігурації у сховищі. Використовуючи потужність Docker, Bitbucket Pipelines пропонує дуже ефективні та швидкі побудови. Врешті-решт сервер Bamboo все ще доступний для попередньої установки та може розміщуватись у хмарному контейнері або VM;

– кілька методів повідомлення. Bamboo Wallboard показує результати побудови на спеціальному моніторі та надсилає результати збирання у вашу поштову скриньку або у вашу кімнату чатів для розробників (наприклад, HipChat, Google Talk);

– багата і проста інтеграція. Bamboo підтримує більшість основних стеків технологій, таких як CodeDeply, Ducker, Maven, Git, SVN, Mercurial, Ant, AWS, Amazon S3 Buckets. Крім того, він ідентифікує нові гілки в цих технологіях і автоматично застосовує налаштування тригерів і змінних. Функція дозволів на бамбукове середовище на середовище дозволяє розгортатися в їх середовищі;

– документація та супровід. Документація на бамбук багата і детальна. Atlassian надає кваліфіковану підтримку. Однак розмір спільноти далеко не охоплює користувачів Jenkins.

Головні недоліки Bamboo:

– погана підтримка плагінів. На відміну від Jenkins та TeamCity, Bamboo не підтримує так багато плагінів. На даний момент у сховищі Atlassian перелічено лише 208 додатків;

– складний перший досвід роботи. Деякі користувачі скаржаться, що процес налаштування першого завдання розгортання не є досить інтуїтивним, і

для того, щоб зрозуміти всі різні варіанти та способи їх використання, потрібен час.

З Bamboo буде добре працювати людям, які шукають інтеграції з продуктами Atlassian стеку.

Travis CI: зріле рішення CI з простою інтеграцією GitHub.

Travis CI – сервіс CI, яка використовується для створення та тестування проектів. Travis CI автоматично виявляє нові комісії, зроблені та переміщені до сховища GitHub. І після кожної нової передачі коду, Travis CI будуватиме проект і відповідно проводити тести.

Інструмент забезпечує підтримку багатьох конфігурацій побудови та мов, таких як Node, PHP, Python, Java, Perl тощо.

Перші 100 збірок безкоштовні. В іншому випадку є чотири плани ціноутворення для хобі-проектів (69 доларів США на місяць), а також для невеликих, зростаючих та великих команд (від 129 до 489 доларів США на місяць). Вони відрізняються за кількістю одночасних завдань, які можна виконати. Ви також можете зв’язатися з Travis CI, щоб отримати індивідуальний план.

Головні переваги Travis CI:

– просте налаштування та налаштування. Travis CI не потребує встановлення – ви можете розпочати тестування, просто зареєструвавшись та додавши проект. Програмне забезпечення можна налаштувати за допомогою простого файлу YAML, який ви розміщуєте в кореневій директорії проекту розробки. Користувацький інтерфейс дуже чуйний, більшість користувачів кажуть, що це зручно для моніторингу збірок;

– пряме з'єднання з GitHub. Travis CI бездоганно працює з популярними системами управління версіями, зокрема GitHub. Більше того, цей інструмент безкоштовний для проектів з відкритим кодом GitHub. Інструмент CI реєструє кожен поштовх до GitHub і автоматично будує гілку за замовчуванням;

– резервне копіювання недавньої збірки. Щоразу, коли ви запускаєте нову збірку, Travis CI клонує ваше сховище GitHub у нове віртуальне середовище. Таким чином у вас завжди є резервна копія.

Недоліки використання Travis CI:

– немає вбудованої безперервної доставки та розгортання. На відміну від інших інструментів CI у списку, програма Travis CI не дозволяє здійснювати постійну доставку;

– хостинг лише для GitHub Оскільки Travis пропонує лише підтримку проектів, розміщених у GitHub, команди, які використовують GitLab або будь-яку іншу альтернативу, змушені покладатися на інший інструмент CI;

– підводячи підсумок, Travis CI – найкраще рішення для проектів з відкритим кодом, які потребують тестування в різних середовищах. Крім того, це правильний інструмент для малих проектів, головна мета – якнайшвидше розпочати інтеграцію [2].

CircleCI: простий та корисний інструмент CI для проектів на ранній стадії.

CircleCI – це інструмент CI / CD, який підтримує швидку розробку та публікацію програмного забезпечення. CircleCI дозволяє здійснювати автоматизацію в конвеєрі користувача, від створення коду, тестування до розгортання.

Ви можете інтегрувати CircleCI з GitHub, GitHub Enterprise та Bitbucket для створення збірок, коли вводяться нові рядки коду. CircleCI також розміщує безперервну інтеграцію в хмарному варіанті або працює за брандмауером приватної інфраструктури.

CircleCI має безкоштовну 2-тижневу пробну версію macOS, яка дозволяє будувати і Linux, і macOS. Безкоштовний пакет Linux включає один контейнер. Додаючи більше контейнерів (50 доларів США на місяць), ви також можете вибрати рівень паралелізації (від однієї до 16 одночасних завдань). Плани MacOS коливаються від 39 доларів США / місяць для 2x одночасності до 249 доларів США на місяць для 7-разової одночасності та підтримки електронної пошти.

Якщо ваше підприємство має спеціалізовані потреби, ви можете зв’язатися безпосередньо з CircleCI, щоб обговорити план цін на основі особистого використання. Для проектів з відкритим кодом CircleCI виділяє чотири безкоштовні Linux-контейнери, а також план macOS Seed безкоштовно за 1x паралельно.

Переваги CirleCI:

– простий інтерфейс користувача. CircleCI визнаний своїм зручним інтерфейсом для управління збірками / робочими місцями. Його веб-додаток на одній сторінці чистий і легкий для розуміння;

– якісна підтримка клієнтів. Члени спільноти StackShare виділяють швидку підтримку CircleCI: вони відповідають на запити протягом 12 годин;

– CircleCI запускає всі типи програмних тестів, включаючи веб, мобільні та контейнерні середовища;

– розробка встановлення вимог та сторонні залежності. Замість того, щоб встановлювати середовища, CircleCI може брати дані з декількох проектів, використовуючи детальні параметри відмітки.

Недоліки використання CircleCI:

– надмірна автоматизація. CircleCI змінює середовище без попередження, що може бути проблемою. З Дженкінсом вона буде запускати зміни лише тоді, коли користувач проінструктує;

– немає кешування зображень Docker. Наприклад, у Jenkins ми можемо кешувати зображення Docker (docker images) за допомогою приватного сервера; з CircleCI це неможливо;

– немає тестування в ОС Windows. CircleCI вже підтримує побудову більшості програм, які працюють на Linux, не кажучи вже про iOS та Android. Однак інструмент CI ще не дозволяє будувати та тестувати в середовищі Windows.

Загалом, якщо вам потрібно швидко побудувати щось, а гроші – це не проблема, CircleCI з його параметрами паралелізації – це ваш інструмент переходу до CI.

GitLab CI.

GitLab – це набір інструментів для управління різними аспектами життєвого циклу розробки програмного забезпечення. Основним продуктом є веб-менеджер репозиторіїв Git з такими функціями, як відстеження проблем, аналітика та Wiki (велика документація з детальним поясненням).

GitLab дозволяє запускати збірки, запускати тести та розгортати код з кожним фіксуванням або натисканням. Ви можете створювати завдання у віртуальній машині, контейнері Docker або на іншому сервері.

Gitlab CI дозволяє користувачам:

– переглядати, створювати та керувати кодом та даними проектів за допомогою інструментів розгалуження;

– розробляти та керувати кодом та даними проектів з єдиної розподіленої системи управління версіями, що забезпечує швидку ітерацію та доставку кінцевих продуктів;

– допомагає командам доставки повною мірою сприймати CI шляхом автоматизації побудови, інтеграції та перевірки вихідних кодів;

– забезпечує сканування контейнерів, статичне тестування безпеки додатків (SAST), динамічне тестування безпеки додатків (DAST) та сканування залежностей для доставки захищених програм разом із дотриманням ліцензії;

– допомагає автоматизувати та скорочувати випуски та доставку додатків;

– є можливість використовувати як встанвлений на сервер додаток, так і «як сервіс» у хмарному середовищі;

– відкритий вихідний код.

Go CD:

GoCD – це інструмент з відкритим кодом для створення та випуску програмного забезпечення, яке підтримує сучасну інфраструктуру на CI / CD.

Go – одне з останніх творінь ThoughtWorks, яке безкоштовно, за винятком комерційних ліцензій.

Він доступний для Windows, Mac та Linux і працює в системі трубопроводів (pipelines), які допомагають тримати робочий процес. Трубопроводи допомагають підключити всіх зацікавлених сторін до одного сценарію, покращуючи комунікацію на всіх фронтах та дозволяючи краще розуміти платформу.

Замість того, щоб покладатися на велику завантаженість, трубопроводи розбивають завдання на менші частини з командами, призначеними для кожної секції. Це врешті-решт призводить до змін, усуваючи вузькі місця та спонукаючи до виконання паралельних завдань.

Він добре створений для обробки складних сценаріїв і безкоштовний при оплаті за підтримку. Також має дуже обширний список плагінів [3].

3.3 Порівняльний аналіз інструментів побудови CI/CD/CT

Виходячи з вище наведених теоретичних відомостей щодо кожної з досліджуваних інструментів та інформації з офіційної документації, в рамках даної роботи було побудовано порівняльну таблицю, яка наглядно відображає усі критерії, які ми обрали для аналізу та приблизну оцінку цих критерій.

Оцінка критерію «плагіни» виставлялась за п’ятибальною шкалою, виходячи з кількості існуючих плагінів, їх корисності та відгуків спільноти.

On Premise (від англ. на передумові) – мається на увазі можливість встановлювати власний сервер з додатком та адміністрування власноруч.

Таблиця 3.1 – Порівняльна таблиця інструментів реалізації CI/CD/CT

	Jenkins	TeamCity	Bamboo	Travis CI	Circle CI	GitLab	Go CD
Ціна	Безкоштовно	$299-1999	$10-800	$69-489	$50-3150	$0-99	Безкоштовно
Операційна система	Windows, Unix-like, macOS	Windows, Linux, macOS, Solaris, FreeBSD	Windows, Linux, MacOS, Solaris	Linux, MacOS	Linux, IOS, Andoird	Only Unix-like OS	Windows, Linux, MacOS
Хостинг	On Premise*/cloud	On premise	On premise/BitBucket Cloud	On premise/Cloud	Cloud	On Premise/Cloud	On premise/Cloud
Підтримка контейнеризації	+	+	+	+	+	+	+
Плагіни	5	4	2	4	3	3	4
Документація та підтримка	Середня	Середньо	Добра	Погана	Добра	Добра	Середня
Складність використання	Легко	Середньо	Середньо	Легко	Легко	Легко	Легко
Ціль використання	Універсальна система	Для комерційних проектів	Для інтеграції з іншими Atlassain продуктами	Для невеликих проектів та стартапів	Для великих бюджетів та швидких проектів	Для невеликих проектів з визначеною метою	Універсальна система

Для більш наглядної оцінки аналізу було побудовано декілька графіків, використовуючи програмне забезпечення Microsoft Excel 2012.

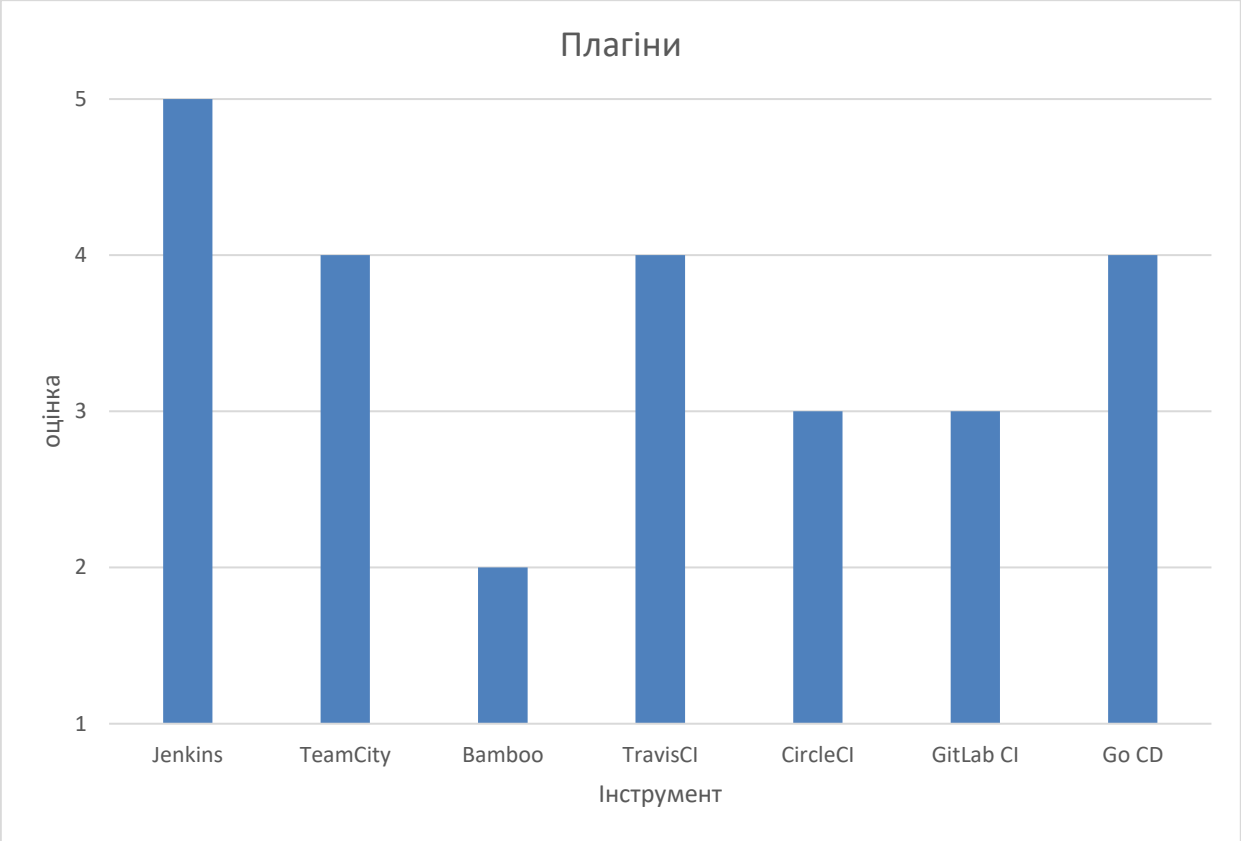


Рисунок 3.2 – Порівняльна діаграма якості та підтримки використання CI інструментами додаткових плагінів

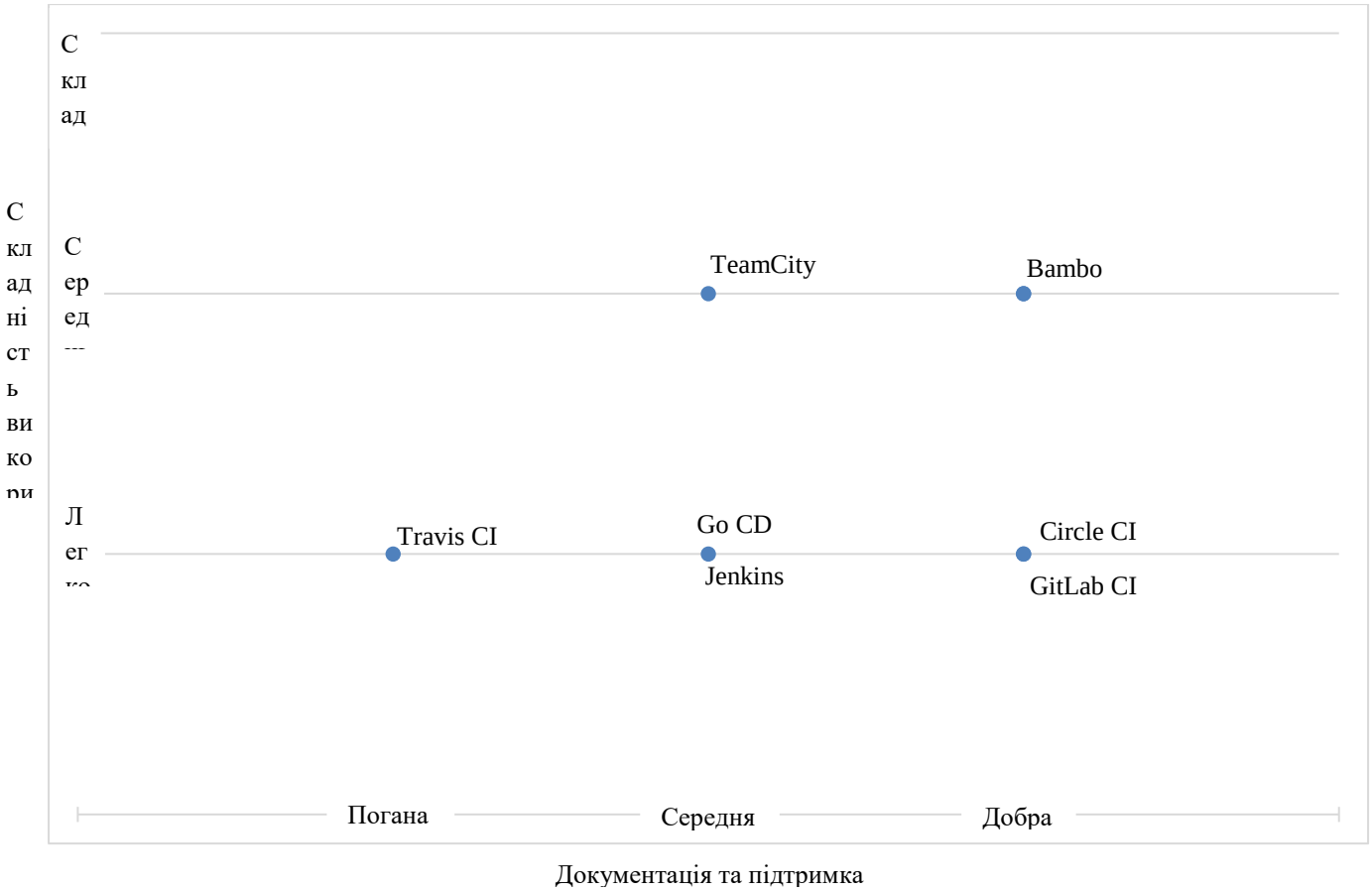


Рисунок 3.3 – Графік на основі аналізу документації та складності використання інструменту

В якості підсумку огляду загальнодоступних публікацій можна навести наступну діаграму відомого видання G2 [5].

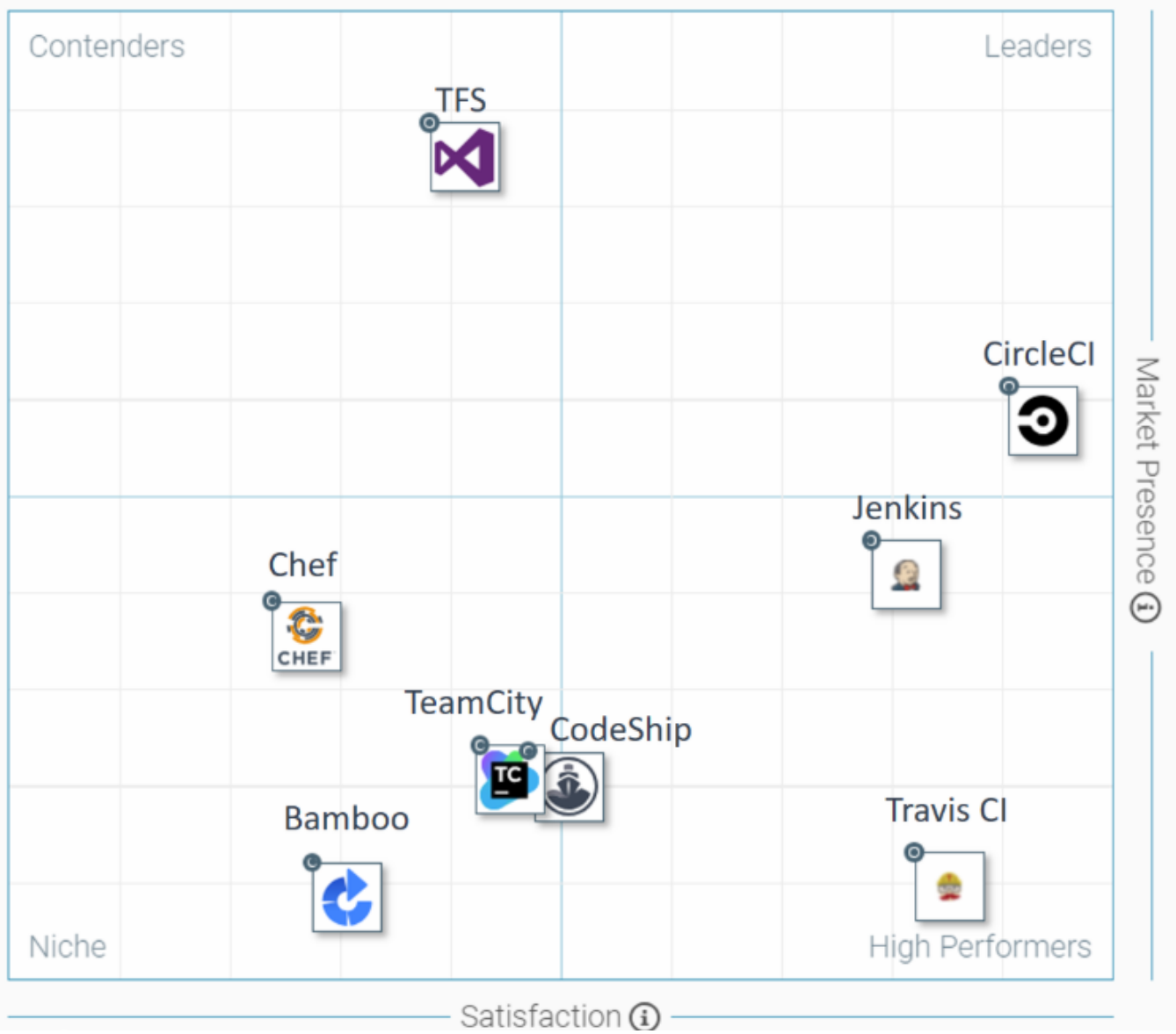


Рисунок 3.4 – Графік розділення CI/CD/CT інструментів на лідерів, високо продуктивних, нішевих та набираючих популярність

Проаналізувавши результати досліджень видання G2 (рис 3.4), можна зрозуміти, що деякі розглянуті інструменти є дуже вузько направленими або нішевими, у тай час як безумовними лідерами є Circle CI та Jenkins. Travis CI за даними G2 виявився найбільш продуктивним.

4 ПРОЕКТУВАННЯ КОМЕРЦІЙНОГО ПРОЕКТУ НА ОСНОВІ ОБРАНОГО CI/CD/CT ІНСТРУМЕНТУ

4.1 Вимоги до організації комерційного проекту

Даний розділ дипломної роботи присвячений впровадженню CI/CD/CT інструментів в існуючий проект на основі вимог описаних у технічному завданні. Необхідно провести аналіз виграшів та надати рекомендацій для подальшої реалізації з метою підвищення загальної ефективності процесу розробки та підтримки проекту.

Задача була запропонована моїм дипломним керівником та максимально наближена до реальних умов імплементації CI/CD/CT інструмента у потенційно існуючий проект. Також у рамках виконання задачі будуть використовуватись допоміжні інструменти, без яких побудувати автоматизовану та працюючу інфраструктуру неможливо, або дуже проблематично, але основна ідея роботи, у тому, щоб показати саме налаштування безперервної інтеграція, доставки, розгортання та тестування коду.

Опис задачі.

Замовник програмного забезпечення поставив за мету побудувати автоматизований CI/CD/CT для існуючого проекту X. Технічна команда проекту складається с 10 розробників та двох автоматизаторів тестування. Мова програмування, яка використовується для розробки – Java. Вихідний код зберігається у системі контролю версій Git, у приватному репозиторії GitHub. Для збереження файлів використовується інструмент JFrog Artifactory. Команда працює за методом Scrum, який було описано у розділі 2. Кожні два тижні клієнт бажає бачити нову версію працюючого додатку з мінімум непрацюючого функціоналу, та на 99% покритого автоматизованим тестуванням. Також замовник надає декілька серверів з встановленою UNIX операційною системою та бюджет для покупки платної ліцензії до 50 доларів США. Його основна вимога щодо вибору інструменту – можливість використовувати його на передумові (встановивши на власні сервер). Деталі побудови автоматизованого CI/CD/CT замовник програмного забезпечення не уточнює.

4.2 Аналіз вимог клієнта

Вивчивши та проаналізувавши вимоги та побажання замовника ПЗ, можна відразу відкинути декілька поганих варіантів. Перш за все, замовник не має можливості платити більш ніж 50 доларів США за платний продукт, отже у цьому випадку логічно буде відмовитись від Travis CI та TeamCity, мінімальна вартість яких складає 299 та 69 доларів США відповідно. Дивлячись на інші вимоги можна побачити, що замовник дуже хотів би мати власний сервер з встановленим інструментом для реалізації CI/CD/CT та навіть надає віртуальні машини з UNIX подібними операційними системами. З цього випливає, що використання Circle CI, нам теж дуже підходить, так як може використовуватись тільки у хмарі, тобто «як сервіс». Залишаються GitLab CI, Jenkins, Go CD та Bamboo. Переглянувши більш уважно зроблений у 3 розділі аналіз, можна зробити висновок, що використання Bamboo доцільне лише коли потрібна гнучка та швидконалаштовуєма інтеграція з продуктами Atlassian стеку. Нам це не потрібно. Вибираючи з тих інструментів, що залишились, виходячи з власного досвіду та спілкування з більш досвідченими спеціалістами у цій галузі, було прийняте рішення використовувати саме Jenkins, як найбільш гнучкий та стабільний інструмент реалізації CI/CD/CT з дуже великим обсягом доступних, безкоштовних плагінів, непоганої документацією та невеликим порогом входу.

4.3 Інсталювання Jenkins серверу

Для того щоб використовувати Jenkins його потрібно встановити на виділені віртуальні машини з UNIX-подібними операційними системами. Перш за все потрібно визначитись з якою саме операційною системою (ОС) ми маємо справу. Для цього підключимось до терміналу любим зручним способом та перевіримо встановлену ОС командою cat /etc/*release , як показано на рис. 4.1.

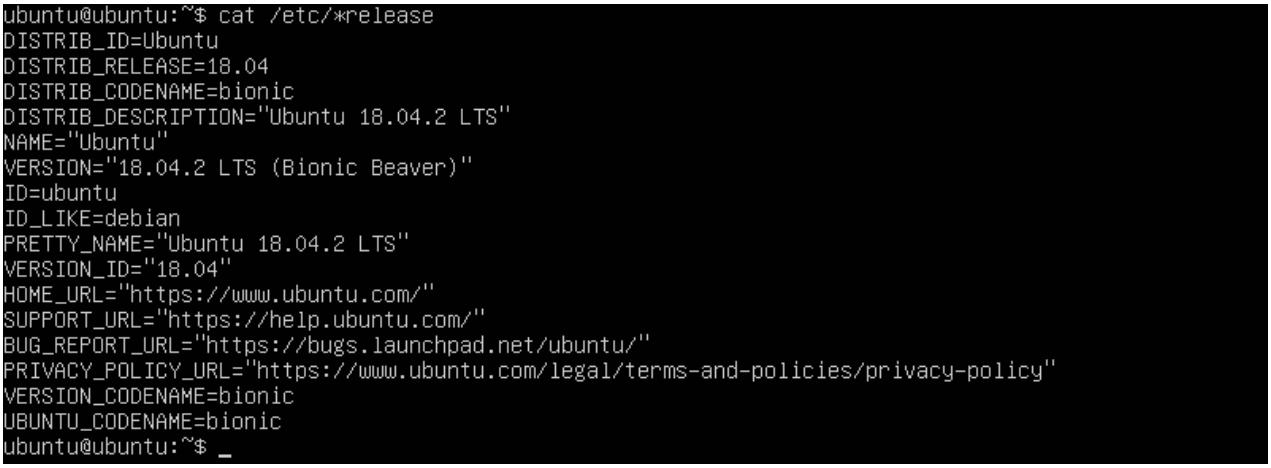
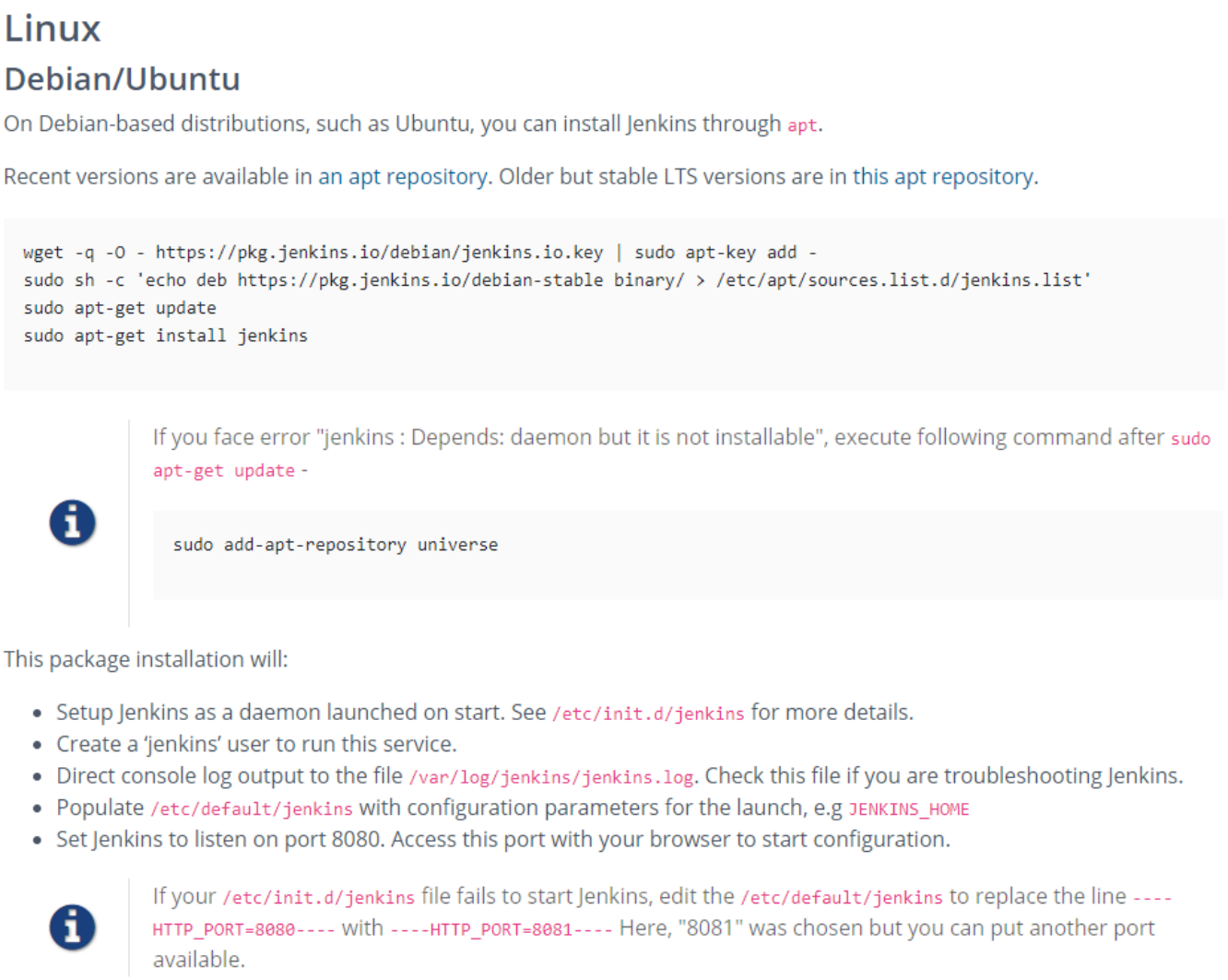


Рисунок 4.1 – Перевірка встановленої ОС

Далі для роботи Jenkins, як вже було вказано у розділі 3, потрібна встановлена Java 8, або новіша. Перевіримо це, та встановимо, якщо це необхідно, використовуючи офіційну документацію [11].

Дізнавшись встановлену ОС, ми використовуємо офіційну технічну документацію до Jenkins [4] для того щоб встановити сервер.



Linux

Debian/Ubuntu

On Debian-based distributions, such as Ubuntu, you can install Jenkins through `apt`.

Recent versions are available in an apt repository. Older but stable LTS versions are in this apt repository.

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -
sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
sudo apt-get update
sudo apt-get install jenkins
```

If you face error "jenkins : Depends: daemon but it is not installable", execute following command after `sudo apt-get update` -

```
sudo add-apt-repository universe
```

This package installation will:

- Setup Jenkins as a daemon launched on start. See `/etc/init.d/jenkins` for more details.
- Create a 'jenkins' user to run this service.
- Direct console log output to the file `/var/log/jenkins/jenkins.log`. Check this file if you are troubleshooting Jenkins.
- Populate `/etc/default/jenkins` with configuration parameters for the launch, e.g `JENKINS_HOME`
- Set Jenkins to listen on port 8080. Access this port with your browser to start configuration.

If your `/etc/init.d/jenkins` file fails to start Jenkins, edit the `/etc/default/jenkins` to replace the line `----HTTP_PORT=8080----` with `----HTTP_PORT=8081----` Here, "8081" was chosen but you can put another port available.

Рисунок 4.2 – Офіційний витяг з документації Jenkins щодо установки інструменту на ОС сімейства Debian

Після того як всі описані шаги були виконані у правильному порядку, як це рекомендую офіційний документ, можна пробувати перевірити чи все працює правильно. Для цього потрібно відкрити будь-який браузер та перейти за посиланням http://localhost:8080 (або інший порт, який ви вказували у конфігурації). Відкривши посилання, я побачив привітальну сторінку (рис. 4.3) і тим самим переконався, що Jenkins встановлений и готовий до використання.

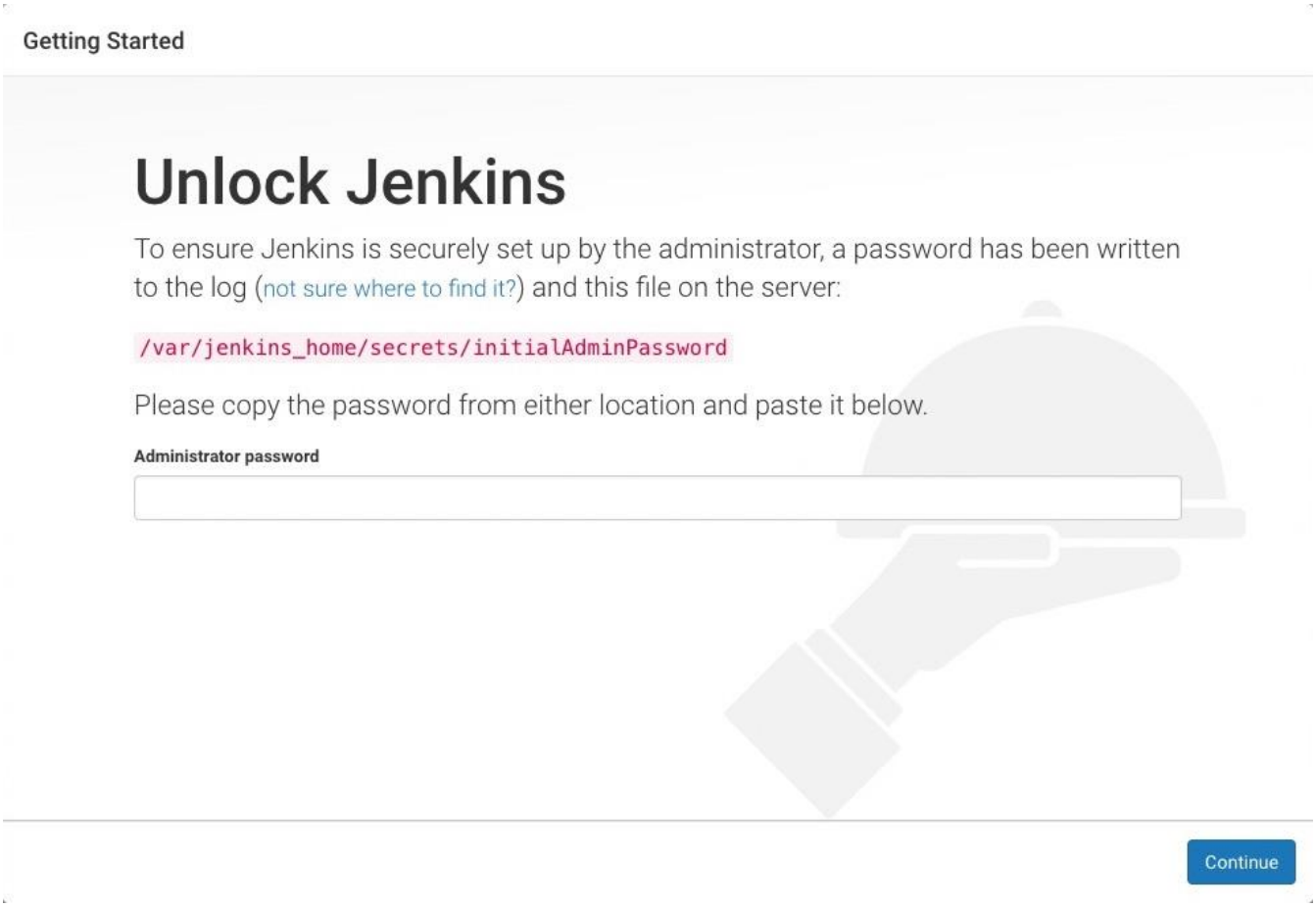


Рисунок 4.3 – Стартова сторінка Jenkins

Для того щоб продовжити працювати та перейти до налаштування, потрібно на сервері, на якому встановлено Jenkins знайти пароль адміністратора. Він знаходиться у /var/Jenkins-home/secrets/initialAdminPassword або ж його можна побачити у журналі під часу його першого запуску, як це показано на рис. 4.4 нижче.

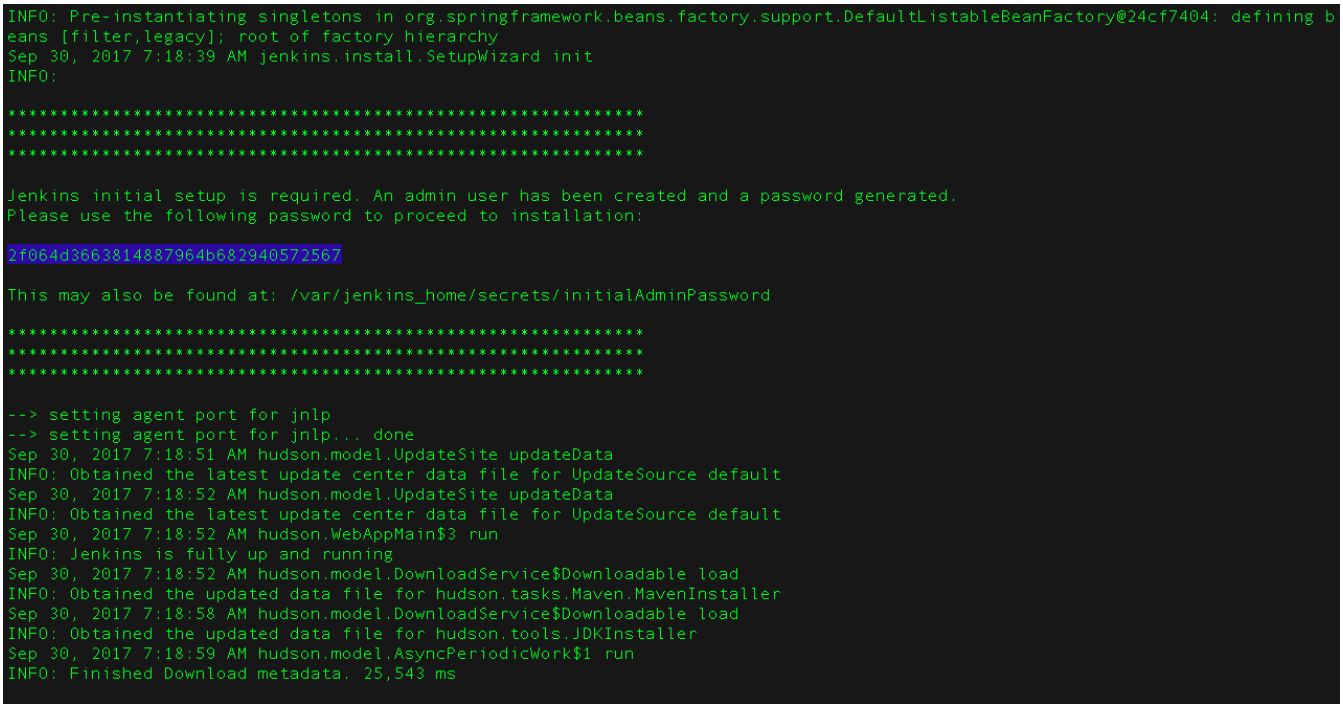


Рисунок 4.4 – Пароль адміністратора Jenkins

Далі Jenkins запропонує за бажанням встановити найпопулярніші плагіни, але нас це зараз не цікавить, тому просто пропустимо цей крок та перейдем до наступного пункту.

Наступним кроком буде спочатку побудова архітектури безперервної інтеграції коду та тестування і вже після цього перейдемо до налаштування безперервної доставки та розгортання.

4.4 Побудова архітектури CI та CT та налаштування

Після того як ми визначились з інструментом реалізації CI/CD/CT та підготували його для подальшого налагодження, я пропоную ознайомитись зі структурною схемою (рис. 4.5) CI та CT процесу для вирішення поставленої задачі.

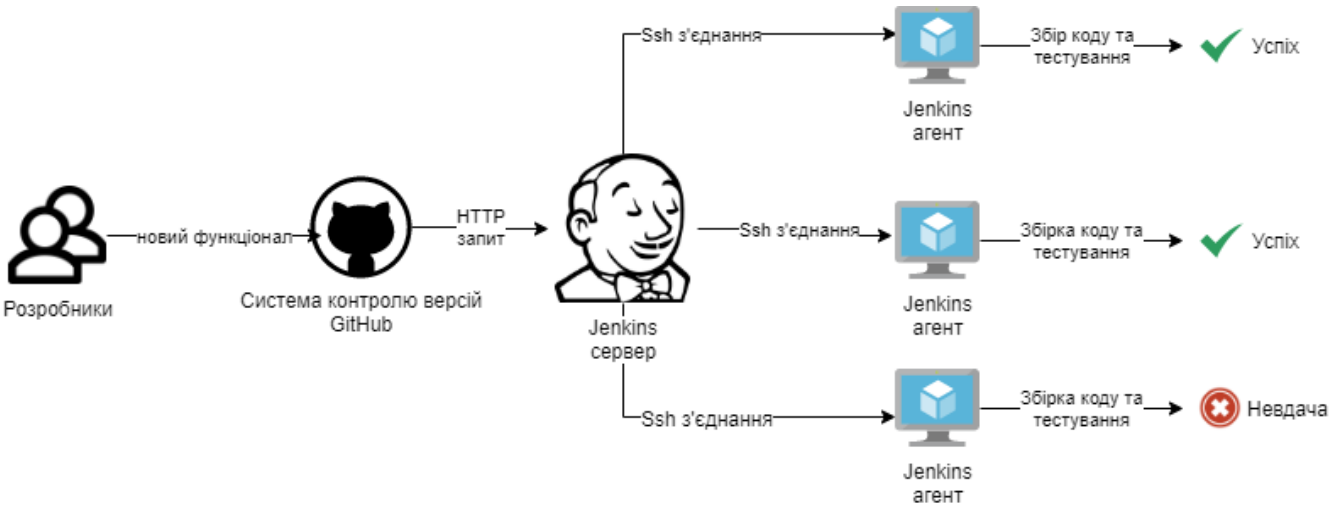


Рисунок 4.5 – Структурна схема CI та CT процесу з використанням Jenkins, git та UNIX агентів

Також, хочу наголосити, що у даній роботі не було поставлено мети розглядати системи контролю версій або інші інструменти, які є допоміжними у розробці ПЗ, тому інформація дуже загальна та стисла.

Перш за все, слід зауважити, що у рамках проекту я буду використовувати найпоширенішу стратегію розгалуження змін коду, яка передбачає створювання окремої гілки у системі контролю версій для кожного окремого функціоналу. Після створення окремим розробником своєї гілки у Git, він повинен створити merge request (з англ. запит на злиття) до основної вітки, яка буде називатись за замовчуванням master. Внаслідок цього буде створюватись HTTP(s) запит до нашого серверу з встановленим Jenkins, де за допомогою WebHook плагіну, ми налаштуємо Jenkins робити повну збірку та тестування коду, якщо такий запит прийде до нього.

Збірка коду буде відбуватись за допомогою спеціального інструменту автоматичної збірки проектів Maven.

На етапі тестування будуть запускатись автоматизовані розробниками та інженерами з автоматизації тести, які в свою чергу також будуть впливати на успішність всього «пайплайну».

Фінальним етапом імплементації та налагоджування CI та CT буде відправка оповіщень розробнику щодо працездатності його коду, та автоматичне підтвердження запиту на злиття у випадку успішного проходження «пайплайну».

4.4.1 Налаштування GitHub Plugin

Перш за все треба встановити GitHub Plugin для Jenkins, який надає можливість дуже гнучко та швидко налаштувати зовнішню інтеграцію між Jenkins та GitHub. Для цього потрібно натиснути кнопку «Manage Jenkins» у лівому стовпці Jenkins і далі обрати пункт «Manage Plugins», як це показано на рис. 4.6 та 4.7.

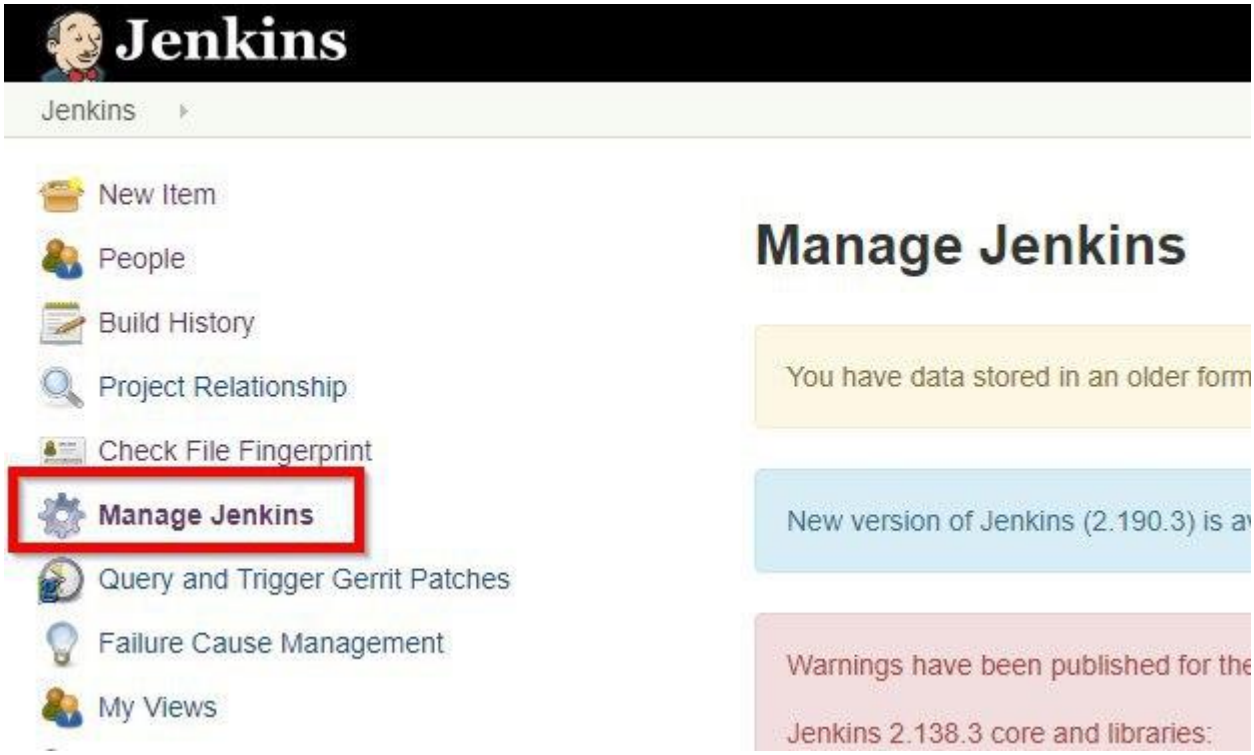


Рисунок 4.6 – Пункт управління Jenkins

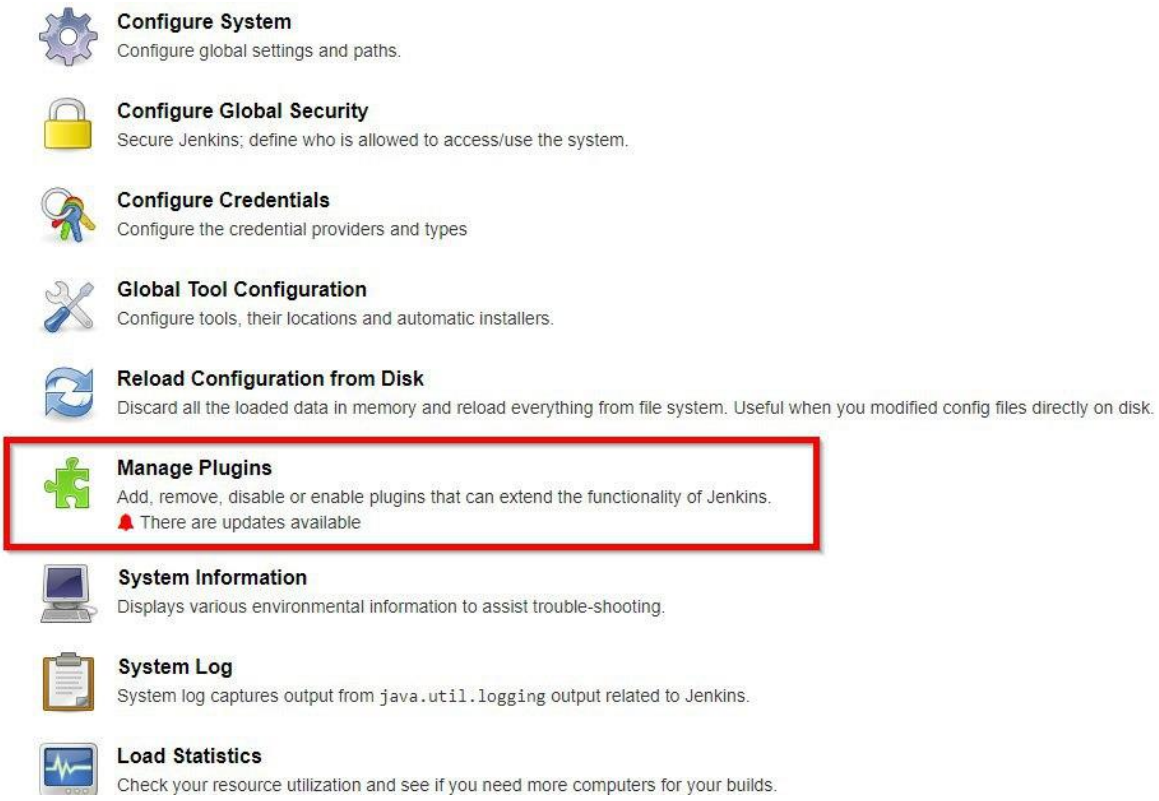


Рисунок 4.7 – Управління плагінами Jenkins

Ми потрапляємо у пункт управління плагінами Jenkins (рис. 4.8), де в нас є можливість переглянути вже встановлені плагіни та встановити доступні у репозиторії плагіни.

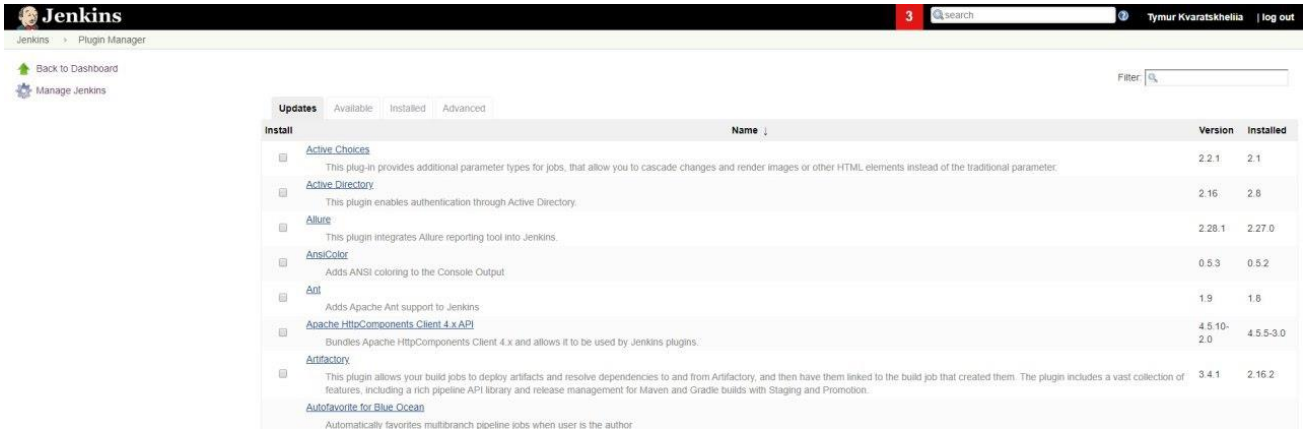


Рисунок 4.8 – Меню управління плагінами

Щоб встановити плагін який нас цікавить достатньо перейти на вкладку «Available» та за допомогою форму пошуку знайти плагін «GitHub Plugin», вибрати галку навпроти плагіна та натиснути кнопку «Install without restart», це дозволить встановити плагін без необхідності перезапуску Jenkins сервера.

Тепер, встановивши необхідний плагін можна перейти до налаштування безпосередньо інтеграції з GitHub, де за ТЗ знаходиться вихідний код розробників. Для цього потрібно створити Jenkins job (рис 4.9) – це і є свого роду pipeline нашого CI процесу, архітектуру якого було описано вище.

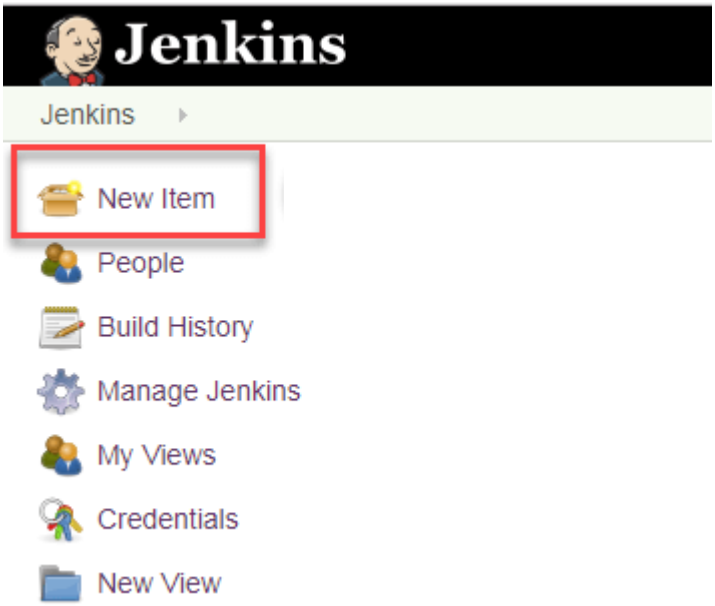


Рисунок 4.9 – Створення нової Jenkin Job для збірки вихідного коду проекту

Створюємо нову Jenkins job в иконуючи кроки зображені на рис. 4.10

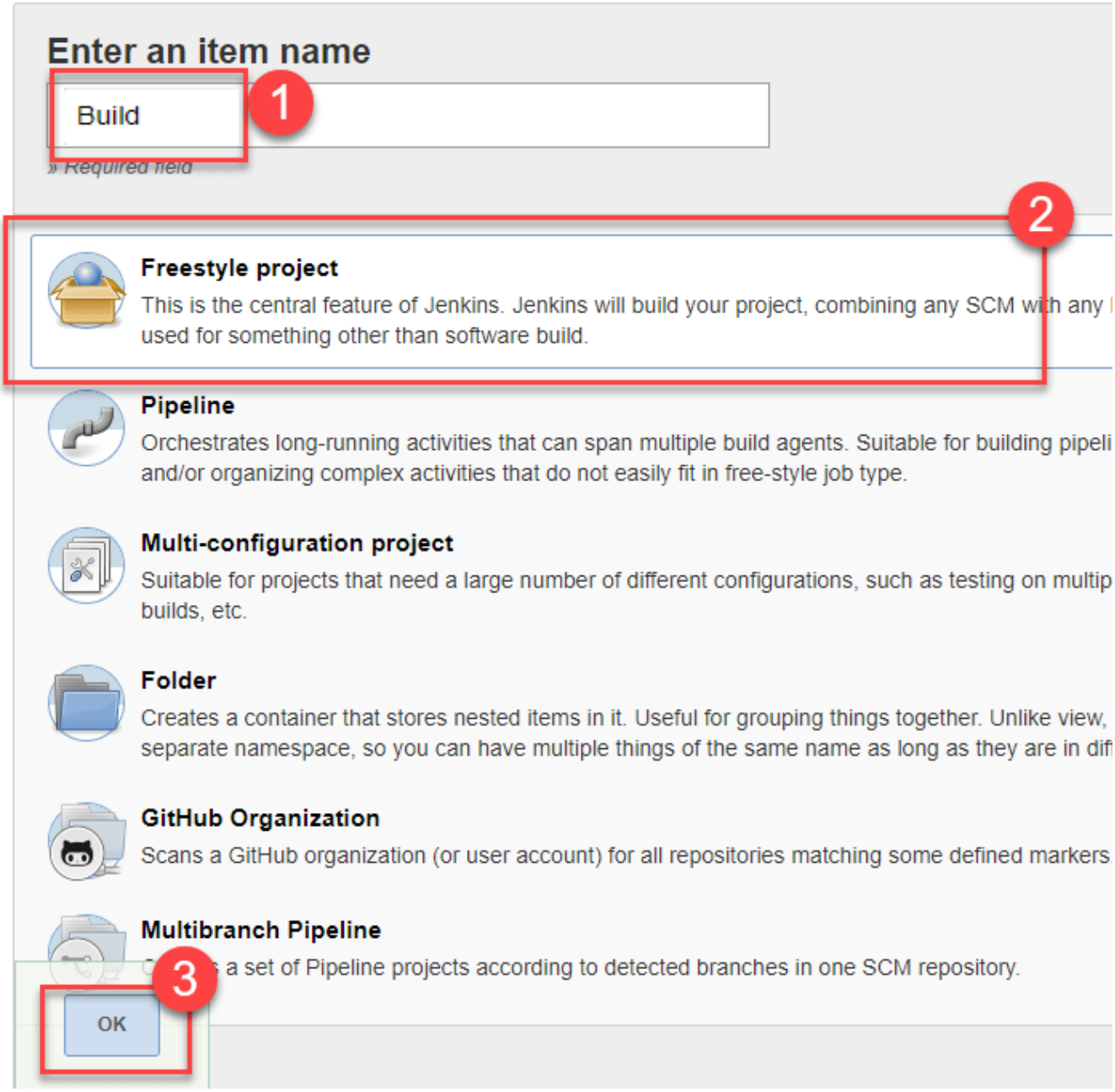


Рисунок 4.10 – створення нової Jenkins Job з назвою Build

Наступним кроком ми потрапляємо на сторінку конфігурації щойно створеної Jenkins Job де можна побачити підрозділ «Source Code Management» (рис 4.11). Обираємо Git, та у полі Repository URL вказуємо посилання на репозитрорій де лежить на код.

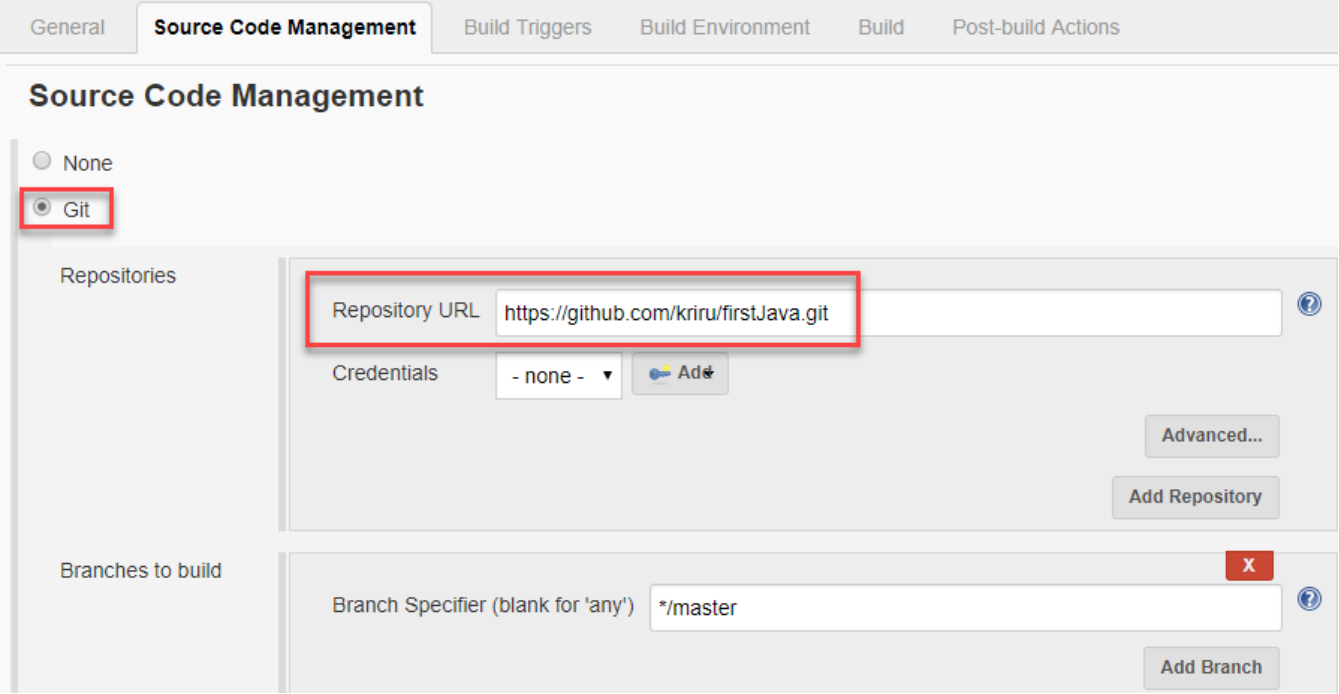


Рисунок 4.11 – Налаштування GitHub плагіну

У полі «Branch Specifier» вказуємо гілку у Git, яка буде використовуватись. Оберемо master.

На цьому налаштування GitHub плагіну можна завершити, тепер Jenkins буде сам «розуміти» де знаходиться вихідний код з яким потрібно працювати у рамках нашого пайплайну.

4.4.2 Налаштування агенту

Для того щоб Jenkins міг зібрати код та виконати автоматичне тестування, потрібно підключити Jenkins «агентів». Саме на цих агентах буде виконуватись збір та тестування. У нашому випадку це будуть звичайні віртуальні машини на базі Linux.

Щоб підключити нового агента до Jenkins потрібно виконати ряд простих дій. Перш за все встановити «Ssh Slaves Plugin», далі переконатись, що ssh сервіс увімкнений на віртуальній машині, яку ми хочемо підключити. Також потрібно створити користувача Linux який має можливість під’єднуватись через ssh та пару ключів шифрування для безпечного з’єднання.

Після вище описаних налаштувань віртуальної машини можна розпочати реєстрацію безпосередньо агента у системі Jenkins.

Потрібно вибрати «Manage Jenkins», як ми це робили у пункті 4.4.1, і далі «Manage Nodes». Ми потрапили на сторінку налаштування агентів, тут потрібно натиснути вкладку «New Node» (рис. 4.12).

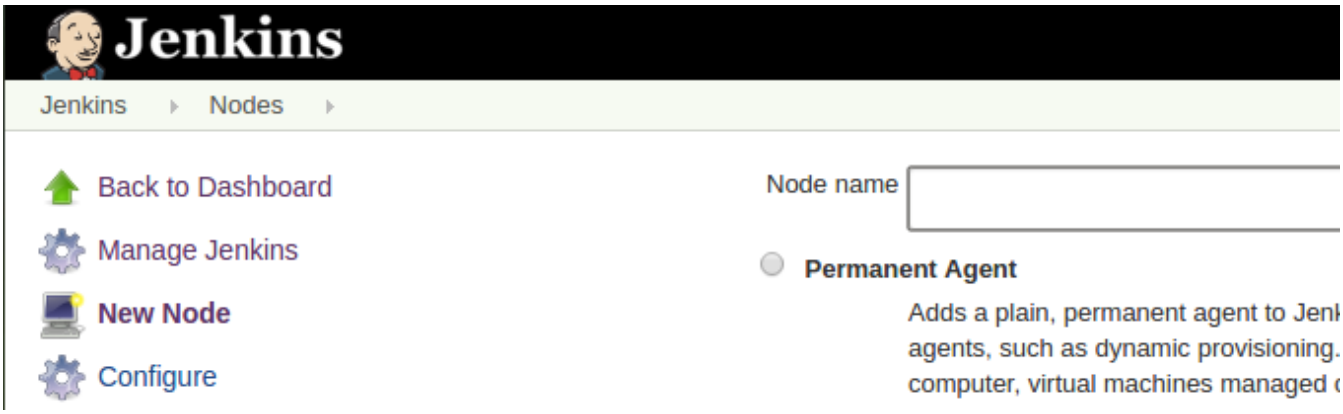


Рисунок 4.12 – Сторінка налаштування Jenkins агентів

Обираємо будь-яке ім’я для нашого агента.

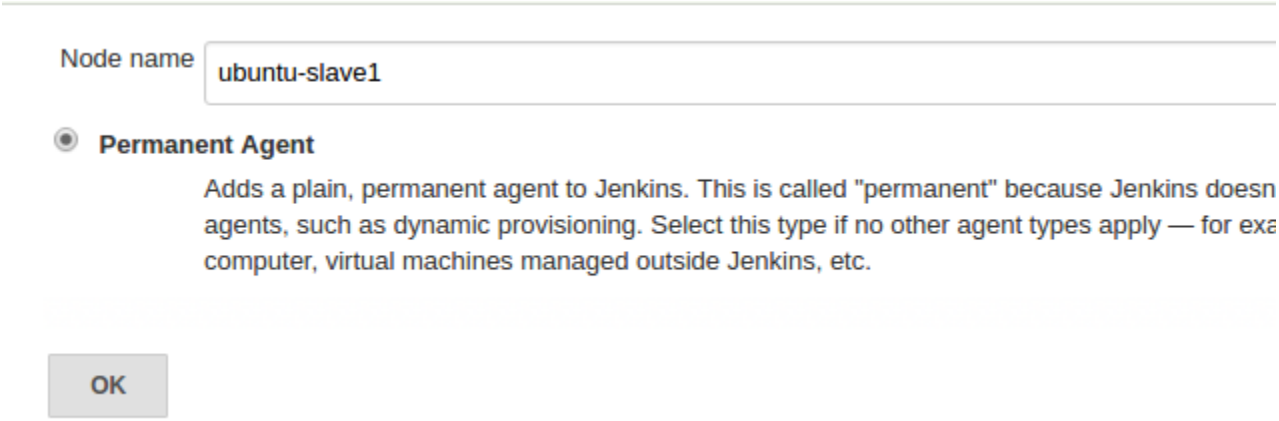


Рисунок 4.13 – Створення агента з ім’ям «ubuntu-slave1»

Натиснувши кнопку ОК, ми потрапляємо на біль детальну сторінку конфігурації створеного агента (рис 4.14)

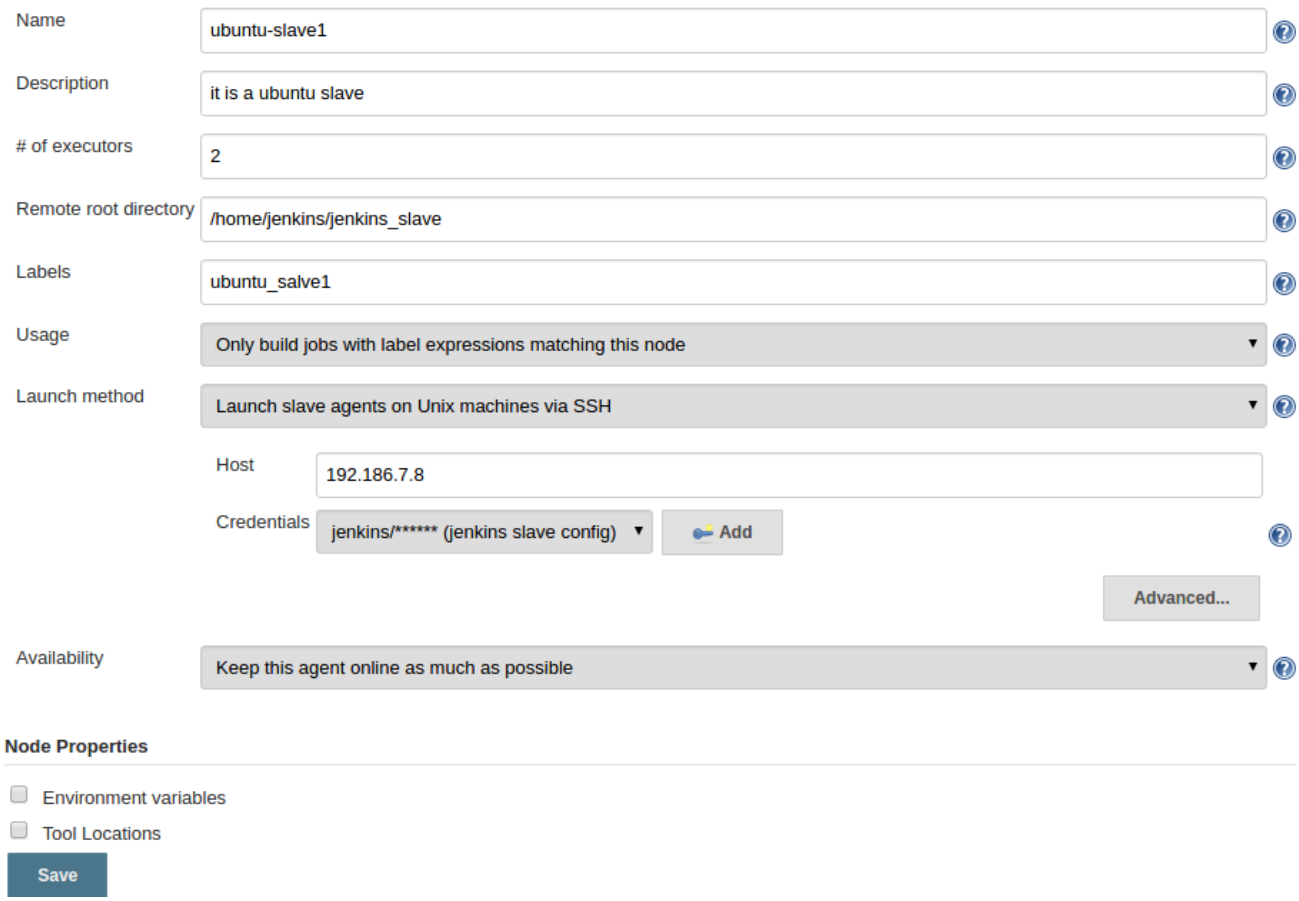


Рисунок 4.14 – Конфігурація агента

Тут можна редагувати вказане ім’я, опис, кількість одночасно можливих Jenkins Job, домашню директорію, у якій буде за замовченням збиратись та тестуватись вихідний код, який буде скачано з GitHub, Host (тобто IP-адреса віртуальної машина яку ми підключаємо) та Credentials (від англ. довірені дані) – логін та пароль, або публічний ключ створеного раніше користувача, який буде використовуватись для ssh з’єднання.

Натиснувши кнопку save, якщо все вказано вірно, ми побачимо, що агент успішно підключився та готовий до роботи.

4.4.3 Налаштування Jenkins Job для збірки проекту та Unit тесту

Налаштувавши агент та плагін для інтеграції з GitHub можна перейти до конфігурування самого пайплайну CI та CT процесів – тобто конфігурування створеної раніше Jenkins Job.

Для того щоб вказати Jenkins використовувати саме той агент який ми щойно підключили потрібно зайти у конфігурацію створеної Jenkins Job з іменем Build, та вибрати галку «Restrict where this project can be run» (рис 4.15). Після цього в полі «Label Expression» потрібно вказати ім’я агента або Label, вказаний при створенні.

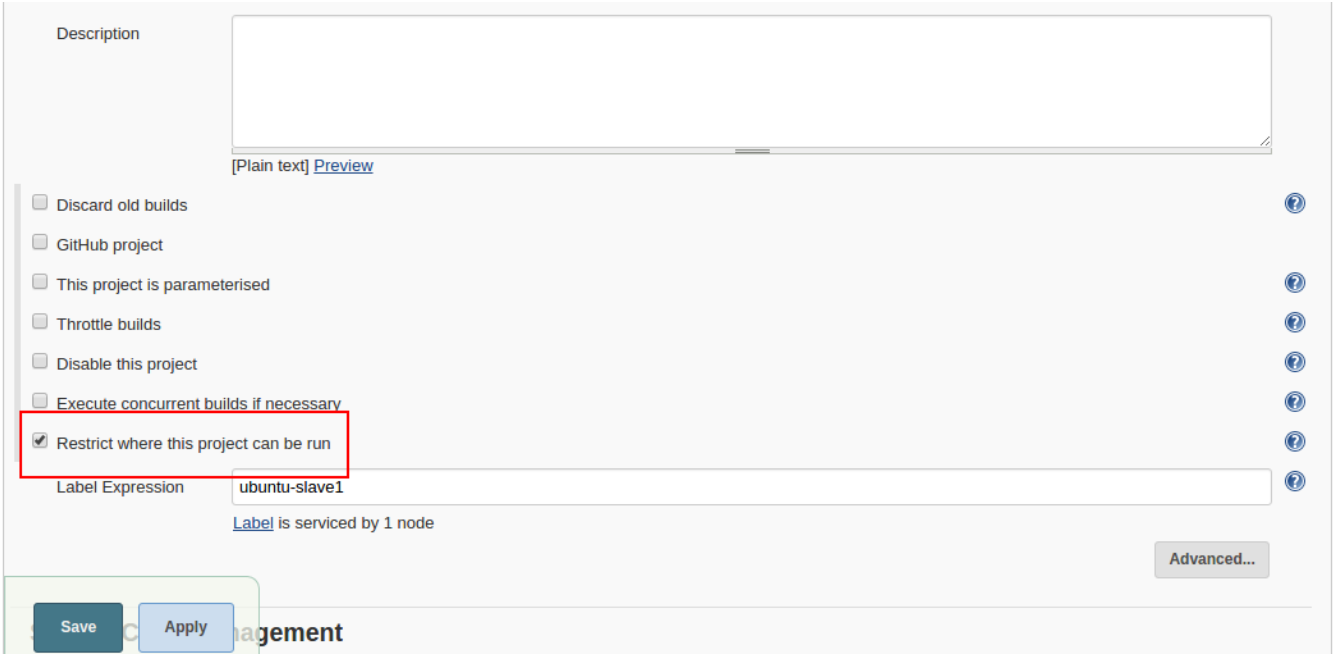


Рисунок 4.15 – Конфігурування Jenkins Job виконуватись на створеному агенті ubuntu-slave1

Задавши свого агента для Jenkins Job переходимо до конфігурування найголовнішої частини CI/CT пайплайну – збірка проекту та тестування. У розділі «Build» обираємо «Add build step» і далі «Invoke top-level Maven targets» (рис 4.16). Внаслідок цих налаштувань Jenkins Job під час виконання буде запускати автоматичний засіб для збірки проекту Maven з вказаними цілями.

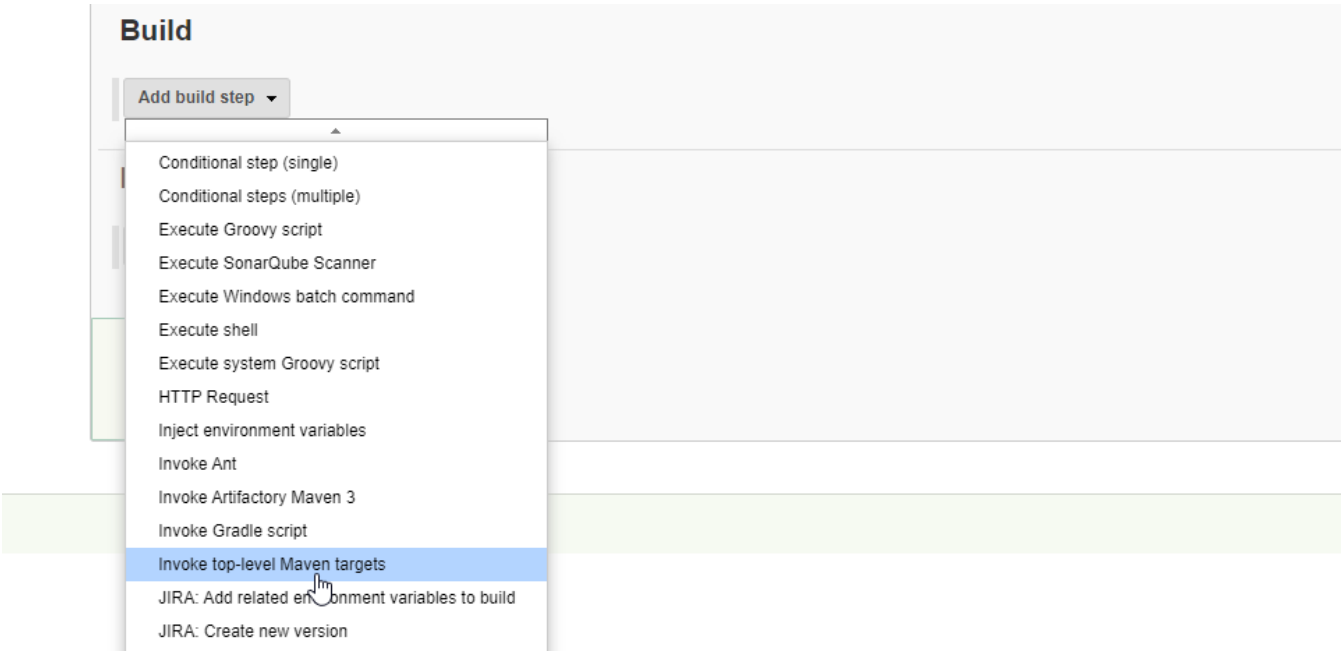


Рисунок 4.16 – Налаштування збірки вихідного коду проекту

Вносимо у поле «Goals» Maven цілі, які вказують на те, як саме буде збиратись вихідний код та чи будуть виконуватись unit (від англ. модульне) тести.

Так як огляд різних видів тестування коду виходить за рамки теми виконуваної роботи, у теоретичній частині немає жодного пояснення. Тому нижче дуже коротко наведена інформація щодо unit тестів, які використовуються у практичній частині як частина безперервного тестування.

Кожна складна програмна система складається з окремих частин - модулів, що виконують ту або іншу функцію в складі системи. Для того, щоб упевнитися в коректній роботі всієї системи, необхідно спочатку протестувати кожен модуль системи окремо. У разі виникнення проблем при тестуванні системи в цілому це дозволяє простіше виявити модулі, що викликали проблему, і усунути відповідні дефекти в них. Таке тестування модулів окремо отримало назву Unit testing (модульне тестування).

Jenkins у своєму арсеналі має дуже корисний плагін який має назву JUnit, і дозволяє дивитись результати unit тестування Java коду та загальний трендовий графік. Встановивши плагін, у секції «Post-build Action», конфігурування Build Job потрібно вибрати відповідний пункт (рис. 4.17).

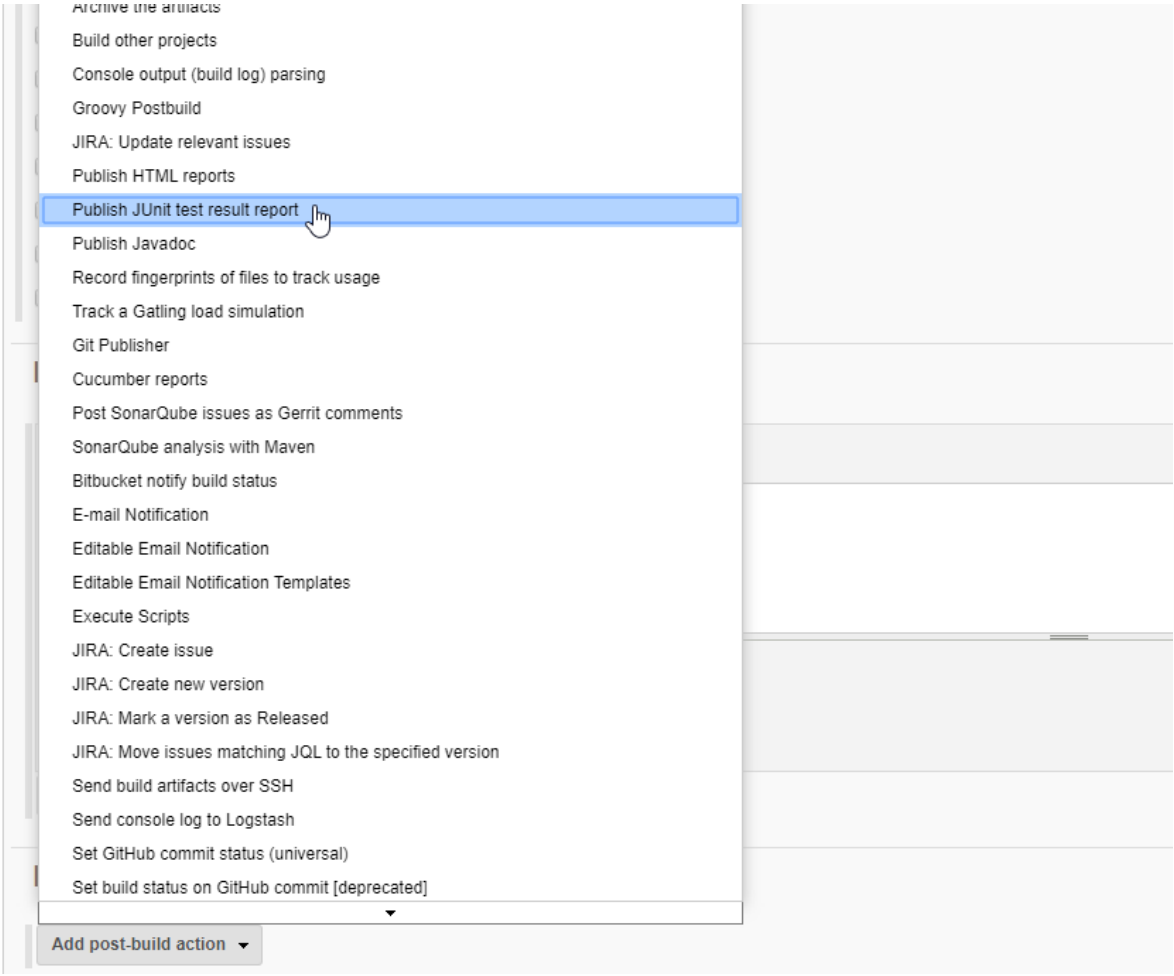


Рисунок 4.17 – Налаштування Build Job відображати результати Unit тестування

У полі Test report XMLs потрібно вказати шлях до XML файлу, який генерується при використанні JUnit тестів у проекті.

У завершенні налаштування даної Build Job потрібно вказати «Build triggers», тобто умови за яких ця Job буде виконуватись автоматично. За замовчуванням є декілька різних умов, але за допомогою величезної кількості плагінів, список цих умов можна легко поширити виходячи з вимог поставленої задачі. Для демонстрації у рамках проекту нам вистачить вбудованих можливостей Jenkins. Оберемо «Build periodically» (рис. 4.18), тобто виконувати цю Jenkins Job автоматично з якоюсь вказаною періодичністю. Нехай це буде 1 раз на годину.

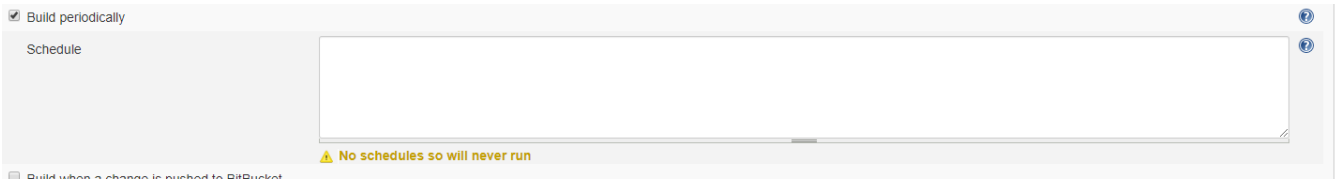


Рисунок 4.18 – Налаштування періодичності збірки та тестування проекту

Jenkins має дуже корисні вбудовані підказки щодо налаштування того чи іншого елементу. Наприклад, для того щоб розібратись як нам правильно налаштувати періодичність виконання, потрібно натиснути на знак запитання, який знаходиться правіше від поля вводу «Schedule» (рис 4.19).

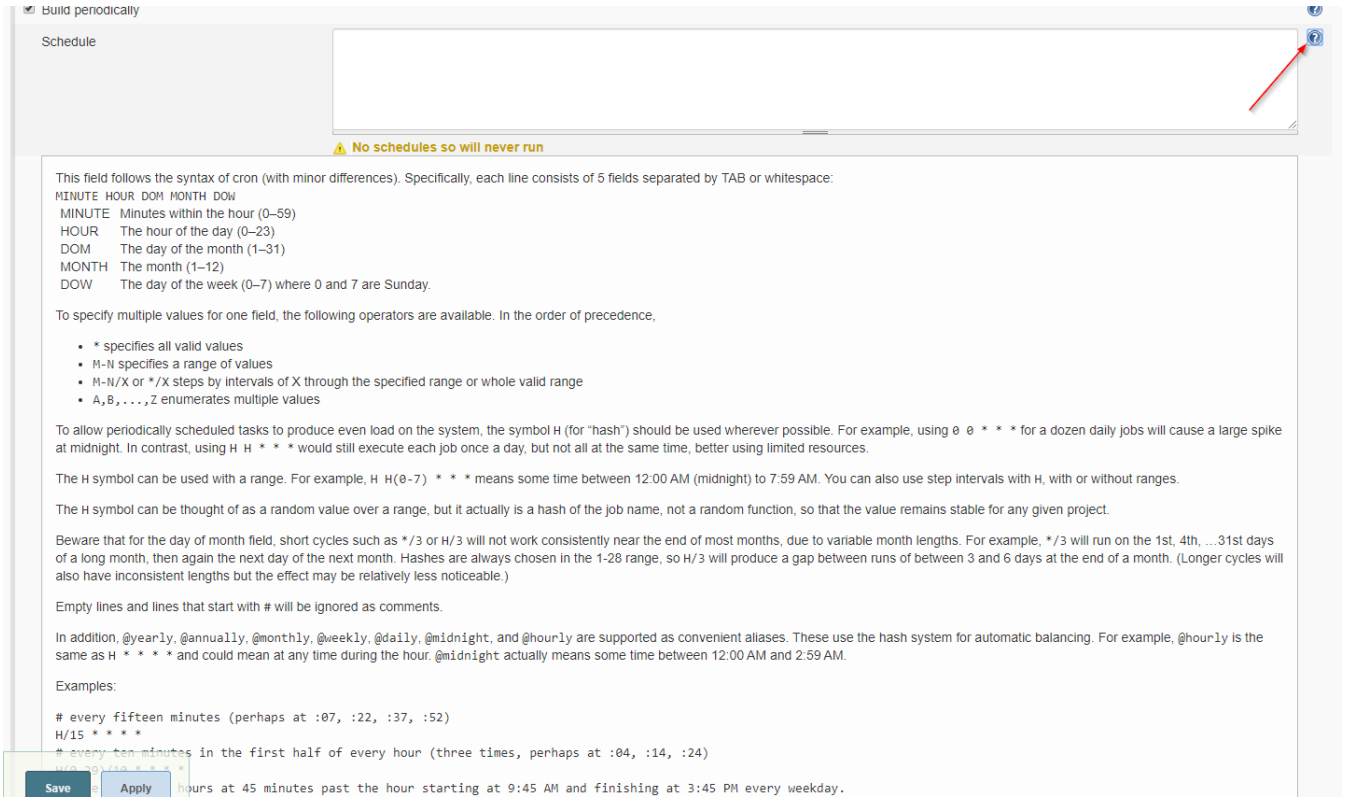


Рисунок 4.19 – Підказки щодо налаштування періодичності виконання Jenkins Job

Детально вивчивши вбудовану документацію щодо використання цієї функції, внесемо у поле «Schedule» потрібні значення для запуску раз на годину, як це показано на рисунку 4.20.

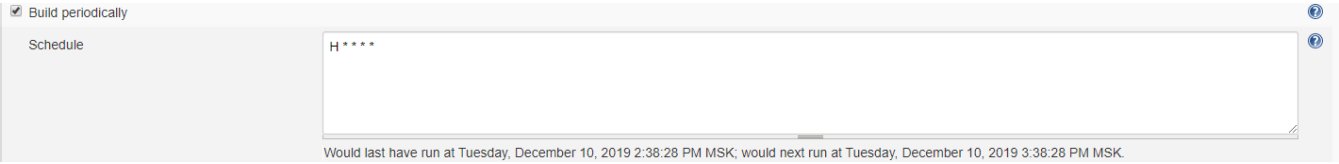


Рисунок 4.20 – Конфігурування запуску Jenkins Job раз на годину

Надивлячись на те, що ми налаштували виконання Jenkins Job кожну годину автоматично, ви все ще маєте можливість запускати цю Job власноруч натиснувши кнопку «Build Now» (рис 4.21)

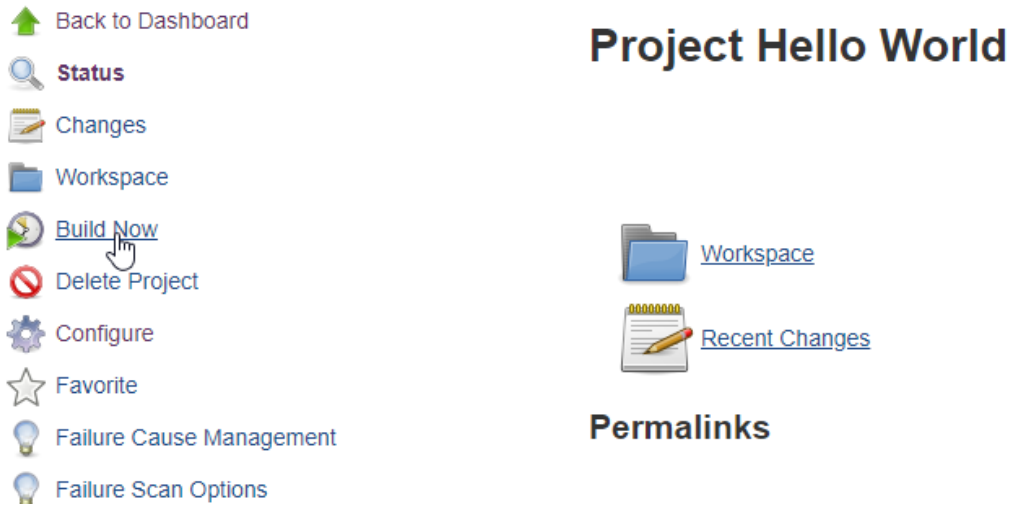


Рисунок 4.21 – Ручне виконання Jenkins Job

Для демонстрації працездатності Jenkins Job, я виконаю її вручну. Після запуску Jenkins Job, у лівому кутку можна бачити історію попередніх збірок та статус виконання поточних. Як можна побачити на рис. 4.22, Jenkins Job зараз у процесі виконання.

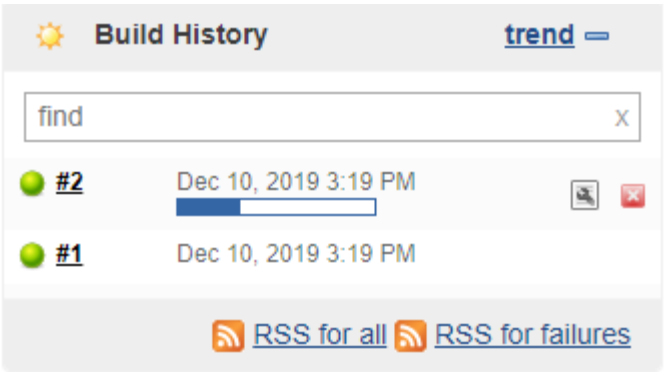


Рисунок 4.22 – Історія запусків Jenkins Job

В результаті вдалої збірки проекту, якщо не було помилок компілятора, всі unit тести пройшли успішно і як наслідок наша Build Job буде мати зелений (або синій) колір.

Для того щоб переглянути результати Unit тестів, необхідно натиснути на Build Job у списку всіх доступних Job. На рис. 4.23 можна побачити графік, який був побудований за допомогою плагіну JUnit Plugin ,який ми встановили раніше.

Рисунок 4.23 – Огляд успішно виконаної Build Job та результати unit тестів

Для більш детального ознайомлення з кожним із виконаних тестів потрібно натиснути кнопку «Latest Test Result».

4.4.5 Створення Jenkins Job для імплементації інтеграційних тестів

Для того щоб розширити наш пайплайн додавши ще один вид тестування - інтеграційне тестування, створимо Jenkins Job, яка буде мати назву Integration Test (рис 4.24).

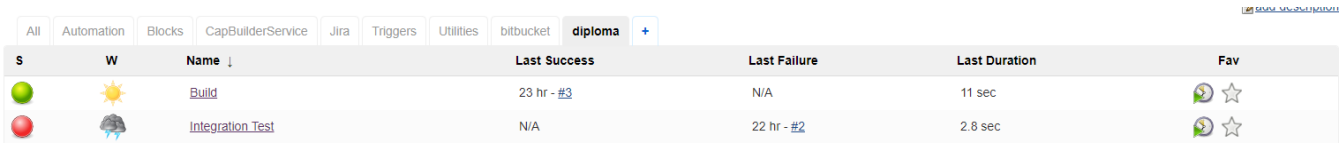


Рисунок 4.24 – Нова Jenkins Job для інтеграційного тестування

Процес написання тестів у рамках даного проекту описуватись не буде. В рамках CI/CD/CT нас цікавить саме налаштування автоматичного пайплайну, який буде збирати, тестувати та розгортувати розроблене програмними інженерами код.

Щодо налаштування Integration Test Job, дуже вдалою практикою буде вказати їй виконуватись кожного разу після вдалої збірки проекту, тобто у разі якщо Build Job була вдалою – запустити тестування. Для того щоб цього домогтись потрібно перейти до налаштування Integration Test та у розділі «Build Triggers», обрати пункт «Build after other projects are built»(рис 4.25), тобно виконувати Jenkins Job одразу після виконання іншої Job.

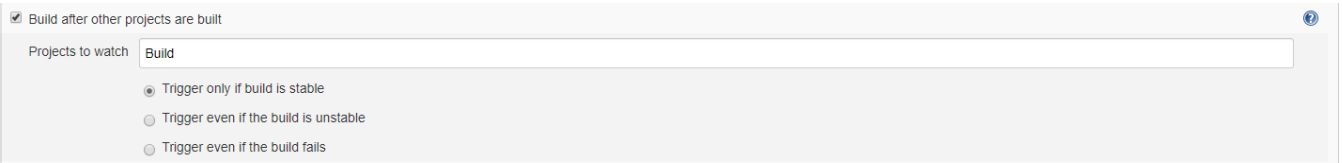


Рисунок 4.25 – Конфігурування Integration Test виконуватись одразу після вдалого виконання Build

4.4.6 Створення та налаштування автоматичних інформувань щодо результатів пайплайну

Фінальним етапом налаштування CI та CT у рамках завдання буде інформування розробника за допомогою E-mail повідомлення щодо результатів проходження пайплайну його вихідного коду.

Для цього у вкладці налаштування кожної з створених раніше Jenkins Job потрібно знайти розділ «Post-build Actions» - тобто дії, які будуть виконані після завершення Jenkins Job, і додати дію, яка має назву «E-mail Notification» (рис 4.26).

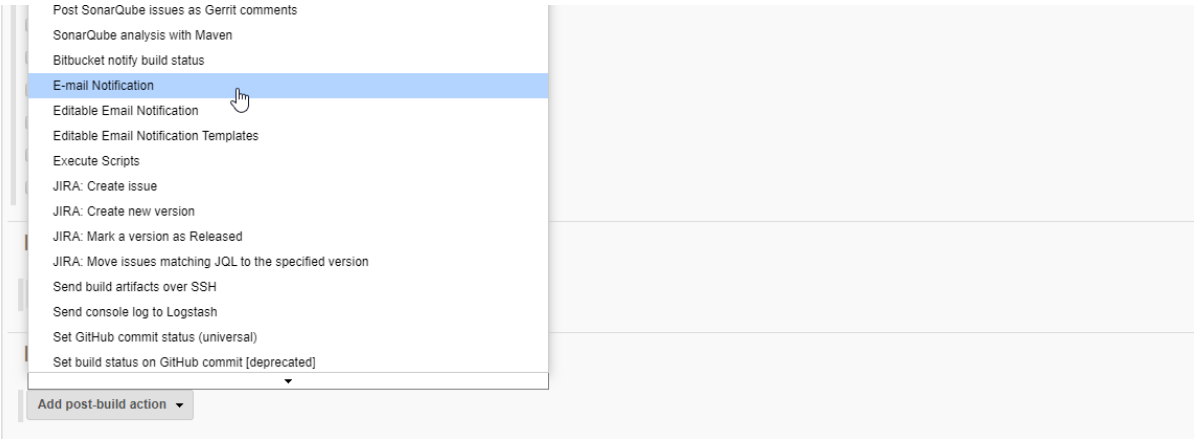


Рисунок 4.26 – Налаштування E-mail інформування щодо завершеної Jenkins Job

Далі у полі «Recipients» потрібно вписати електронну адресу одержувача інформувань.

В результаті дуже нескладних дій, ми налаштували працюючий CI та CT який виконується автоматично раз на один час, та інформує розробника про результати збірки та тестування його частини коду яка знаходиться у гілці master.

4.5 Підготовка архітектури CD та поєднання з налаштованим CI/CT

На практиці у командах розробки ПЗ прийнято називати continuous delivery та continuous deployment однією абревіатурою та дуже часто вважають, що це одне й теж, не виділяючи ніяких чітких розмежувань, але як вже було сформульовано у другому розділі, де було наведена теоретична інформація щодо безперервної інтеграції (CI), тестування (CT), розгортання та доставки (CD), існує дуже чіткі границі між цими поняттями. Безперервна доставка – всі кроки пайплайну повністю автоматизовані, окрім розгортання у працюючу середу, у той час як безперервне розгортання вказує на те, що і останній крок (розгортання у

працюючу середу) є автоматизованим. У рамках практичної частини даної роботи я наведу архітектуру та приклад використання саме continuous deployment підходу, який буде доповнювати попередньо налаштовані continuous integration та continuous testing процеси у межах одного великого CI/CD/CT пайплайну на базі Jenkins.

4.5.1 Архітектура CD процесу з урахуванням вимог технічного завдання

Проаналізувавши ТЗ з точки зору імплементації безперервного розгортання, можна виділити той факт, що замовник вже використовує налаштовані сервери для тестування (staging), та налаштовану робочу середу (production). Саме автоматичне розгортання на staging середу, автоматизоване тестування прикладного інтерфейсу розробленого ПЗ та подальше розгортання на production середу і буде означати повністю налаштований процес CD. На рис. 4.27 схематично зображене безперервне розгортання коду в межах ТЗ.

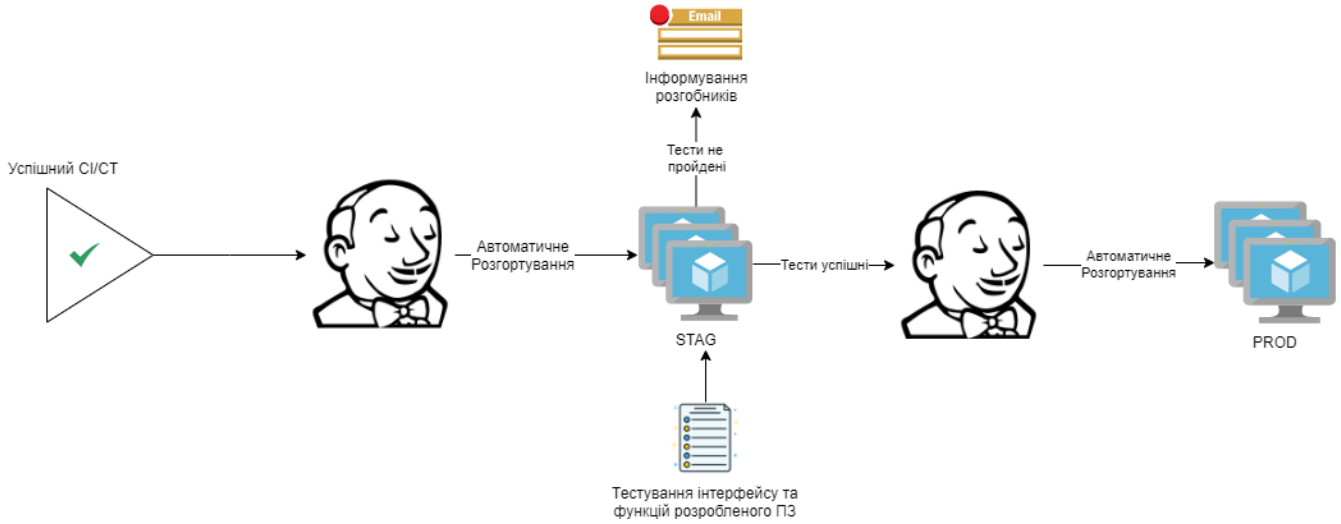


Рисунок 4.27 – Структурна схема архітектури CD процесу

Побудову CD слід почати з створення двох нових Jenkins Job: staging deployment та production deployment, які будуть відповідати за розгортання на тестову та у кінцеву працюючу середу відповідно. Також буде створено ще одну Jenkins Job, яка буде відповідати за тестування працюючого ПЗ у staging середі,

назвемо Test Regression. На рис. 4.28 зображено усі Jenkins Job, які ми маємо у рамках цього проекту.

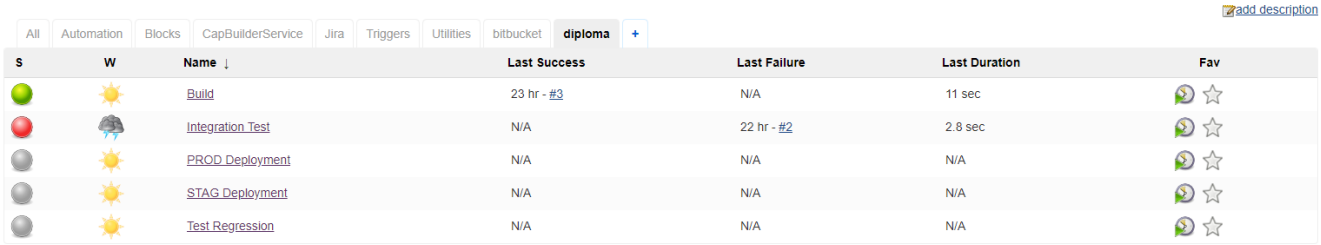


Рисунок 4.28 – Нові Jenkins Job для побудови CD та проміжного тестування

4.5.2 Налаштування атоматичного розгортання та проміжного тестування

Перш за все потрібно під’єднати тестову середу staging до Jenkins як звичайного агента, для того щоб надати Jenkins серверу можливість виконувати дії на цих тестових серверах. Після налаштування агентів, у конфігурації staging deployment Job було вказано параметр «Restrict where this project can be run», саме так як ми робили для Jenkins Job, які налаштовувались у рамках створення CI та CT у підпункті 4.4. Також слід не забувати налаштувати параметр «Build after other projects are built», який буде виконувати цю Jenkins Job тільки у випадку, якщо Integration Test був успішним.

Наступним кроком буде розгортання успішної версії коду на тестовій середі за допомогою того ж самого «Build» кроку, але вже з іншими Maven цілями, які окрім збіру проекту, ще й розгортує код і надає можливість тестувати працюючий додаток.

Конфігурація Test Regression буде складатись із запуску тестів для розгорнутого додатку у тестовій середі. Ця Jenkins Job буде виконуватись одразу після успішного розгортання у тестовому середовищі.

Така ж сама концепція налагодження розгортання ку production середовище.

4.6 Огляд кінцевої схеми створеного CI/CD/CT за допомогою Jenkins

У підсумку, для наглядності повного пайплайну на рис. 4.29 наведено блок-схему алгоритму роботи автоматичного CI/CD/CT.

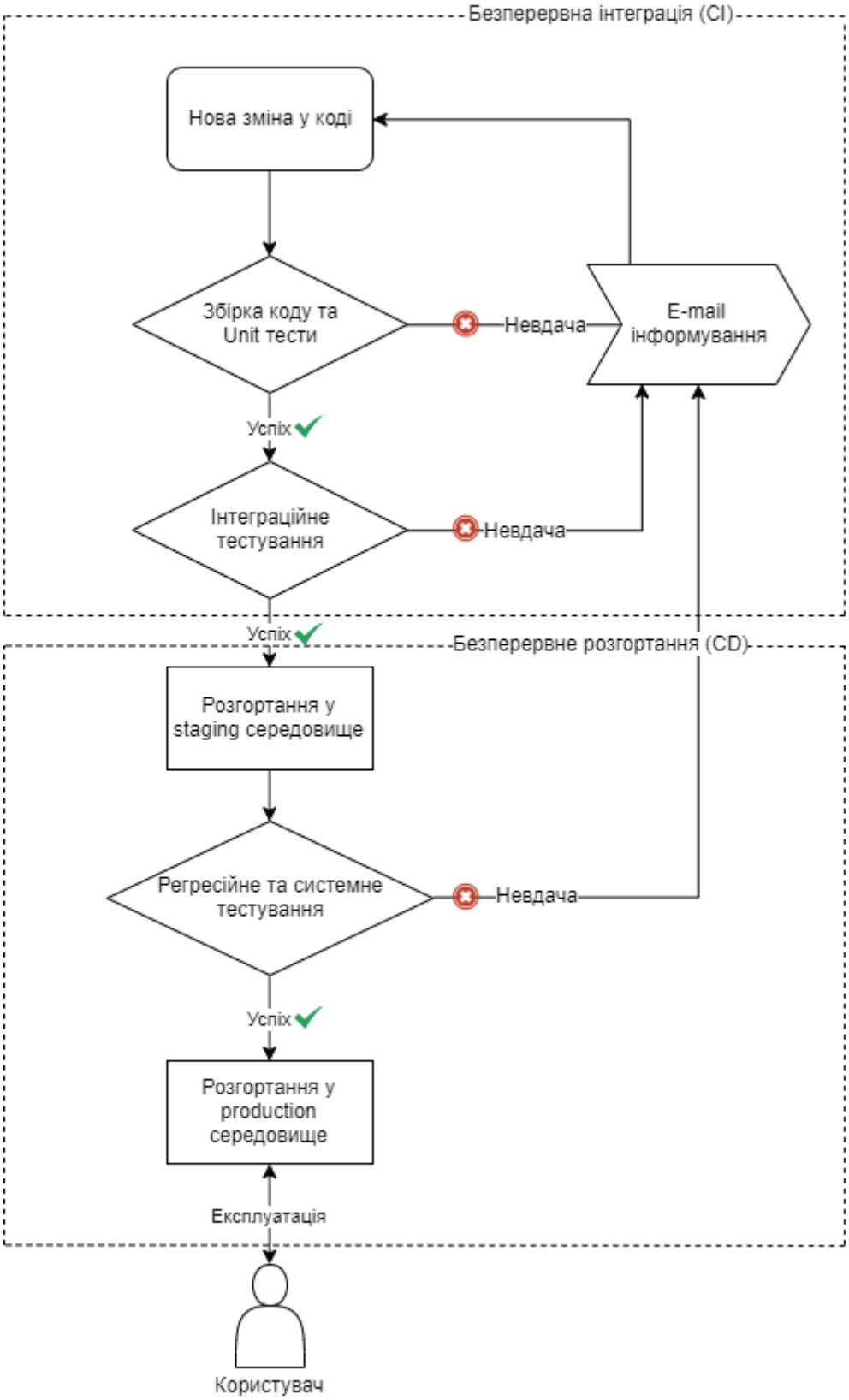


Рисунок 4.29 – Алгоритм роботи налаштованого CI/CD/CT у рамках ТЗ

Так як автоматизація всього процесу CI/CD/CT за допомогою обраного інструмента Jenkins не потребує дуже складних конфігурацій або грошових вкладів, дуже легко побачити виграш використання концепції на базі цього програмного засобу. Як було сказано у розділах 1 та 2, процес розробки ПЗ завдяки автоматизації значно прискорюється та зменшується людська помилка при тестуванні. Спроектована система у четвертому розділі дозволяє замовнику ПЗ мати нову версію додатку кожен час, часто випускати оновлення та змінювати вимоги. Без використання цих підходів, домогтися таких результатів було б неможливо. Процес розробки, виявлення помилок під час написання коду та знаходження несправностей за рахунок автоматизованого збору проекту та тестування кожен наступний час, у декілька десятків разів прискорює процес розробки на всіх етапах життєвого циклу. Переваги такої системи очевидні.

ВИСНОВКИ

В ході виконання магістерської дипломної роботи у першому розділі, для того щоб зрозуміти чому з’явилися поняття continuous integration, testing, delivery та deployment, були розглянуті всі етапи життєвого циклу розробки програмного забезпечення, еволюцію методологій розробки. Після детального ознайомлення з теоретичною інформацією, виявилось що Agile та DevOps набули надзвичайної популярності серед сучасних розробників у зв’язку з тим, що дозволяють адаптувати бізнес під теперішні шалені темпи та вимоги з боку кінцевих користувачів. Саме такі умови посприяли виникненню та розповсюдженню автоматизованого CI, CD та CT у рамках методології DevOps.

Другий розділ було присвячено більш точному визначенню термінів безперервної інтеграції, тестування, розгортання та доставка та наведенню різниці між ними. Далі, у рамках третього розділу було зібрано та проаналізовано значну кількість інструментів реалізації цих автоматичних процесів. Було обрані критерії аналізу: ціна використання, хостинг, підтримка контейнеризації, наявність плагінів, документація та підтримка та складність використання опираючись на оцінку яких були наведені діаграми та графіки, які виявили явних лідерів серед досліджуваних інструментів, та зробили можливість на основі аналізу обрати саме той варіант, який буде кращим у тому або іншому випадку, в залежності від вимог та технічного завдання. Наприклад, Jenkins – ідеальний варіант безкоштовного інструменту з можливістю безмежного розширення можливостей за рахунок використання великої кількості доступних плагінів, а Bamboo – кращий вибір випадку, коли замовник ПЗ використовує інструменти Atlassian стеку.

Використовуючи ці аналітичні дані та проведений порівняльний аналіз, ґрунтуючись на отриманому технічному завданні, вдалось обрати найбільш гнучкий та вдалий для виконання завдання інструмент. Це в свою чергу дозволило спроектувати та побудувати повністю функціонуючу та готову для використання автоматизовану систему безперервної інтеграції, тестування, розгортання та доставки коду. Саме цьому був присвячений останній розділ дипломної роботи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Continuous Integration: CircleCI vs Travis CI vs Jenkins vs Alternatives [Електронний ресурс] // Django Stars. – 2019. – Режим доступу до ресурсу: https://djangostars.com/blog/continuous-integration-circleci-vs-travisci-vs-jenkins/

2. Travis CI Tutorial [Електронний ресурс] – 2019. – Режим доступу до ресурсу: https://docs.travis-ci.com/

3. GoCD User Documentation [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://docs.gocd.org/current/

4. Installing Jenkins [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://jenkins.io/doc/book/installing/

5. G2 Grid for Continuous Integration [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.g2.com/categories/continuous-integration

6. ВВЕДЕНИЕ В НЕПРЕРЫВНУЮ ИНТЕГРАЦИЮ, ДОСТАВКУ И РАЗВЕРТЫВАНИЕ [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://www.8host.com/blog/vvedenie-v-nepreryvnuyu-integraciyu-dostavku-i-razvertyvanie/

7. Сайт хмарного сервісу Amazon Web Services [Електронний ресурс] – Режм доступу до ресурсу https://aws.amazon.com/ru/devops/continuous-delivery/ – 12.10.2019 р. Загл з екрану.

8. Что такое непрерывная доставка? [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://aws.amazon.com/ru/devops/continuous-delivery/

9. What is DevOps? The Ultimate Guide to DevOps [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://resources.collab.net/devops-101/what-is-devops

10. Подходы к разработке ПО: как правильно выбрать методологию разработки программного обеспечения [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://issoft.by/blog/podkhody-k-razrabotke-po-kak-pravilno/

11. Linux 64-bit installation instructions for Java [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://java.com/en/download/help/linux_x64_install.xml

12. Kvaratskheliia T. E. Дослідження загроз та методів забезпечення безпеки хмарних обчислень [Електронний ресурс] / T. E. Kvaratskheliia, M. I. Lysenko // Topical issues of the development of modern science. Abstracts of the 3rd International

scientific and practical conference.Publishing House “ACCENT”. Sofia, Bulgaria.. – 2019. – Режим доступу до ресурсу: http://sci-conf.com.ua.