БАКАЛАВРСЬКА РОБОТА

БР. ІП - 98.00.00.000 ПЗ

Група ІП-22-4

Зозулин Артем

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Зозулин Артем Русланович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Техніки створення внутрішніх платформ розробника (Internal Developer Platforms)

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____Зозулин А.Р._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____Корнута Володимир Андрійович, к.т.н., доцент_
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Зозулин Артем Русланович

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Техніки створення внутрішніх платформ розробника (Internal Developer Platforms)"

керівник проекту (роботи) Корнута Володимир Андрійович, к.т.н., доцент

затверджені наказом вищого навчального закладу від “ 21_ ” _квітня 2026 р. № ___179/7____

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

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

1. Аналіз вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

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

1) - Ключові принципи DevOps. (рис. 1.1, ст. 17)
2 Загальний огляд процесу GitOps (рис. 1.2, ст. 22)
4. - Багаторівнева архітектура IDP (рис. 2.1, ст. 39)
4 Загальний огляд IDP (рис. 3.1, ст. 55)
5 Програмні засоби для проведення спринтів (рис. 3.6, ст. 63)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	17.01.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.02.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.03.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.04.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	02.05.2026	виконано

Студент _____Зозулин А.Р.
(підпис)

Керівник роботи _____Корнута В.А.
(підпис)

АНОТАЦІЯ

Бакалаврська робота робота містить 85 сторінок, 12 рисунків, 1 таблиця, список використаних джерел із 40 найменувань.

Метою роботи є дослідження технік створення внутрішніх платформ розробника та аналіз підходів до автоматизації процесів розробки, розгортання і супроводу програмного забезпечення в сучасних IT-системах.

Об'єкт дослідження: внутрішні платформи розробника.

Предмет дослідження: методи, архітектурні підходи та інструменти створення внутрішніх платформ розробника, зокрема практики DevOps, Platform Engineering, CI/CD та автоматизації інфраструктури.

Результати дослідження: проведено аналіз концепції Internal Developer Platforms, досліджено архітектурні підходи до побудови, розглянуто сучасні інструменти автоматизації процесів розробки та впровадження ПЗ.

У першому розділі розглянуто теоретичні основи внутрішніх платформ розробника, концепції DevOps та Platform Engineering, а також IDP.

У другому розділі досліджено методи та інструменти побудови внутрішніх платформ розробника, зокрема референтні архітектури, архітектурні компоненти та основні функціональні можливості IDP.

У третьому розділі розглянуто реалізацію внутрішньої платформи розробника, процеси впровадження та інтеграції платформи у цикл розробки програмного забезпечення, а також проведено аналіз ефективності.

Висновок: використання внутрішніх платформ розробника дозволяє підвищити продуктивність команд розробки, спростити процеси розгортання та супроводу програмного забезпечення, забезпечити стандартизацію інфраструктури та автоматизацію DevOps-процесів.

КЛЮЧОВІ СЛОВА: INTERNAL DEVELOPER PLATFORM, IDP, DEVOPS, PLATFORM ENGINEERING, CI/CD, KUBERNETES, AUTOMATION, CLOUD INFRASTRUCTURE, SOFTWARE DEVELOPMENT, MICROSERVICES

ANNOTATION

T The thesis consists of 85 pages, 12 figures, 1 tables, and a reference list containing 40 sources.

The aim of the work is to study techniques for creating Internal Developer Platforms (IDP) and to analyze approaches to automating software development, deployment, and maintenance processes in modern IT systems.

Research object: Internal Developer Platforms (IDP).

Research subject: methods, architectural approaches, and tools for building Internal Developer Platforms, including DevOps practices, Platform Engineering, CI/CD, and infrastructure automation technologies.

Research results: the concept of Internal Developer Platforms was analyzed, architectural approaches to IDP implementation were studied, modern tools for automating software development and deployment processes were reviewed, and the effectiveness of using internal platforms in development environments was evaluated.

The first section describes the theoretical foundations of Internal Developer Platforms, DevOps and Platform Engineering concepts, as well as approaches to IDP design.

The second section analyzes methods and tools for building Internal Developer Platforms, including reference architectures, architectural components, and the main functional capabilities of IDP.

The third section covers the implementation of an Internal Developer Platform, platform deployment and integration into the software development lifecycle, and the evaluation of IDP effectiveness.

Conclusion: the use of Internal Developer Platforms improves development team productivity, simplifies software deployment and maintenance processes, ensures infrastructure standardization, and automates DevOps processes.

KEY WORDS: INTERNAL DEVELOPER PLATFORM, IDP, DEVOPS, PLATFORM ENGINEERING, CI/CD, KUBERNETES, AUTOMATION, CLOUD INFRASTRUCTURE, SOFTWARE DEVELOPMENT, MICROSERVICES.

ЗМІСТ

					БР.ІП –98.00.00.000 ІПЗ		
Змн.	*Арк.*	*№ докум.*	*Підпис*	*Дата*			
Розроб.		Зозулин А.Р.			Техніки створення внутрішніх платформ розробника (Internal Developer Platforms	*Літ.*	*Арк.* 7	*Акрушів*
Перевір.		Корнута В.А.						
Реценз.		Романишин Т.Л.			**Пояснювальна записка**	**ІФНТУНГ ІП-22-4**		
Н. Контр.		Піх М.М.						
Затверд.		Бандура В. В.						

ВСТУП

У сучасних умовах стрімкого розвитку інформаційних технологій та зростання складності програмних систем особливої актуальності набувають підходи до автоматизації процесів розробки, розгортання та супроводу програмного забезпечення. Сучасні компанії активно переходять до мікросервісної архітектури, хмарних обчислень і практик DevOps, що вимагає створення ефективних інструментів для координації процесів розробки та управління інфраструктурою. У таких умовах важливу роль відіграють внутрішні платформи розробника (Internal Developer Platforms, IDP), які забезпечують стандартизацію процесів, автоматизацію рутинних операцій та підвищення продуктивності команд розробки.

Зростання кількості сервісів, складності інфраструктури та потреби у швидкому випуску програмного забезпечення створює нові виклики для IT-компаній. У зв’язку з цим виникає потреба у створенні централізованих внутрішніх платформ, які дозволяють спростити взаємодію між командами розробки та операційної підтримки, забезпечити автоматизацію життєвого циклу програмного забезпечення та мінімізувати кількість ручних операцій.

Актуальність теми зумовлена необхідністю підвищення ефективності процесів розробки програмного забезпечення, скорочення часу розгортання нових сервісів та забезпечення стабільності й масштабованості IT-інфраструктури. Internal Developer Platforms дозволяють реалізувати концепції Platform Engineering та DevOps, забезпечуючи єдине середовище для автоматизації розробки, тестування, розгортання та моніторингу програмних систем.

Метою роботи є дослідження технік створення внутрішніх платформ розробника (Internal Developer Platforms), аналіз сучасних підходів до Platform Engineering та розробка архітектурних рішень для автоматизації процесів розробки, розгортання й супроводу програмного забезпечення.

Завданнями дослідження є аналіз предметної області та сучасних підходів до створення Internal Developer Platforms, дослідження концепцій DevOps і Platform Engineering, визначення функціональних та нефункціональних вимог до

внутрішніх платформ розробника, аналіз архітектурних компонентів та інструментів автоматизації.

Об'єктом дослідження є процеси побудови та використання внутрішніх платформ розробника у середовищі сучасної розробки програмного забезпечення.

Предметом дослідження є методи, архітектурні підходи, технології та інструменти створення Internal Developer Platforms, зокрема практики DevOps, Platform Engineering, CI/CD, контейнеризації, оркестрації та автоматизації інфраструктури.

Методи дослідження включають аналіз наукових джерел для вивчення теоретичних основ Platform Engineering та DevOps, порівняльний аналіз сучасних технологій автоматизації, моделювання архітектури внутрішньої платформи розробника, експериментальний метод для реалізації та тестування компонентів платформи, а також аналіз результатів використання IDP у процесі розробки програмного забезпечення.

У роботі передбачається дослідження сучасних технік створення внутрішніх платформ розробника, аналіз інструментів автоматизації та оркестрації, а також реалізація архітектурного підходу до побудови IDP. Особлива увага приділятиметься автоматизації процесів CI/CD, стандартизації інфраструктури, забезпеченню масштабованості та підвищенню продуктивності команд розробки. Очікується, що використання Internal Developer Platforms дозволить скоротити час розгортання сервісів, підвищити стабільність програмних систем та оптимізувати процеси управління інфраструктурою.

Бакалаврська робота містить 85 сторінок, 12 рисунків, 1 таблиця, 3 розділи, список використаних джерел із 40 найменувань.

РОЗДІЛ 1 . ТЕОРЕТИЧНІ ОСНОВИ ВНУТРІШНІХ ПЛАТФОРМ РОЗРОБНИКА

1.1 Поняття та призначення Internal Developer Platforms (IDP)

У сучасному швидкозмінному технологічному середовищі організації стикаються зі значними труднощами у впровадженні робочих процесів розробки програмного забезпечення. Зростаючий попит на швидші терміни виконання, а також потреба в надійній роботі, створює посилений тиск на компанії щодо впровадження інновацій у свої робочі процеси розробки.

DevOps виник у відповідь на ці виклики, інтегруючи розробку програмного забезпечення та операції для підвищення гнучкості та надійності. Завдяки впровадженню культури співпраці між раніше ізольованими командами, практики DevOps допомагають скоротити час виведення на ринок нових функцій та забезпечити вищу якість обслуговування [1]. DevOps надає багато цінності, але деякі організації все ще мають проблеми з впровадженням практик DevOps. Поширеними проблемами під час впровадження практик DevOps є відсутність чіткого визначення та кроків, які потрібно виконати, а також високий попит на кваліфікований персонал. Ще однією поширеною проблемою є те, що автоматизації, необхідної для операцій та моніторингу, зазвичай недостатньо.

Внутрішня платформа розробника (IDP) пропонує інноваційний підхід до подальшого оптимізації життєвого циклу розробки програмного забезпечення. IDP забезпечують структуровану систему, яка централізує доступ до різних інструментів, зменшує когнітивне навантаження на розробників та автоматизує складні операційні завдання, підвищуючи ефективність та забезпечуючи автономію розробників [2].

У цій роботі досліджується, що таке IDP та яку користь воно може принести організаціям, які його впроваджують. Щоб зрозуміти, звідки виникла потреба в такій ідеї, необхідний короткий вступ до DevOps та його методологій. Робота також охоплює деякі відомі проблеми, з якими стикаються організації, коли

впроваджують DevOps. Щоб краще зрозуміти ідею, було проведено тематичне дослідження організації, яка має проблеми з існуючими робочими процесами розробки. У тематичному дослідженні розглядаються потенційні переваги, які впровадження IDP може надати організації.

Сучасні прояви передового досвіду розробки та розгортання програмного забезпечення, що відбулися протягом останніх кількох років, спонукали компанії переглянути свої підходи до управління інфраструктурою, автоматизації робочих процесів та продуктивності розробників [3]. Традиційні DevOps-проекти, хоча й є революційними, як правило, наражають розробників на незрозумілий набір пристроїв, середовищ та операційних проблем. Зі зростанням швидкості розробки та зростанням розподілених та динамічних систем, особливо в хмарних та мікросервісних системах, команди стикаються з дедалі більшими труднощами у забезпеченні узгодженості, надійності та швидкої доставки протягом усього життєвого циклу розробки програмного забезпечення.

Однією зі стратегічних реакцій на ці проблеми є платформна інженерія. Вона пропонує новий спосіб мислення про внутрішні платформи розробників (IDP), платформу, створену та підтримувану досвідченою командою платформних інженерів, яка налаштована та самообслуговується. Ці фреймворки приховують компоненти інфраструктури та розкривають спільні точки абстракції, інтерфейсів та автоматизованих систем, що дозволяє розробникам створювати, тестувати, розгортати та контролювати програми у спрощений спосіб. Замість того, щоб нав'язувати розробникам проблеми низького рівня працездатності, IDP забезпечують ручно налаштований досвід розробника (DevEx), який є швидким, безпечним та послідовним. Впровадження IDP набирає обертів в організаціях будь-якого розміру, особливо у великих масштабах у 2024 році. Платформна інженерія дозволяє компаніям розділяти проектування інфраструктури відповідно до потреб розробки продукту, створюючи чітке розмежування між командою DevOps та командою продукту, тим самим зменшуючи розумове навантаження та дозволяючи компаніям більше зосередитися на отриманні цінності за допомогою підходу, орієнтованого на продукт. У цьому документі розглядається

структура, переваги та шлях IDP у ширшому середовищі DevOps, де платформна інженерія служить рушійною силою, покращуючи як розробку програмного забезпечення, так і операції.

Принципи та основні концепції платформної інженерії

Інженерія платформи стосується практики розробки, експлуатації та підтримки внутрішніх платформ розробників (ВРП), які дозволяють командам розробників програмного забезпечення самостійно обслуговувати свої потреби в інфраструктурі, одночасно впроваджуючи стандартизацію та абстракцію складності базової інфраструктури [4]. По суті, інженерія платформи керується побудовою на повторному використанні, компонуванні, автоматизації, стандартизації та продуктовому мисленні.

Інженерія розглядає платформу як продукт з користувачами, враховуючи потреби цих користувачів, насамперед розробників, як основу для ретельного проектування та негайного зворотного зв'язку. Метою є розробка зручного для розробників, масштабованого та надійного інтерфейсу до інфраструктури, який покращує як швидкість, так і якість доставки програмного забезпечення. Центральні принципи включають самостійне надання послуг, золоті шляхи (попередньо визначені робочі процеси), інфраструктуру як код (IaC) та чітко визначені угоди про рівень обслуговування (SLA) між командами платформи та додатків.

Еволюція від DevOps до платформної інженерії

DevOps трансформував світ програмного забезпечення, зруйнувавши стіни, що розділяли світ розробки та експлуатації. Однак, зі зростанням складності та впровадженням хмарних дизайнів, фахівці DevOps часто стикалися з проблемою розростання інструментів, розбіжних середовищ та відсутності єдиних практик. Наступним логічним кроком у цій еволюції є платформна інженерія [5]. У той час як DevOps заохочує ідею спільної відповідальності, практика платформної інженерії кодифікує цю відповідальність через конкретні ролі та інфраструктуру, що призводить до масштабованості та узгодженості між командами. Платформна інженерія зосереджена на пом'якшенні больових точок, що виникли в стабільних операціях DevOps, шляхом визначення стандартизованих

інструментів, API та сервісів, які розробники можуть використовувати без необхідності глибоких операційних знань.

Важливість внутрішніх платформ розробників (IDP)

Будівельним блоком інженерії платформи є внутрішні платформи розробників (ВПР). Такі платформи не є готовими продуктами, а є вузькоспеціалізованими для відображення потреб організації та її конкретних вимог, обмежень та робочих процесів. Врятована ВПР надає розробникам інтуїтивно зрозумілий варіант самообслуговування для створення, розгортання, моніторингу та захисту програм, приховуючи базову інфраструктуру. Таким чином, ВПР можуть значно скоротити час, необхідний для впровадження, оскільки платформа вже має вбудовані найкращі практики та стандарти відповідності, а також автоматизацію, що дозволяє частіше розгортати та підтримувати узгодженість у різних середовищах. Вони також дозволяють прийняти культуру «створи та запусти», де операційні обов'язки будуть полегшені розробникам. Коротко кажучи, ВПР допомагають змінити спосіб роботи розробників завдяки покращеній масштабованості багатоплатформних систем, підвищенню надійності системи та швидкості реагування бізнесу.

Ключові зацікавлені сторони: команди платформи **проти** *команд розробників*

Розробка платформи є успішною лише за умови чіткої співпраці між командами розробників платформи та командами розробників. Розробка та підтримка IDP доручаються командам розробників платформи, які поступово покращують її функціональність відповідно до відгуків, отриманих від розробників. Вони розглядаються як власники інфраструктурного продукту та забезпечують безпеку, масштабованість та прив'язку платформи до цілей організації [6]. Однак споживачами IDP є команди розробників. Вони використовують платформу для автоматизації своїх процесів, включаючи внесення коду до розгортання у виробничому середовищі. Ці дві групи потребують міцних робочих відносин; розробникам необхідно надати право власності на розробку в межах їхньої структури, але в межах параметрів, встановлених платформою. Інженерів платформи, у

свою чергу, потрібно заохочувати до прийняття клієнтоорієнтованого дизайну, розвиваючи платформу відповідно до змін потреб розробників. Це також підвищує продуктивність, зменшує операційне навантаження та покращує узгодженість у конвеєрі розробки програмного забезпечення завдяки такій співпраці.

1.2 Практики DevOps та Platform Engineering у створенні IDP

Розробка та експлуатація (DevOps) – це набір практик, спрямованих на оптимізацію процесів розробки, випуску та експлуатації програмного забезпечення. Концепція DevOps зумовлена необхідністю скоротити час, необхідний для впровадження змін у систему, одночасно забезпечуючи високу якість. Цей підхід визнає, що неефективність цих процесів часто спричинена організаційним розколом між командами розробки та експлуатації. DevOps сприяє спільному підходу, який зосереджений на комунікації та інтеграції між цими командами для досягнення швидшої та надійнішої доставки програмного забезпечення. Руйнуючи ізоляцію та сприяючи культурі співпраці, DevOps дозволяє організаціям долати труднощі, пов'язані з традиційними моделями розробки та експлуатації, та швидше та ефективніше надавати високоякісне програмне забезпечення [7].

DevOps – це відносно новий підхід, що виник у відповідь на виклики, з якими стикаються сучасні процеси розробки програмного забезпечення та експлуатації. В результаті його швидкої еволюції, досі точаться численні дискусії та обговорення щодо фактичного визначення DevOps. Дехто стверджує, що це філософія, яка наголошує на співпраці, автоматизації та постійному вдосконаленні [5], тоді як інші розглядають його як набір практик, що зосереджені на інтеграції команд розробки та експлуатації для оптимізації процесу доставки програмного забезпечення [1, 3]. Незважаючи на відсутність чіткого та послідовного визначення, DevOps отримав широке визнання як спосіб вирішення неефективності та вузьких місць, які часто виникають у розробці та експлуатації програмного забезпечення [8].

DevOps може надати організаціям кілька переваг, включаючи покращену

якість програмного забезпечення, швидший вихід на ринок, підвищену гнучкість та кращу задоволеність клієнтів. Роблячи акцент на співпраці та тестуванні, DevOps дозволяє організаціям створювати високоякісне програмне забезпечення з меншою кількістю дефектів, що призводить до зниження витрат та підвищення задоволеності клієнтів. Крім того, зосередження на автоматизації та оптимізації процесів призводить до швидшого виходу на ринок та підвищеної гнучкості, що дозволяє організаціям швидко реагувати на зміну ринкових умов та потреб клієнтів.

Принципи

Хоча єдиного, остаточного набору принципів DevOps не існує, існує кілька загальноприйнятих принципів, якими організації керуються в своїх DevOps-ініціативах [3]. До них належать побудова культури співпраці та комунікації між командами розробки та експлуатації, використання автоматизації для оптимізації та пришвидшення процесів доставки програмного забезпечення, а також впровадження постійного тестування та моніторингу для забезпечення високої якості програмного забезпечення. Інші ключові принципи DevOps включають руйнування ізольованих структур та сприяння роботі міжфункціональних команд, прийняття змін та експериментів, а також зосередження на комплексному досвіді клієнтів. Дотримуючись цих принципів, організації можуть досягти більшої гнучкості, швидшого виходу на ринок та покращеної якості програмного забезпечення, що зрештою забезпечить більшу цінність для своїх клієнтів [1 , 3].

Галл та Піньї пропонують концептуальну основу для DevOps, яка складається з трьох основних категорій: безперервна співпраця, безперервна автоматизація та безперервний моніторинг [9]. Ця основа буде використана як джерело принципів DevOps у цій роюоті з підтримкою з інших джерел. Основні принципи зображено на рисунку 1.1.

Безперервна співпраця – це перший принцип DevOps, представлений Галлом та Піньї, який зосереджується на необхідності співпраці та командної роботи між розробниками та операційними командами [3]. Співпраця сприяє культурі спільної відповідальності та підзвітності, руйнуючи ізольованість та

сприяючи міжфункціональній комунікації [10]. Вона заохочує впровадження гнучких методологій, де зворотний зв'язок враховується протягом усього -процесу розробки, сприяючи постійному вдосконаленню. Співпраця також вимагає використання інструментів та технологій, що забезпечують безперебійну співпрацю, таких як системи контролю версій, конвеєри безперервної інтеграції (CI) та безперервної доставки (CD), а також гнучкі інструменти управління проектами.

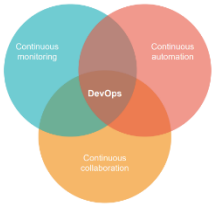


Рисунок 1.1 - Ключові принципи DevOps.

Другий принцип – це безперервна автоматизація. Вона передбачає автоматизацію ручних процесів для підвищення ефективності, зменшення кількості помилок та усунення повторюваних завдань. Автоматизація відіграє життєво важливу роль у досягненні швидкого зворотного зв'язку та підходів, що дозволяють уникнути відмов [3]. Принцип швидкого реагування на відмову полягає в тому, що чим раніше помічено помилку чи проблему, тим дешевше її виправити. Завдяки неперервній інтеграції (CI) та компактній інтеграції (CD) розробники можуть швидко виявляти та виправляти проблеми в застосунку, зменшуючи ризик дефектів у виробництві. Тестування також є важливим аспектом автоматизації, що дозволяє проводити ретельне тестування протягом усього циклу розробки. Процес розгортання також можна автоматизувати, що призводить до швидших та надійніших релізів [11].

Останній принцип – це безперервний моніторинг. Безперервний моніторинг є важливою практикою в методології DevOps, яка включає безперервний збір та спостереження за показниками протягом усього життєвого циклу розробки та експлуатації програмного забезпечення [3]. Метрики надають уявлення

про продуктивність та стан програмної системи та дозволяють командам виявляти та вирішувати проблеми проактивно. Прозорість є ключовим аспектом моніторингу, оскільки вона гарантує, що всі члени команди мають доступ до однієї й тієї ж інформації та можуть ефективно.

Галузь розробки програмного забезпечення з часом розвивалася, потреба в ефективніших та результативніших методах роботи стала більш важливою, а необхідність співпраці між командами стала очевидною. Організації зараз зосереджені на швидкому та безперервному наданні високоякісного програмного забезпечення, а також на управлінні складнощами, пов'язаними зі швидким зростанням та змінами. Основна концепція DevOps обертається навколо набору практик та інструментів, які дозволяють командам розробки та операцій тісніше співпрацювати, тим самим підвищуючи продуктивність, скорочуючи час виведення на ринок та покращуючи загальну якість програмного забезпечення [1].

Однак, досягнення бажаного рівня співпраці та ефективності в DevOps вимагає впровадження нових методів роботи. Ці методи роботи розроблені для забезпечення безперебійного робочого процесу протягом усього життєвого циклу розробки програмного забезпечення, від планування та кодування до розгортання та моніторингу. У цьому розділі буде розглянуто чотири ключові методи роботи, які довели свою ключову роль в успішному впровадженні DevOps: неперервна інтеграція та компактна інфраструктура, мікросервіси, інфраструктура як код (IaC) та GitOps.

Ці методи роботи не є винятковими для DevOps, але й широко використовуються спільнотою розробників програмного забезпечення завдяки їхній здатності оптимізувати процеси, покращувати співпрацю та підвищувати якість програмного забезпечення, що постачається. Натомість вони доповнюють один одного та, використовуючи їх разом, можуть значно підвищити загальну ефективність середовища DevOps. Розуміючи та впроваджуючи ці практики, організації можуть розкрити весь потенціал DevOps та створити більш гнучке, стійке та спритне середовище розробки програмного забезпечення. Важливо зазначити, що ці практики не завжди є правильним рішенням, їх можна адаптувати та

модифікувати відповідно до конкретних потреб та вимог певної організації чи проекту [1].

Безперервна інтеграція (CI) та безперервна доставка (CD) – це тісно пов'язані, але різні практики, які відіграють вирішальну роль в успішному впровадженні принципів DevOps [12]. Хоча вони мають спільні цілі, CD розширює CI, автоматизуючи подальші етапи процесу розгортання. CI передбачає часту інтеграцію змін коду в спільний репозиторій та автоматичне тестування цих змін, тоді як CD зосереджується на автоматичному розгортанні цих змін у продакшені безпечним та ефективним способом. Разом CI та CD забезпечують швидку ітерацію та експериментування з швидким та простим способом розгортання [8].

Для впровадження CD організації повинні спочатку створити міцну основу для неперервної інтеграції (CI). Це передбачає використання спільного та централізованого репозиторію, де налаштовано автоматизацію, яка автоматично тестує та перевіряє всі зміни. Зазвичай успішний запуск CI необхідний, перш ніж зміни можна буде об'єднати з основною гілкою. CI зазвичай розглядається як конвеєр, що складається з різних кроків або завдань. Ці кроки можуть включати статичні перевірки файлів, модульні або інтеграційні тести або наскрізні тести. В ідеалі результат конвеєра показує, чи код функціонує належним чином і чи відповідає код встановленим стандартам [13]. Конвеєр має бути розроблений таким чином, щоб надавати зворотний зв'язок якомога швидше, оскільки виявлення помилок на ранніх стадіях процесу зменшує час і зусилля, необхідні для їх виправлення [8].

Розгортання на компакт-дисках (CD) – це процес автоматичного розгортання певних змін у продакшені. Традиційно процес розгортання був рідкісним і повільним. CD має на меті зробити цей процес простішим і відтворюваним. CD зазвичай складається з таких кроків, як збірка, перевірка та розгортання. Фактична реалізація сценаріїв розгортання сильно залежить від використовуваних технологій та середовищ . У деяких випадках розгортання може бути таким простим, як просто копіювання файлів у певне місце, в інших випадках може бути багатоетапний процес збірки та складні розгортання [1].

Впровадження ефективних конвеєрів неперервної інтеграції (CI) та розподіленої інтеграції (CD) вимагає зосередження на послідовності та постійному вдосконаленні. Команда повинна узгодити загальні правила, яких дотримуватися конвеєр. Повинна бути можливість модифікувати конвеєри в стислі терміни, щоб адаптуватися до змін у бізнес-логіці [8]. В ідеалі, система неперервної інтеграції (CI) та розподіленої інтеграції (CD) повинна надавати можливість перевіряти та розгортати будь-які зміни, як того бажають розробники. Етапи перевірки повинні показувати, чи спричиняють зміни проблеми, і якщо так, надавати зворотний зв'язок розробнику. За допомогою належного моніторингу та спостереження можна помітити проблему в розгортанні, і за потреби має бути можливим відкат до відомої робочої версії в стислі терміни [14].

Архітектура мікросервісів у наш час є досить поширеним підходом до створення програмного забезпечення, який фокусується на використанні невеликих, окремо розгортаних сервісів, що працюють разом для формування більшого застосунку. Мікросервіси часто організовані навколо бізнес-можливостей, а не технічних питань, і розроблені для легкої зміни та обслуговування. Однією з ключових переваг архітектури мікросервісів є покращена масштабованість,ожен мікросервіс можна масштабувати незалежно від інших, що дозволяє краще використовувати ресурси та може призвести до зниження витрат. Ще однією перевагою є краща ізоляція несправностей, оскільки збої в одному мікросервісі з меншою ймовірністю вплинуть на решту системи. Мікросервіси також можуть забезпечити швидші цикли випуску, коли зміни в окремих сервісах можна розгортати незалежно. [8, 9].

Мікросервіси також створюють нові виклики. Однією з найбільших проблем є підвищена складність керування великою кількістю сервісів. Розподілене управління системою стає складнішим, оскільки комунікація між сервісами потребує ретельного управління. Крім того, кожен мікросервіс повинен мати чітко визначену відповідальність та чіткі межі сервісу. Комунікація між мікросервісами повинна базуватися на легких протоколах, таких як REST або обмін повідомленнями, і кожен мікросервіс повинен мати власне сховище даних, дані якого

доступні лише через API сервісу. Успішне керування архітектурою мікросервісів вимагає автоматизації розгортання та тестування. [9] Автоматизація має вирішальне значення для забезпечення послідовного розгортання кожного мікросервісу, а зміни в окремих сервісах не впливають негативно на решту системи. Тестування також є важливим, оскільки кожен мікросервіс має бути ретельно протестований перед розгортанням, щоб переконатися, що він працює правильно та не призводить до регресій [15] .

Організаційні міркування також важливі для успіху мікросервісної архітектури. Структуру команди та моделі комунікації необхідно ретельно продумати, оскільки культура співпраці та постійного вдосконалення необхідна для адаптації до змінних вимог та технологій. Також важливо враховувати компроміси та проблеми архітектури мікросервісів, і це може бути не найкращим підходом для кожної програми чи організації [9]. Для вирішення деяких проблем архітектури мікросервісів з'явилися різні моделі та методи. Шлюзи API можуть забезпечити єдину точку входу для всіх запитів клієнтів і можуть виконувати такі функції, як автентифікація та обмеження швидкості. Виявлення сервісів може використовуватися для автоматичного пошуку та підключення до інших сервісів. Автоматичні вимикачі можуть використовуватися для запобігання каскадним збоям шляхом ізоляції сервісів, що вийшли з ладу.

Інфраструктура як код (LAC) – це підхід до управління компонентами та конфігураціями інфраструктури як кодом. Це дозволяє контролювати, тестувати та розгортати код, а також інфраструктуру так само, як і програмний код [10, 11]. IaC надає кілька переваг, включаючи покращену узгодженість, підвищену видимість, відтворюваність та масштабованість . IaC також спрощує керування складними середовищами та внесення змін до інфраструктури, зберігаючи при цьому узгодженість та надійність. Для впровадження доступні кілька інструментів IaC, наприклад, Puppet, Chef та Ansible. У хмарних середовищах для управління інфраструктурою можна використовувати такі інструменти, як AWS CloudFormation, Terraform та Ansible [16].

До IaC слід ставитися як до розробки програмного забезпечення, і слід

дотримуватися найкращих практик у розробці. В іншому випадку зміни в інфраструктурі можуть призвести до помилок та невідповідностей , які важко виявити та виправити [1]. В IaC добре застосовувати модульний дизайн коду, оскільки це дозволяє використовувати спільний код, що спрощує внесення змін та оновлень. Замість того, щоб змінювати код для всіх сервісів, можна просто оновити спільний код, і всі сервіси оновляться. [11]

GitOps - це методологія розробки та експлуатації, яка використовує систему керування версіями (зазвичай Git) як основне джерело достовірних даних для декларативної інфраструктури [12]. Термін був введений Weaveworks у 2017 році як спосіб керування кластерами Kubernetes. Відтоді термін набув популярності як методологія для управління складними розподіленими системами в масштабі, включаючи хмарні додатки та мікросервіси [17]. Спрощений процес GitOps описано на рисунку 1.2. Управління змінами представляє запити на зміни, відкати та інші концепції контролю версій.

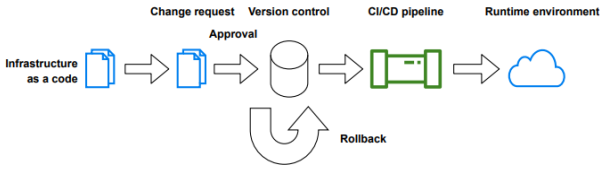


Рисунок 1.2. Загальний огляд процесу GitOps

У GitOps зміни в інфраструктурі та додатках управляються з використанням найкращих практик Git , що забезпечує структурований підхід до управління змінами. Ці структури – це гілки, теги та запити на зміни зі схваленнями. Це гарантує, що зміни перевіряються та схвалюються відповідним органом перед їх застосуванням [15]. Перед об'єднанням змін у репозиторій можна виконувати автоматичні перевірки, щоб переконатися, що вони відповідають заздалегідь визначеним правилам і стандартам, таким як лінтинг та перевірки безпеки. Після об'єднання змін запускаються автоматизовані конвеєри доставки для розгортання змін у цільовому середовищі, гарантуючи, що фактичний стан інфраструк

тури відповідає бажаному стану, зазначеному в репозиторії Git. Цей процес можна спостерігати, а інструменти моніторингу та реєстрації використовуються для забезпечення зворотного зв'язку в режимі реального часу про стан інфраструктури та виявлення проблем у міру їх виникнення. Ці автоматизовані перевірки та розгортання допомагають забезпечити узгодженість, повторюваність та можливість аудиту змін, сприяючи культурі співпраці та прозорості серед членів команди [18].

Перевагами GitOps є підвищення швидкості та ефективності, більша узгодженість та надійність, покращена співпраця, покращена спостережливість, а також може забезпечити кращу безпеку та відповідність вимогам [13]. Співпраця сприяє перевірці коду та запитам на зміни, а зворотний зв'язок у режимі реального часу надається за допомогою інструментів моніторингу та реєстрації. Контроль доступу та журнали аудиту допомагають забезпечити дотримання політик та правил безпеки, а також надають запис про всі зміни, внесені до інфраструктури [12].

Впровадження GitOps створює деякі труднощі, які користувачі повинні враховувати перед його застосуванням. Ці труднощі включають складність управління кількома рівнями інфраструктури, залежність від зовнішніх сервісів та конфлікти контролю версій. Надмірна залежність від автоматизації може ускладнити усунення неполадок. Впровадження GitOps вимагає культурного зрушення в бік більш спільного та прозорого підходу. Щоб пом'якшити ці труднощі, команди повинні встановити чіткі процеси відповідальності та перевірки змін, мінімізувати зовнішні залежності та забезпечити достатнє навчання та підтримку членів команди. [19]

Впровадження DevOps в організацію пов'язане з широким спектром труднощів. Серед них – відсутність чітких кроків, проблеми з налаштуванням автоматизації та задоволення високого попиту на кваліфікований персонал. Часто виникають проблеми з управлінням проектами та ресурсами, покращенням комунікації та співпраці команд, а також внесенням значних культурних змін, необхідних для успішного впровадження. Процес впровадження DevOps в організації

часто обмежується відсутністю чітких та загальноприйнятих рекомендацій. Це ускладнює процес, роблячи незрозумілим, які практики слід впроваджувати та як трансформувати поточні робочі процеси [6]. Кожна організація має різну структуру та робочі процеси, що ускладнює створення загальних рекомендацій. Крім того, відсутність чітких рекомендацій ускладнює оцінку підвищення якості після впровадження DevOps .

Спільною цільовою оцінкою є частота розгортань у продакшені, але це лише один показник, який не відображає всю якість. [6] Впровадження DevOps передбачає прийняття нового підходу до процесів розробки програмного забезпечення, який вимагає від розробників широкого спектру навичок. Раптом усі розробники повинні вміти керувати системами та використовувати специфічні інструменти та практики, що є частиною культури DevOps. Це включає в себе здатність автоматизувати різні етапи розробки, тестування та розгортання, щоб зробити процеси швидшими та ефективнішими [4]. У дослідженнях було показано, що багатьом організаціям бракує автоматизації впровадження DevOps [4, 6, 7].

Для успішного впровадження DevOps необхідний значний культурний зсув в організації, відхід від традиційних ізольованих ролей до більш спільного та інтегрованого підходу. Ця трансформація може бути складною, оскільки вимагає змін у мисленні, процесах та практиках на всіх рівнях організації. Проекти часто стикаються з обмеженнями щодо доступних ресурсів та стислих термінів, що ускладнює повне впровадження методологій DevOps. Ці обмеження можуть перешкоджати прогресу та ефективності ініціатив [1, 4 , 7]. Потреба в безперебійній співпраці та комунікації між різними відділами, такими як розробка, операції та забезпечення якості, може бути складною для встановлення та підтримки. Це посилюється існуючими комунікаційними бар'єрами та адаптацією до нових способів співпраці. [4 , 7]

1.3 Підходи до проєктування та дослідження внутрішніх платформ розробника

Мета цього розділу - запропонувати принциповий підхід до проектування IDP, що задовольняє вимоги (a)-(c), викладені у вступі. Як зазначено в цьому розділі, наш підхід ґрунтується на попередньому досвіді деяких авторів, а також на інших дослідженнях гнучких практик та інструментів, як в освітній, так і в державному управлінні. Покроковий процес, якого ми дотримувалися, викладено нижче в загальних рисах.

- Виходячи з предметної області застосування та дотримуючись позначень, визначте архітектурні драйвери для IDP.
- Розробіть відповідні епічні елементи, що представляють архітектурні рушійні сили.
- Оберіть стратегію управління проектами та зіставте епічні показники з професійними показниками, що відповідають цій стратегії.
- Оберіть програмні інструменти, які можуть бути корисними для впровадження Epic-пакетів, та забезпечте відповідність між Epic-пакетами та інструментами. Політика вибору повинна враховувати відкрите програмне забезпечення, власницьке програмне забезпечення, веб-програмне забезпечення тощо.
- Два попередні зіставлення забезпечують бажаний IDP з точки зору того, хто що використовує та для якої частини обраної методології управління проектами.

Враховуючи пункти (a)-(c) вступу, а також досвід, отриманий в результаті розробки внутрішньопроменевих систем (ВПР) у двох конкретних областях.

Для галузей університетської освіти та державного управління виникли такі архітектурні драйвери. Перші три пов'язані з розробкою та управлінням процесами програмного забезпечення, і їх можна легко знайти як невід'ємну частину програмної інженерії в SWEBOK , як зазначено нижче. Останній є відповіддю на вимогу (a), викладену у вступі.

(1) Інтегрований розробник розробки (IDP) має забезпечувати співпрацю між учасниками проєкту розробки програмного забезпечення, ефективне управління проєктом, якість коду, автоматизацію процесів та ітеративну та поступову розробку високоякісного програмного забезпечення.

(2) План розробки проєктів (IDP) повинен підтримувати (принаймні) методологію управління проєктами та дозволяти призначати завдання та види діяльності ключовим професійним фігурам, присутнім у обраній методології. Як уже було обґрунтовано у вступі, ми обрали Scrum.

(3) План інтегрованого розвитку має бути легко адаптованим, зокрема, у відповідь на змінні потреби професійних діячів, зазначених у пункті (2).

(4) ІПР має включати основні програмні інструменти з відкритим кодом для підтримки реалізації завдань та діяльності, зазначених у (1), та для використання професійними діячами, зазначеними у (2).

Ця частина враховує архітектурні рушійні сили (1), частково (2) та (3). Зокрема, ми визначили вісім основних епічних елементів, які можна додатково згрупувати в чотири родини:

Управління проектами (Епік 1), Розробка та тестування основного програмного забезпечення (Епіки 2-4), Співпраця та документація (Епіки 5-6) та Адаптивність (Епіки 7-8). Зрозуміло, що діяльність, пов'язана з кожним із цих Епік, повинна мати кілька рівнів взаємодії під час розробки процесу. Ці взаємодії та інтеграції добре задокументовані в літературі. Що стосується пріоритетів цих Епік під час реалізації проекту, оскільки ми використовуємо методологію Scrum, їх найкраще враховувати в рамках цієї методології. Відповідно, цей пункт коротко розглянуто в наступному розділі. Деталі наведено нижче.

- E1 – Управління проектами, ми виділимо такі ключові моменти.

1) Забезпечте ефективне відстеження завдань.

2) Грантова підтримка для організації та визначення пріоритетів діяльності з розвитку.

3) Забезпечте візуалізацію стану проекту, виявлення вузьких місць та забезпечення постійного вдосконалення за допомогою панелей моніторингу та

аналітики.

- E2 – Контроль версій та управління конфігурацією , ми виділимо такі ключові моменти.

1) Забезпечте належне управління змінами коду з часом, використовуючи зворотний зв'язок.

2) Забезпечити спільне управління кодом, включаючи перевірку коду та відстеження змін, внесених членами команди, можливо, за допомогою метрик продуктивності системи, використання та помилок за допомогою інструментів моніторингу.

3) Забезпечте одночасну розробку та інтеграцію коду. Це вимагає підтримки стратегій розгалуження та об'єднання.

4) Забезпечити можливість призначення, визначення пріоритетів та відстеження проблем, помилок, тобто функціональних дефектів програмного забезпечення, які призводять до несправності програмного забезпечення або серйозних недоліків у продуктивності, а також запитів на функції для забезпечення своєчасного вирішення.

5) Забезпечте відповідність робочого процесу процесу розробки та порядку вирішення проблем у команді.

- E3 – Якість коду та статичний аналіз, ми виділимо наступні ключові моменти.

1) Виконуйте статичний аналіз коду та забезпечуйте дотримання стандартів кодування.

2) Підвищуйте якість коду та зручність його підтримки за допомогою автоматичної перевірки коду та зворотного зв'язку.

3) Надати ідентифікацію «запахів» коду, тобто симптомів у вихідному коді, які вказують на потенційну проблему в дизайні, вразливостей, тобто недоліків, присутніх у коді, які часто використовуються зловмисниками для отримання несанкціонованого доступу до мереж, крадіжки цінних і конфіденційних даних, компрометації систем організації та потенційних проблем з продуктивністю.

4) Надайте кількісну оцінку технічного боргу, тобто, у розробці програмного забезпечення та інших сферах IT, вартість (з точки зору часу) майбутньої переробки, коли обмежене рішення обирається замість кращого варіанту, який може зайняти більше часу.

- E4 - Автоматизація тестування та безперервна інтеграція, ми виділимо такі ключові моменти.

1) Забезпечити автоматизацію процесів збірки, тестування та розгортання, можливо, шляхом надання підтримки ланцюжка інструментів.

2) Впроваджуйте автоматизовану конфігурацію тестових наборів та запускайте їх як частину робочого процесу розробки.

3) Забезпечте своєчасне вирішення проблем за допомогою сповіщень та оповіщень про невдалі збірки або тести.

- E5 – Комунікація та співпраця, ми виділимо такі ключові моменти.

1) Забезпечте зв'язок та співпрацю в режимі реального часу між членами команди, навіть з можливим використанням платформ обміну повідомленнями або інструментів чату.

2) Надайте можливість ділитися фрагментами коду, файлами та документами всередині команди.

- E6 – Документація, ми виділимо такі ключові моменти.

1) Надати можливість створення та обміну знаннями про проект, проектними рішеннями, документацією та інструкціями.

2) Забезпечити спільне створення та підтримку проектів, документації API та посібників користувача, включаючи функції співпраці.

- E7 – Гнучка конструкція, ми виділимо наступні ключові моменти.

1) Модульність та розділення завдань. Дійсно, модульна конструкція дозволяє розбити програмне забезпечення на менші, цілісні одиниці, які можна гнучко рекомбінувати або замінювати. Такий підхід підтримує інтеграцію інструментів та бібліотек з відкритим кодом, сприяючи повторному використанню та зменшуючи зусилля на розробку.

2) Підтримка побудови шляхом конфігурації. Дійсно, використання

інструментів з відкритим кодом у конфігурованому фреймворку дозволяє системі адаптуватися до потреб команд, що постійно змінюються.

- E8 – Гнучка експлуатація, ми виділимо такі ключові моменти.

1) Адаптивність в операційних середовищах. Зокрема, гнучка експлуатація вимагає ефективного функціонування програмного забезпечення в різних контекстах розгортання, включаючи різне обладнання, платформи та потреби користувачів.

2) Використання систем моніторингу та зворотного зв'язку для постійного вдосконалення. Дійсно, інтеграція моніторингу

Інструменти та механізми відстеження дозволяють системі адаптуватися до змінних операційних умов, надаючи інформацію про продуктивність, моделі використання та потенційні проблеми. Власно розміщені механізми моніторингу пропонують гнучкість та економічну ефективність, водночас узгоджуючи це з гнучкими принципами прозорості та оперативності реагування.

Ця частина враховує архітектурний драйвер (2). Як вже передбачалося та обґрунтовувалося у вступі, ми використовуємо Scrum. Далі ми наводимо короткий опис його професійних показників, того, як дії, згадані в попередньому розділі, пріоритетні в рамках методології, та як залучені сторони спілкуються. Наша презентація базується на довідковому посібнику [14], де також детально обговорюються додаткові деталі, зокрема щодо ролей, обов'язків та їх можливого дублювання. Крім того, корисно також згадати з цієї книги, що при застосуванні методології Scrum реалізація програмного проекту здійснюється за допомогою серії програмних інкрементів, які називаються спринтами. Спринт має таку ж структуру з точки зору дій, а процес інкрементів можна представити як ітерацію через той самий цикл. Зокрема, на рисунку виділено основні дії, де для стислості та ясності викладу ті, що стосуються управління проектами, не включені через їхню поширеність [20].

Далі йдуть професійні показники.

- Команда Scrum. Це колективна фігура, для стислості відома як ST, що вказує на сукупність професійних осіб, що складають команду. Конкретні

ролі та обов'язки цих осіб детально описані далі. Окрім них, вся команда відповідає за Epics E7 та E8.

- Власник Продукту. Така фігура, яку для стислості називають Власником Продукту (PO), є членом Scrum-команди та відповідає за вимоги до продукту, пріоритети їх реалізації та якість кінцевого продукту. Тому вона/його повинна бути залучена до завдань, що належать до більшості Епічних Розділів, зокрема з відповідальними ролями в Епічних персонажах E2, E3, E5 та E6.
- Зацікавлена сторона. Ця фігура – це особа або група осіб, для стислості позначених як S, які є зовнішніми по відношенню до Scrum-команди, мають конкретні інтереси щодо продукту та знання, необхідні для розробки продукту. Зацікавлені сторони представлені PO в Команді та можуть активно взаємодіяти зі Scrum-командою під час огляду спринту. Тому, така фігура має бути задіяна у завданнях, що належать до Епік E5-E6.
- Розробник. Ця фігура, для стислості відома як Розробник, є членом Scrum-команди та разом зі своїми колегами-розробниками займається створенням цінного інкременту у Спринті, незалежно від спеціалізації. Таким чином, така фігура бере участь у завданнях, що належать до всіх Епік, , зокрема з відповідальною роллю в Epic E4.
- Scrum Master. Ця фігура, для стислості відома як SM, є членом Scrum-команди та відповідає за допомогу, коучинг, навчання та координацію команди та її діяльності відповідно до методології Agile Scrum. Таким чином, він/вона відповідає за Epic E1.

Щодо епічних завдань, описаних у попередньому розділі, їх пріоритезація керується конкретними цілями та вимогами проекту, при цьому базові дії, такі як управління проектами (Епік 1) та контроль версій (Епік 2), зазвичай мають пріоритет через їхню важливу роль у створенні структурованої та добре регульованої платформи розробки. Що стосується епічних завдань 3-6, нагадуючи структуру циклу розробки Scrum на рисунку, вони є частиною ітеративних спринтів Scrum. Постачальник продукту (PO) відповідає за створення та оновлення Беклогу продукту, тобто списку, відсортованого за пріоритетом, усіх робіт, які

необхідно виконати для створення, підтримки та підтримки продукту. ST, плануючи спринт, вибирає дії в Беклозі відповідно до тривалості спринту та дотримуючись пріоритетів. Сімейство адаптивності (Epics 7 8) відіграє допоміжну роль, забезпечуючи відповідність еволюції платформи потребам ST та вимогам до продукту, але його ефективність залежить від основи, встановленої іншими Epic. Для того, щоб зробити такий план діяльності більш детальним, може бути корисною методологія Essence [35] .

Координація, щоб забезпечити безперебійність робочого процесу та належну комунікацію, забезпечується зустрічами Спринту. У цьому відношенні Спринт-менеджер має відповідну та добре задокументовану [14, 36]). Наприклад, Спринт-менеджер сприяє та забезпечує продуктивність зустрічей, наприклад, огляду Спринту та ретроспективи Спринту. Essence може бути досить корисним для досягнення такої мети.

Для управління проектами розробки програмного забезпечення доступний широкий вибір базових інструментів, спрямованих на підтримку Agile-команд у досягненні їхніх цілей. Оскільки нас цікавлять інструменти розробки програмного забезпечення, які є локальними та з відкритим вихідним кодом, такі вимоги виключають усі пропрієтарні системи, наприклад, продукти Atlassian (Jira [21] та Trello [37]), Confluence [38] , BitBucket [39]) або GitHub [40] .

Зосереджуючись на інструментах з відкритим кодом, ми застосовуємо такі критерії.

- Вибір. Оберіть інструменти, які найкраще підтримують роботу, пов'язану з Epics E1-E8, надаючи перевагу тим, які популярні в Agile-спільноті.
- Реалізація відбору з академічними стандартами. Наскільки нам відомо, хоча доступні академічні дослідження, що містять порівняльний аналіз програмних інструментів, що підтримують Agile, наприклад, єдиним академічним дослідженням, яке можна використовувати з точки зору критерію відбору, описаного раніше, якого ми тут дотримуємося. Оскільки наш вибір певного інструменту має на меті проілюструвати проектування та реалізацію IDP, а не вказати на «найкращий» у відповідній категорії для IDP, ми також включаємо

посилання на альтернативні інструменти та порівняльний аналіз, щоб читач міг навіть вибрати інструменти для власної реалізації IDP. Для повноти картини зазначимо, що комерційний світ пропонує інформацію про оцінку користувачами, яка може бути цікавою для розробників IDP.

Далі повідомляється про обрані інструменти із зазначенням того, які завдання Epic вони мають підтримувати.

- Трекери продуктивності на рівні коду. Один з ключових

Особливістю нашої методології проектування є надання інструментів, що дозволяють відстежувати продуктивність коду, як з точки зору створеного коду, так і з точки зору статистики, за допомогою візуалізації даних. Програмні інструменти, що використовуються для цієї мети, описані далі.

- Логер. Це унікальний програмний інструмент, що добре підходить для epic E1, спеціально розроблений для запропонованого тут дизайну. Це оновлена версія інструменту [22]. Він дозволяє відстежувати продуктивність команди розробників з точки зору кількості рядків коду, які вставляються або видаляються певним програмістом у заданому файлі. Він має панель інструментів, яку можна використовувати для надання синоптичного опису всіх зібраних даних, тобто для кожного програміста або для кожної команди. Хоча цей інструмент дуже простий, він дає певний показник продуктивності, корисний для команди. Він використовує як сервер додатків Nginx, а як механізм баз даних MongoDB.

- Gitinspector. Це інструмент статистичного аналізу з відкритим кодом, добре підходить для Epic E1, спеціально розроблений для репозиторіїв Git (згаданих нижче). Він дозволяє користувачам перевіряти вихідний код проекту, надаючи детальний огляд метрик коду, наприклад, рядків коду, кількості комітів та добре відомої цикломатичної складності. Цей аналіз може дати чітке уявлення про кількість та якість роботи, виконаної командою, та дає змогу оцінити загальну якість виконаної роботи та визначити можливі області для покращення з точки зору спрощення та оптимізації коду. Це рішення використовує як сервер додатків Nginx, а як механізм баз даних MongoDB. Переваги та недоліки цього інструменту з точки зору взаємодії з користувачем обговорюються.

Альтернативою може бути GitStats [21].

- Gource. Це кросплатформний інструмент візуалізації з відкритим кодом для репозиторіїв системи контролю версії, добре підходить для epic E1. Він може запропонувати інноваційний спосіб відстеження процесу розробки програмного проекту з часом за допомогою візуального представлення, оскільки надає цінну інформацію про діяльність команди розробників, таку як інтуїтивна візуалізація, виявлення змін, аналіз шаблонів розробки та залученість команди. Це рішення використовує Nginx як сервер додатків, а MongoDB як механізм баз даних.

- Taiga. Це загальний інструмент управління проектами, який завдяки своєму інтерфейсу сприяє спільному управлінню проектами, природним чином доповнюючи інструменти продуктивності коду, згадані раніше. Taiga добре підходить для epic E1, оскільки забезпечує ефективне планування спринту на основі оцінок для конкретних ролей та першокласного управління завданнями, які необхідно виконати в S print Backlog. Він використовує Nginx як сервер додатків, а PostgreSQL як механізм баз даних.

- GitLab. Він надає репозиторій коду та середовище для спільної розробки програмного забезпечення для великих Agile-проектів. Він добре підходить для епічних проектів E2, E4, E5 та E6. Базований на системі контролю версій Git, він дозволяє підтримувати стандартні практики, що забезпечують якість програмного забезпечення та швидку доставку, наприклад, архівувати код онлайн, відстежувати проблеми та автоматизувати процеси безперервної інтеграції/доставки (CI/CD). Він використовує як сервер додатків Apache2, а як механізми баз даних PostgreSQL та Redis. Переваги та недоліки цього інструменту представлені, де також згадуються альтернативи.

- SonarQube. Це інструмент статичного аналізу коду, добре підходить для epic E3, який можна використовувати для виявлення помилок та тестування безпеки. Серед його ключових особливостей ми згадуємо можливість (1) безперервно вимірювати та контролювати якість у часі та (2) оцінювати технічний борг. Він використовує як сервер додатків Apache2, а як механізм баз даних PostgreSQL. Додаткові відомості доступні за посиланням [66] .

- Jenkins. Jenkins - це інструмент автоматизації, що добре підходить для епіс E4, який оптимізує процеси розробки програмного забезпечення, включаючи створення, тестування та розгортання додатків. Він підтримує різні інструменти контролю версій та може виконувати проекти на основі, наприклад, Apache Ant або Apache Maven, скриптів оболонки та пакетних команд Windows. Він використовує Apache2 як сервер додатків, а PostgreSQL як механізм баз даних.

- Mattermost. Це платформа обміну повідомленнями з відкритим кодом, що самостійно розміщується, розроблена для підтримки комунікації та співпраці команди розробників, що добре підходить для епіс E5. Вона забезпечує безпечну співпрацю для технічних та операційних команд, які працюють у середовищах зі складними вимогами до безпеки та довіри національного рівня. Вона використовує Nginx як сервер додатків, а PostgreSQL як механізм баз даних. Переваги та недоліки цього інструменту, а також його здатність до масштабування, де також представлені альтернативи.

- Draw.io. Це кросплатформний налаштовуваний інструмент для побудови діаграм, добре підходить для епіс E6, з вихідним кодом, який користувачі можуть переглядати та змінювати відповідно до своїх потреб. Він підтримує співпрацю в режимі реального часу, що полегшує віддаленим командам одночасну роботу над одним проектом [23]. Він містить багато попередньо визначених фігур та шаблонів і водночас є налаштовуваним. Він використовує як сервер додатків Apache2, а як механізми баз даних PostgreSQL та Redis.

- Docker. Docker - це платформа з відкритим кодом, яка спрощує створення, розповсюдження та запуск додатків, відокремлюючи їх від базової інфраструктури. Вона особливо добре підходить для реалізації мікросервісних архітектур, забезпечуючи ефективне розгортання та управління програмними системами.

- Ansible. Це інструмент для написання сценаріїв з відкритим кодом для Linux, розроблений для *інфраструктури як коду*. Зокрема, він підтримує налаштування, керування конфігурацією та оркестрацію завдань IT-адміністрування. У

нашому випадку це дозволяє автоматизувати процес створення, налаштування та керування всіма компонентами стеку контейнерів Docker, навіть на різних цільових хостах, за потреби для визначення бажаної архітектури IDP. У двох попередніх підрозділах наведено (1) зіставлення професійних показників із завданнями та (2) зіставлення програмних інструментів із завданнями. Фактично, інструменти використовуються професійними показниками як допоміжний засіб для виконання завдань, над якими вони працюють. Тому, виходячи з цих двох зіставлень, корисно надати короткий огляд того, «хто» використовує «що» та «коли», де останній термін стосується різних фаз, що характеризують Scrum Sprint. Такий короткий огляд виражає дизайн IDP для Agile. Ми також згадуємо адаптивність і, нарешті, наводимо приклад того, як різні інструменти можна використовувати синергетично.

Таблиця 1.1

Зіставлення епосів, професійних персонажів та інструментів.

Епік (EPIC)	**Відповідальний**	**Учасники**	**Інструменти**
E1	SM (Scrum Master)	PO, S, Dev	CAS Logger, Gitinspector, Gource, Taiga
E2	PO (Product Owner)	SM, Dev	GitLab
E3	PO (Product Owner)	Dev	SonarQube
E4	Dev (Developer)	PO, S	GitLab, Jenkins
E5	PO (Product Owner)	Dev, SM, S	Mattermost, GitLab, Draw.io
E6	PO (Product Owner)	Dev, S	GitLab, Draw.io
E7–E8	ST (Stakeholder/Tester*)	PO, SM, Dev	Ansible, Docker

Цю частину можна легко представити за допомогою таблиці, що нагадує матрицю RACI, тобто таблиці 1.1. Звертаючись до нього, зліва направо, у першому стовпці ми знаходимо окремий Епік. У другому та третьому стовпцях ми знаходимо фахівців Scrum, які, відповідно, представляють відповідальних за Епік, залучених у першому стовпці, та учасників процесів, що задіяні в рамках того ж Епіку. У четвертому стовпці ми знаходимо Agile-інструменти, що підтримують роботу згаданих фахівців. За винятком діяльності з управління проектами, яка є поширеною протягом кожного Спринту. Цифри 2–4 вказують

які використовуються на кожному етапі спринту.Як видно з перших 6 шести рядків таблиці 1.1 , інструменти можна легко видаляти, замінювати або додавати у відповідь на зміну потреб Scrum-фахівців або з урахуванням альтернатив нашим пропозиціям . Аналогічно, надають план Scrum Sprint з точки зору програмних інструментів, які будуть використовуватися, та які легко адаптуються до змін у таблиці 1.1. Як ці зміни фактично вносяться, згідно з останнім рядком таблиці 1.1 передбачає використання спеціального програмного забезпечення, зазначеного там. Технічні деталі щодо реального IDP обговорюються , також загальні інструкції щодо репозиторію проєктів [32], зокрема щодо репозиторію налаштувань та
частини, що стосується додавання та видалення сервісів у ньому.

Різні програмні інструменти, що використовуються під час розробки проекту, можуть потребувати доступу до інформації, доступної іншим інструментам. Типовий сценарій комунікації для кожного програмного інструменту є наступним.

- GitLab можна використовувати для запуску робочих процесів на Jenkins та для запуску статичного аналізу коду на SonarQube. Якщо ввімкнено, GitLab оновлює статус користувацьких історій на Taiga та може надсилати повідомлення на Mattermost (наприклад, щоб повідомити про збій робочого процесу).
- Тайга взаємодіє з GitLab лише для автентифікації єдиного входу (скорочено SSO).
- Jenkins взаємодіє з GitLab для публікації оновлень щодо стану робочих процесів та для автентифікації SSO. Він також зв'язується з SonarQube для виконання статичного аналізу коду (зауважте, що це альтернативний підхід, щоб GitLab самостійно обробляв зв'язок із SonarQube).
- SonarQube повідомляє про результати статичного аналізу коду за допомогою Jenkins та/або GitLab. Він також обробляє автентифікацію через GitLab SSO.
- Mattermost взаємодіє з GitLab для SSO-автентифікації.

Трекери продуктивності зчитують дані з GitLab, Taiga, SonarQube та Mattermost. Зв'язок з GitLab також дозволяє автентифікацію SSO.

1.3 Висновки по розділу

У першому розділі розглянуто теоретичні основи внутрішніх платформ розробника (Internal Developer Platforms). Проаналізовано їх призначення, ключові концепції та роль у сучасних підходах DevOps і Platform Engineering. Визначено основні принципи проєктування IDP та підходи до їх дослідження в контексті підвищення ефективності розробницьких процесів. Отримані результати формують теоретичну базу для подальшого аналізу архітектури та

реалізації таких платформ.

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ ПОБУДОВИ ВНУТРІШНІХ ПЛАТФОРМ РОЗРОБНИКА

2.1 Референтна архітектура внутрішніх платформ розробника

На рисунку 2.1 представлено багаторівневе зображення внутрішньої платформи розробника (IDP), яке корисне для орієнтації читачів у розділенні питань, властивих інженерії платформи. На верхньому рівні рівень досвіду розробника (DX) інкапсулює портали, шаблони та робочі процеси самообслуговування, які абстрагують складність інфраструктури та операційної діяльності від команд розробників додатків. Цей рівень розроблено для мінімізації когнітивних витрат, надаючи інтуїтивно зрозумілі інтерфейси для поширених завдань, таких як створення сервісів, розгортання додатків та доступ до документації. Для команд Java це зазвичай включає створення сервісних каркасів на основі Spring Boot, стандартизовані конфігурації збірки та тверді значення за замовчуванням, що кодують найкращі практики організації. Взаємодіючи переважно з цим рівнем, розробники можуть залишатися продуктивними, не вимагаючи глибоких знань у хмарній інфраструктурі або інструментах розгортання [24].

Площина керування платформою, розташована під рівнем, орієнтованим на розробника, координує оркестрацію, забезпечення дотримання політик та автоматизацію життєвого циклу. Цей рівень визначає, як сервіси створюються, тестуються, розгортаються та керуються в різних середовищах, часто за допомогою шаблонів конвеєрів повторного використання та механізмів «політика як код». Для платформ, орієнтованих на Java, площина керування керує кешами збірки, стратегіями вирішення залежностей та правилами просування артефактів, забезпечуючи узгодженість та відтворюваність між командами. Вона також діє як центр інтеграції, пов'язуючи дії розробників із системами CI/CD, сканерами безпеки та інструментами спостереження. Централізуючи ці можливості, площина керування дозволяє організаціям розвивати стандарти платформи незалежно від коду додатків, зменшуючи труднощі під час оновлень або змін політик.

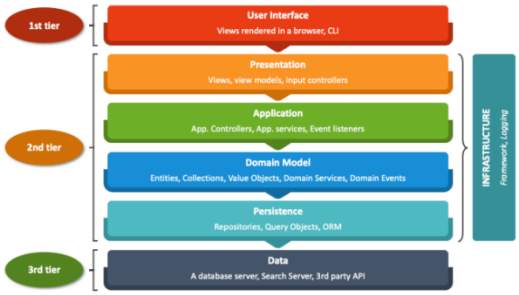


Рисунок 2.1 - Багаторівнева архітектура IDP

В основі лежить інфраструктурний рівень, який забезпечує середовище виконання для програм і платформних сервісів. Цей рівень зазвичай складається з кластерів Kubernetes, примітивів хмарних провайдерів, мережевих, сховищних та ідентифікаційних сервісів. IDP захищає команди розробників програм від прямої складності цього рівня, водночас надаючи необхідні можливості через чітко визначені API. Для сервісів Java абстракції інфраструктури підтримують контейнеризовані середовища виконання, горизонтальне масштабування та стандартизовані топології розгортання. Багаторівнева архітектура, показана на рисунку 2.1, ілюструє, як інженерія платформи забезпечує масштабованість та управління шляхом чіткого відокремлення робочих процесів розробника від проблем інфраструктури, принцип, який сильно підкреслюється в літературі з інженерії платформи та практиці DevOps [25].

На рисунку 2.2 показано роль Backstage як центрального порталу розробника та об'єднуючого інтерфейсу в IDP. Каталог програмного забезпечення Backstage надає структуроване представлення організаційних активів, включаючи сервіси, бібліотеки, API та компоненти інфраструктури. Для команд Java сутності каталогу зазвичай відповідають сервісам Spring Boot, спільним бібліотекам або пакетним завданням, кожне з яких збагачене метаданими, такими як власність, стан життєвого циклу та посилання на документацію. Ця централізована видимість вирішує поширену проблему у великих екосистемах Java, де сервіси Розрізненість та фрагментованість документації ускладнюють пошук та

розуміння системи в єдиному порталі. Backstage вирішує цю проблему, забезпечуючи доступність інформації та підзвітність між командами.

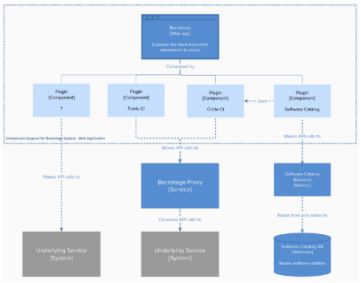


Рисунок 2.2 - За лаштунками порталу для розробників

Окрім каталогізації, архітектура Backstage на основі плагінів забезпечує тісну інтеграцію з площиною керування платформою та навколишнім інструментарієм. Плагіни можуть відображати стан конвеєра CI/CD, показники якості коду, результати сканування безпеки та індикатори справності виконання безпосередньо на порталі [26]. Для команд Java це означає, що збої збірки, звіти про вразливості або регресії продуктивності видно поряд з метаданими сервісів, що зменшує перемикання контексту та покращує цикли зворотного зв'язку. Шаблони сервісів ще більше розширюють ці можливості, дозволяючи розробникам створювати нові сервіси за допомогою керованих робочих процесів, які автоматично підключають репозиторії, конвеєри та маніфести розгортання. В результаті розробники можуть створювати сумісні, готові до роботи Java-сервіси з мінімальним ручним налаштуванням.

З точки зору платформної інженерії, Backstage виступає як інтерфейс користувача, так і механізм управління. У той час як розробники сприймають його як інструмент продуктивності, команди платформ використовують його для кодування стандартів та керування поведінкою за допомогою курованих шаблонів і плагінів. Ця подвійна роль узгоджується з філософією «платформа як продукт», де досвід розробників ставиться до першочергового значення. Як підкреслюється в галузевих аналізах та рекомендаціях постачальників, Backstage став фактичним стандартом для IDP саме тому, що він поєднує гнучкість з думкою, що робить

його особливо ефективним для великих масштабні організації Java, які прагнуть узгодженості без шкоди для автономії.

На рисунку 2.3 наведено загальний огляд ландшафту інструментарію платформи, що відображає такі поширені категорії, як CI/CD, інфраструктура як код, GitOps, управління секретами та спостережуваність. Ця екосистемна перспектива є важливою для розуміння того, що IDP - це не окремий продукт, а сукупність інструментів, інтегрованих за цілісною абстракцією. Для команд Java інструментарій пояснює, як знайомі технології, такі як Maven або Gradle, вписуються в ширший конвеєр розробки, який охоплює збірку, розгортання та операції виконання. Візуалізуючи ці категорії, малюнок допомагає командам обмірковувати архітектурні компроміси та уникати спеціального вибору інструментів.

У цьому ландшафті поширені Java-орієнтовані інструментальні ланцюжки виникають як еталонні шаблони. Системи збірки, такі як Maven та Gradle, враховують створення образів контейнерів за допомогою таких інструментів, як build packs або kaniko, які добре підходять для середовищ, нативних для Kubernetes. Безперервна інтеграція та розгортання часто реалізуються за допомогою конвеєрів Tekton та контролерів GitOps, таких як Argo CD, що забезпечує декларативні, аудитовані робочі процеси доставки. Під час виконання Kubernetes оркеструє виконання сервісів, тоді як стеки спостережуваності на основі Prometheus та Grafana збирають метрики та трасування. Рисунок 3 контекстуалізує ці варіанти, показуючи, як кожен інструмент відіграє певну роль на платформі, а не існує як ізольоване рішення.

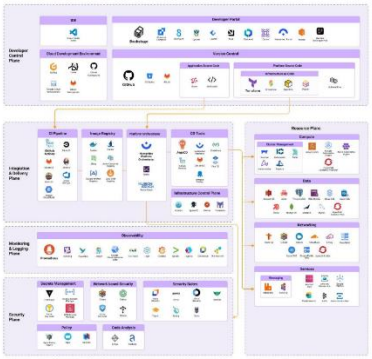


Рисунок 2.3 - Огляд інструментарію платформ

Важливо, що інструментарій наголошує на компонуванні та еволюції, а не на жорсткій стандартизації. Інженерія платформи не вимагає фіксованого набору інструментів; натомість вона визначає інтерфейси та контракти, які дозволяють замінювати або оновлювати інструменти з мінімальними збоями. Для команд Java ця гнучкість є критично важливою, оскільки екосистема продовжує розвиватися, наприклад, з впровадженням нативних образів або нових стандартів спостережуваності. Закріплюючи вибір інструментів у чітко визначеній архітектурі платформи, організації можуть адаптуватися до технологічних змін, зберігаючи при цьому узгодженість та надійність. Таким чином, рисунок 3 підкріплює центральний аргумент інженерії платформи: сталий темп виникає не внаслідок поширення інструментів, а внаслідок продуманої інтеграції та абстракції [27].

2.2 Архітектурні компоненти та структура IDP

IDP, Внутрішня платформа розробника (IDP), - це добре розроблена та створена система, яка інтегрує набір інструментів та послуг у єдине середовище для команд розробників програмного забезпечення. [28] IDP завжди унікальна для кожної організації, проте має кілька спільних ключових елементів для різних реалізацій, які співпрацюють для абстрагування складності інфраструктури, впровадження стандартів та досягнення самообслуговування .

Кожен постачальник ідентифікаційних даних (IDP) має в основі зручний для розробника інтерфейс (веб-портал, CLI або API), через який розробник може працювати з платформою без значних знань базової інфраструктури. Ці інтерфейси публікують середовища підготовки, розгортання служб, моніторинг стану та робочі процеси управління конфігурацією. Деякі компанії використовують такі інструменти, як Backstage, для надання порталів для розробників, які забезпечують паритет досвіду для всіх служб платформи.

IDP покладаються на безперервну інтеграцію та безперервне розгортання (CI/CD). Автоматизовані конвеєри поєднують процедури збірки, тестування та розгортання, щоб забезпечити швидку, надійну та відтворювану доставку. Ці

конвеєри зазвичай шаблонізуються та стандартизуються для різних команд з використанням таких інструментів, як GitHub Actions, GitLab CI, Jenkins або ArgoCD, та інтегровані з політиками безпеки, політиками відповідності та політиками продуктивності.

Інфраструктура як код (IaC) допомагає IT-фахівцям визначати та керувати ресурсами інфраструктури в програмному процесі. Програмне забезпечення IaC, таке як Terraform, Pulumi та Helm, робить середовище відтворюваним, відстежуваним за версіями та має журнал аудиту [29]. Цей компонент дозволяє розробникам розгортати середовища на вимогу, що призводить до загального зменшення тертя, операційних накладних витрат та помилок, що виникають вручну.

Потужний IDP забезпечує функціональність для керування середовищами розробки, проміжного тестування та виробництва. Це включає надсилання та розповсюдження ресурсів, контроль секретів та конфігурацій, а також дотримання правил ізоляції середовища. Динамічні або ефемерні середовища, що створюються автоматично під час створення гілки функції або запиту на зняття, стають поширеними в сучасних IDP для пришвидшення тестування та циклу зворотного зв'язку.

Щоб розробники могли працювати з аналітикою в режимі реального часу, IDP включають інструменти спостереження, які надають можливості метрик, ведення журналу та трасування. Інструменти, орієнтовані на розробників, такі як Prometheus, Grafana, Loki та OpenTelemetry, дозволяють розробникам спостерігати за станом, продуктивністю або збоями своїх програм, не покладаючись на команди інфраструктури. Вбудована спостереження є критично важливою під час налагодження, налаштування продуктивності та досягнення цілей рівня обслуговування (SLO).

Архітектура IDP рівноцінно пріоритезує як безпеку, так і відповідність вимогам. Авторизовані користувачі можуть отримувати доступ до ресурсів інфраструктури та змінювати їх, оскільки контроль доступу на основі ролей (RBAC) запобігає неавторизованим користувачам. Платформа включає управління секретами, забезпечення дотримання політик (наприклад, за допомогою

OPA/Gatekeeper), сканування вразливостей та ведення журналу аудиту для дотримання внутрішніх та нормативних вимог.

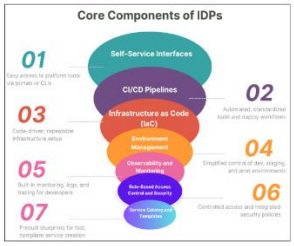


Рисунок 2.4 - Основні компоненти внутрішніх платформ розробників (IDP)

Спеціальний каталог шаблонів сервісів багаторазового використання дозволить стандартизувати створення та розгортання сервісів в організації. Мікросервіси, бази даних, API або креслення фронтенд-додатків можуть бути попередньо визначені, що дозволить розробникам легко вибирати їх, гарантуючи, що всі нові сервіси будуть створені правильно та відповідно до найкращих практик і нормативних вимог [30].

Внутрішня платформа розробника (IDP) ретельно розроблена, щоб об'єднати кілька рівнів та сервісів, які взаємодіють один з одним для забезпечення безперебійного взаємодії з клієнтами, включаючи інфраструктуру та конвеєри розгортання, а також можливості самообслуговування для розробників. Найпоширеніші модулі та історичні взаємодії IDP виробничого рівня. Ці рівні логічно організовані в архітектурі наступним чином: рівень інтерфейсу розробника, рівень оркестрації платформи, рівень автоматизації інфраструктури, інфраструктура розгортання та спостереження та управління. Рівень інтерфейсу розробника зверху служить основною точкою доступу для команд розробників програмного забезпечення. Серед них портали розробників, документація, шаблони та CLI/API, які приховують складність за інфраструктурою та робочими процесами. Використовуючи цей рівень, розробники запускають такі дії, як розгортання або налаштування середовища, і цей рівень інформує нижній рівень, рівень оркестрації платформи. Рівень оркестрації платформи діє як мозок або ядро IDP. Він

містить серце механізму оркестратора, який здійснює надання ресурсів, виконання конвеєра CI/CD та визначення сервісів [31]. Каталог послуг, Засіб забезпечення середовища та Механізм обробки даних – це ключові модулі, які обробляють стандартизовані робочі процеси та забезпечують узгодженість, а також обчислювальні середовища. Цей рівень служить інтерфейсом до репозиторіїв артефактів, систем керування версіями (наприклад, GitHub, GitLab) та сховищ, забезпечуючи безпечну та ефективну взаємодію для отримання або надсилання необхідних даних.

Рівень автоматизації інфраструктури отримує команди оркестрації платформи на високому рівні та перетворює їх на конкретну конфігурацію інфраструктури. Тут застосовуються механізми інфраструктури як коду (IaC), такі як Terraform або Pulumi, разом з інструментами керування конфігурацією, такими як Ansible або Chef. Він автоматично розгортає робочі навантаження на різні цільові об'єкти розгортання, включаючи локальні центри обробки даних, кластери Kubernetes або постачальників публічних хмарних послуг, таких як AWS, Azure або GCP. Нарешті, є рівень спостереження та управління, який пов'язує моніторинг, ведення журналу та забезпечення дотримання політик безпеки з IDP. Prometheus, Grafana, стек ELK та інструменти для збору та розкриття метрик і журналів, а також журнали аудиту (Open Policy Agent (OPA) та Kyverno) та базові механізми політик – це всі ці засоби, які допомагають підтримувати відповідність та управління. Ці компоненти не тільки надають інформацію в режимі реального часу та автоматичні сповіщення, але й покращують видимість, відстежуваність та безпеку всієї діяльності на платформі.

Внутрішні платформи розробників (IDP) – це не монолітні платформні рішення, а радше набір інструментальних ланцюжків, які поєднуються, щоб забезпечити комплексний досвід розробника. Ці інструментальні ланцюжки охоплюють кілька областей, включаючи забезпечення інфраструктури, управління кодом, [32] безперервну інтеграцію/безперервне розгортання, безпеку, спостережуваність та портали розробників. Інструменти вибираються на основі потреб, зрілості та розміру організації та часто створюються разом для підтримки всього

життєвого циклу розробки програмного забезпечення в автоматизований та самообслуговуваний спосіб.

Інструменти «інфраструктура як код» (IaC), такі як Terraform, Pulumi або Crossplane, зазвичай розгортаються для забезпечення інфраструктури та підтримки її забезпечення. Вони дозволяють командам декларативно розміщувати визначення на хмарних ресурсах, а потім послідовно надавати їх у різних середовищах. Для керування конфігурацією може бути виконана автоматизація всієї системи за допомогою інструментів керування конфігурацією, таких як Ansible, Chef або SaltStack. Конвеєри CI/CD включають такі движки, як Jenkins, GitHub Actions, GitLab CI/CD, ArgoCD або Tekton, які автоматизують процес виконання всього коду від компіляції до надсилання образів контейнерів для розгортання. Вимога цих інструментів для зберігання та розповсюдження результатів збірки пов'язана з репозиторіями артефактів, такими як Docker Registry, JFrog Artifactory або GitHub Packages.

Контроль вихідних кодів та співпраця зазвичай здійснюються на деяких платформах, таких як GitHub, GitLab або Bitbucket [33]. З боку розробників використовуються фреймворки порталів розробників, такі як Backstage, створений Spotify, для забезпечення внутрішніх порталів розробників, які надають каталоги послуг, документацію та засоби керування розгортанням. Prometheus, Grafana, Loki та OpenTelemetry використовуються як інструментальні ланцюжки спостережуваності. Натомість, Open Policy Agent (OPA), Kyverno або HashiCorp Vault використовуються для забезпечення дотримання політик та управління, а також для управління секретами. Такі технології, у поєднанні, забезпечують надійну та розширювану основу для створення IDP, що спрощує інфраструктуру та доставку додатків.

Поява хмарних технологій суттєво вплинула на дизайн та набір функцій внутрішніх платформ розробників (Internal Developer Platforms). Центральним елементом цієї екосистеми є Kubernetes, який перетворився на фактичний стандарт для оркестрації контейнерів [34]. Поточні IDP тісно інтегруються з Kubernetes для полегшення масштабованого та декларативного розгортання

мікросервісів та елементів інфраструктури. Kubernetes надає нативні API для керування робочими навантаженнями, які можуть бути використані рішеннями для керування ідентифікацією та доступом (IDP), що запроваджують динамічний розподіл ресурсів, забезпечення дотримання політик, автоматичне масштабування та стійкість. З'явилися технології Service Mesh, такі як Istio, Linkerd та Consul, щоб розширити цю оркестрацію завдяки багатогранному управлінню трафіком, спостережуваності та безпеці на мережевому рівні. IDP потім можуть включати функції Service Mesh, що спрощує такі завдання, як безпечний зв'язок між сервісами, розгортання за принципом «канарейка», розподіл трафіку та мережеві політики з нульовою довірою.

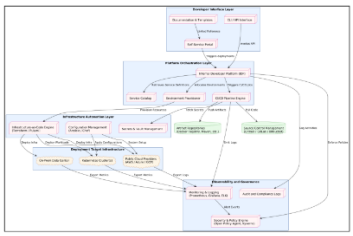


Рисунок 2.5 - Комплексна архітектура внутрішніх платформ розробників (IDP) у платформній інженерії

Ці характеристики будуть особливо корисними в багатокористувацьких та розподілених ситуаціях, де видимість та контроль набагато важливіші. Хмарно-орієнтований підхід також користується перевагами контейнерного підходу, що пропонується середовищами виконання, сумісними зі стандартами Docker та OCI, оскільки вони спрощують упаковку та розгортання програм. IDP зазвичай забезпечують чіткі конвеєри, які дозволяють розробляти, доставляти та випускати контейнери для кластерних систем Kubernetes. Маніфестами Kubernetes та робочими процесами GitOps можна керувати за допомогою таких інструментів, як Helm та Kustomize. IDP інтегровані з нативними сервісами хмарних постачальників (наприклад, AWS, Azure, GCP) для використання в сховищі, мережах, управлінні ідентифікацією та моніторингу. Це дозволяє командам платформ надавати гібридні функції, поєднуючи функції Kubernetes та нативної хмари,

приховуючи складність від розробників. Поєднання хмарно-орієнтованої екосистеми та IDP дозволяє розробляти високостійкі, масштабовані, портативні, перспективні та зручні для розробників платформи.

2.3 Основні функції, сервіси та можливості внутрішніх платформ розробника

Внутрішні платформи розробників (ВПР) призначені для автоматизації процесу життєвого циклу розробки програмного забезпечення, пропонуючи розробникам робочі процеси та інструменти, необхідні для ефективного створення, [35] тестування, розгортання та моніторингу програм. ВПР відрізняються від класичних інструментальних ланцюжків DevOps тим, що вони зосереджені на самообслуговуванні, абстракції, автоматизації та досвіді розробника. У цьому підрозділі автори описують три основні характеристики, які визначають виживання та вплив сучасних ВПР.

Портал самообслуговування, через який розробники можуть отримувати доступ до необхідних ресурсів та інструментів на вимогу та більше не зобов'язані вивчати інфраструктуру чи покладатися на ручне сприяння команд платформи, є однією з найефективніших характеристик IDP. Портали об'єднані як центральні панелі інструментів, за допомогою яких розробники можуть створювати нові сервіси на основі шаблонів, налаштовувати середовища, переглядати журнали, ініціювати розгортання та керувати секретами. Ці портали часто реалізуються за допомогою таких інструментів, як Backstage, Port або користувацьких веб-інтерфейсів, що забезпечує висококваліфікований досвід, специфічний для робочих процесів організації [36]. Портали самообслуговування скорочують час адаптації, зменшують когнітивне навантаження та дозволяють розробникам зосередитися на функціях, а не на вирішенні проблем інфраструктури, оскільки вони спрощують складні фактори, надаючи попередньо визначені шляхи (затверджені робочі процеси розробки) та приховуючи основну складність. Таке спрощення доступу підвищує продуктивність, а також пришвидшує темпи розробки в

організації.

Абстракція інфраструктури – це ще один стовп IDP, оскільки вона захищає розробників від труднощів налаштування та експлуатації інфраструктури. Розробники взаємодіють за допомогою уніфікованих високорівневих інтерфейсів, а не пишуть низькорівневий код Terraform, працюють з наземними хмарними консолями та забезпечують стандартизовані середовища у фоновому режимі [37]. Ця абстракція досягається завдяки використанню шаблонів багаторазового використання, API та модулів Infrastructure-as-Code (IaC), якими керують інженери платформи. Стандартизація гарантує, що сервіси, реалізовані за допомогою IDP, відповідатимуть найкращим практикам організації, політикам відповідності та вимогам безпеки всіх розгорнутих сервісів. Вона також сприяє ретельності в різних середовищах розробки, проміжного та виробничого середовища, що призводить до зменшення можливостей неправильної конфігурації та помилок, специфічних для середовища. Абстракція інфраструктури усуває невідповідності та ручні процеси, тим самим підвищуючи стабільність, мінімізуючи операційний ризик та покращуючи надійність розгортання.

Автоматизація CI/CD (безперервна інтеграція та безперервне розгортання) є ключовою особливістю IDP, оскільки вона забезпечує швидку, повторювану та послідовну доставку програмного забезпечення. Після того, як код передано до системи контролю версій, IDP запускає попередньо налаштований конвеєр, який автоматично збирає, тестує, упаковує та розгортає програму. Такі конвеєри зазвичай є стандартними та можуть бути розгорнуті кількома командами для забезпечення контролю якості, перевірок безпеки та політик розгортання. IDP підтримують популярні механізми конвеєрів, включаючи GitHub Actions, GitLab CI, ArgoCD та Tekton, для керування автоматизацією в усьому середовищі. Автоматизація не тільки прискорює час до випуску продукту, але й зменшує можливість людських помилок та необхідність дотримання вимог, одночасно сприяючи таким практикам, як розробка на основі транків та GitOps. Вирішуючи проблему масштабування, конвеєри CI/CD забезпечують підтримку планів відкату, синьо-зелених або канаркових розгортань та інтеграційних

тестів, слугуючи стабільною основою для масштабування доставки програмного забезпечення.

Повні можливості спостереження, моніторингу та ведення журналу інтегровані в зрілу внутрішню платформу розробників (IDP), що забезпечує розробникам та операторам негайну прозорість того, як працюють та працюють програми та інфраструктура [38]. У цьому випадку спостереження виходить за рамки базового моніторингу безвідмовної роботи, пропонуючи метрики, журнали та трасування, які надають уявлення про те, як функціонують сервіси, наявність вузьких місць та процес вирішення інцидентів у обмежені терміни.

Інструменти спостереження зазвичай вбудовані в платформу за замовчуванням, при цьому розподілені інструменти спостереження використовують рішення на базі Prometheus для збору метрик, рішення на базі Grafana для візуалізації даних та централізоване ведення журналу на базі ELK Stack (Elasticsearch, Logstash, Kibana) або Loki. Ці інструменти надають розробникам інформаційні панелі та сповіщення, адаптовані до конкретних сервісів або середовищ. Існують також розподілені інструменти трасування, такі як Jaeger або OpenTelemetry, які спрощують запити на трасування через мікросервіси та спрощують налагодження проблем у складних середовищах. Розподілений інструмент трасування оптимізує операційні процеси, централізуючи моніторинг та ведення журналу, а також пришвидшуючи виявлення першопричини. Розробникам не потрібно встановлювати та налаштовувати зовнішні журнали інфраструктури або отримувати доступ до цих інструментів, оскільки показники продуктивності можна передавати назад через інтерфейс з платформою. Ця здатність є важливою для захисту від цілей рівня обслуговування (SLO), підвищення надійності та підтримки реагування на інциденти та аналізу пост-мортем. Безпека та відповідність вимогам є важливими складовими будь-якої платформи внутрішнього розробника, і оскільки світ стає все більш багатохмарним та розподіленим, він також стає все більш жорстко регульованим. Внутрішні розробники використовують засоби безпеки та забезпечення дотримання політик на рівнях, які забезпечують інфраструктуру, розгортають інфраструктуру, впроваджують контроль доступу та

забезпечують дотримання політик виконання. Це досягається шляхом інтеграції фреймворку замість використання ручних або адаптивних режимів безпеки [39]. Контроль доступу на основі ролей (RBAC) є одним з основних механізмів, що використовуються в управлінні ідентифікацією та доступом (IDM), щоб гарантувати, що користувачам не дозволено отримувати доступ до ресурсів та виконувати дії, які їм не потрібні. Цей тип доступу зазвичай нав'язується шляхом підключення до постачальників ідентифікаційних даних (наприклад, SSO, LDAP, OAuth).

Крім того, управління секретами покращується за допомогою таких інструментів, як HashiCorp Vault або AWS Secrets Manager, які пропонують контроль зберігання та доступу, а також детальний доступ до конфіденційних даних (наприклад, ключів API та облікових даних). IDP також включають ведення журналу аудиту, механізм політики як коду, такий як Open Policy Agent (OPA) або Kyverno, та рішення для сканування вразливостей, таке як Trivy або Aqua Security, для досягнення стандартів відповідності. Ці інструменти виконують перевірки безпеки, створюють звіти про відповідність та гарантують, що будь-який небезпечний код або конфігурації ніколи не потраплять у робочий режим. IDP можуть дозволити організаціям змістити безпеку вліво, зменшуючи потенційні ризики та створюючи впевненість у своїх механізмах доставки програмного забезпечення, інтегруючи безпеку та відповідність у життєвий цикл розробки та розгортання.

2.4 Висновки по розділу

У другому розділі розглянуто методи та інструменти побудови внутрішніх платформ розробника. Досліджено референтну архітектуру IDP, її основні компоненти та структурні елементи, а також ключові сервіси та функціональні можливості. Проаналізовано підходи до організації платформ, що забезпечують стандартизацію, автоматизацію та прискорення процесів розробки. У результаті сформовано узагальнену модель побудови внутрішньої платформи розробника.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ВНУТРІШНІХ ПЛАТФОРМ РОЗРОБНИКА

3.1 Розробка та реалізація внутрішньої платформи розробника

Внутрішня платформа розробника (IDP) – це інструмент, розроблений для спрощення робочих процесів розробників завдяки можливостям самообслуговування [2] . Вона складається з набору різних інструментів для створення єдиної системи, з якою має бути легко взаємодіяти. Оскільки внутрішні інфраструктури відрізняються від організації до організації, реалізація IDP дуже різниться. З'являється багато інструментів для підтримки створення таких платформ.

У перших розділах представлено ідею IDP з різних точок зору та описано деякі переваги, які вона може забезпечити. У розділі 3.6 наведено деякі загальні пропозиції щодо впровадження, а в останньому розділі представлено деякі існуючі рішення, як з відкритим кодом, так і платні сервіси.

Внутрішня платформа розробника (IDP) – це набір різних інструментів та процесів, спрямованих на оптимізацію та спрощення життєвого циклу розробки програмного забезпечення. Вона розроблена для мінімізації складнощів, з якими розробникам доводиться стикатися під час використання ідеології DevOps, дозволяючи розробникам більше зосередитися на бізнес-логіці. IDP слугує способом роботи, призначеним для забезпечення самообслуговування розробників, спрощення та уніфікації робочих процесів, пов'язаних з операціями. IDP не слід розглядати як черговий монолітний інструмент , а радше як набір усіх існуючих інструментів, які команди розробників вже використовують [40]. Процес впровадження цих інструментів називається платформною інженерією [18] .

IDP прагне зменшити когнітивне навантаження команди розробників продукту. Замість того, щоб кожен розробник вивчав схеми керування Terraform або Kubernetes, вони можуть використовувати інструменти для виконання всіх операційних завдань самостійно. [16, 19] Інтерфейс для керування та обробки всього всередині IDP відомий як площина керування. Поширеними способами

реалізації площини керування є веб-застосунок, до якого користувачі мають доступ. Іншими поширеними інтерфейсами є інтерфейс командного рядка (CLI), API або навіть чат-бот. Основним критерієм вибору інтерфейсу є простота використання. [2 , 16]

IDP має забезпечувати прозору абстракцію над існуючою інфраструктурою. Команди розробників повинні мати можливість налаштовувати базову інфраструктуру за потреби. Крім того, впровадження IDP має бути добровільним та пропагуватися з метою спрощення завдань розробників. [19]

IDP підходить не для кожного випадку використання. Для невеликих організацій може бути краще використовувати PaaS (платформа як послуга) для розробки програмного забезпечення. Одним із відомих PaaS для розробки програмного забезпечення є Heroku [16]. Впровадження власного IDP може стати в пригоді, якщо PaaS більше не підходить для варіантів використання в організації. IDP є більш настроюваним та гнучким, але вимагає серйозних зусиль та спеціальної команди для впровадження [19] .

IDP слід розглядати як продукт, кінцевим користувачем якого є розробники. Таким чином, усі добре зарекомендували себе найкращі практики розробки продуктів можна застосувати до розробки платформи. Кінцеві користувачі повинні надавати інформацію про додаткові функції та покращення платформи [16 , 20]. Для ефективного впровадження необхідна тісна співпраця між командою платформи та розробниками.

Коли IDP розглядається як продукт, слід також враховувати його надійність. Це продукт, який використовують клієнти, і як такий, він має бути надійним, стабільним та зручним у використанні [19]. Платформа повинна мати достатню аналітику та моніторинг, щоб реагувати на простої та збирати аналітику використання для покращень.

Добре реалізована платформа зменшує когнітивне навантаження на впровадження додатків. Коли IDP має шаблони для різних типів продуктів, створення нового сервісу може бути лише одноразовим завданням, замість того, щоб витрачати дні на налаштування інфраструктури, моніторингу та аналітичних

фреймворків [18, 19]. Таким чином, розробники менше страждають від перемикання контексту, коли їм не потрібно зосереджуватися на деталях інфраструктури . Раптом написання Terraform або Kubernetes helm-чартів може бути складним завданням для деяких розробників.

IDP забезпечує самообслуговування. Це означає, що розробники можуть виконувати певні заздалегідь визначені дії самостійно, не покладаючись на інших співробітників. Це дозволяє операційному персоналу більше зосередитися на розробці інфраструктури, а не на складанні заявок для інших розробників. Завдяки заздалегідь визначеним діям практично обов'язково дотримуватися найкращих практик, встановлених на платформі. Ці функції роблять створення нових сервісів простішим та швидшим, що сприяє швидшому впровадженню інновацій [2 , 16 , 21].

Деякі IDP розвиваються на основі каталогу послуг. Це інструмент для документування різних послуг, які вже існують в організації. Він може надати основну інформацію про послугу, її відповідальну команду та залежності, які вона має для різних послуг. У деяких випадках він може навіть включати документацію API і служити основним місцем для документації.

Під час розробки необхідно прийняти багато рішень щодо реалізації програми. Як структурувати код, які інструменти використовувати та як використовувати їх найефективніше. Золоті шляхи – це термін, що пропонує рішення цієї проблеми [18]. Він представляє набір шаблонів, стандартів та добре відомих найкращих практик, яких слід дотримуватися. Золоті шляхи мають чітко визначену позицію та відображають організаційні рішення щодо інфраструктури та вподобання щодо інструментів і методологій [19].

IDP має підтримувати та сприяти дотриманню визначених золотих шляхів [16]. Автоматизація та дії повинні відповідати цим обраним найкращим практикам. Коли розробники дотримуються золотих шляхів, різні програми приймають однакові рішення, що спрощує зміну проектів. Це також спрощує впровадження автоматизації, коли проекти є однорідними. Таким чином, адміністратори безпеки можуть легко впроваджувати напрямні політик, і навіть складне

завдання оптимізації використання хмари може стати простіше. [18]

Повинні бути різні золоті шляхи для різних випадків використання та різних стеків застосунків. Загальний огляд робочого процесу розробки з IDP на рисунку 3.1

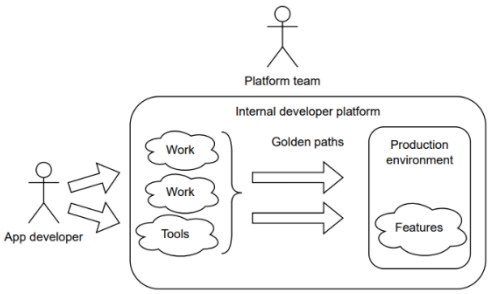


Рисунок 3.1 - Загальний огляд IDP

На рисунку вище команда платформи керує всім IDP. Розробник додатків дотримується заздалегідь визначених золотих шляхів для внесення змін у робоче середовище. В IDP вони також мають доступ до інших інструментів, таких як моніторинг або каталог послуг.

Одне з визначень IDP дається шляхом визначення основних компонентів, які він повинен мати. Основними компонентами IDP є:

- **Керування конфігурацією програм**: Керування налаштуваннями та конфігураціями програм включає обробку налаштувань, специфічних для програм та їхнього операційного середовища. Хоча керування кількома налаштуваннями вручну є можливим, складність зростає з динамічними конфігураціями, такими як прапорці функцій. Такі проблеми, як контроль версій та обробка секретних даних, також ще більше ускладнюють налаштування. Постачальник ідентифікованих даних спрощує це, пропонуючи централізовану систему для керування всіма потребами конфігурації, забезпечуючи гнучкість та безпеку.
- **Оркестрація інфраструктури**: В ідеалі, IDP має відповідати за налаштування та управління інфраструктурою, на якій працюють програми. Це включає все: від хмарних сервісів до фізичних серверів. Мета полягає в тому, щоб зробити управління цією інфраструктурою простим, дозволяючи

розробникам легко налаштовувати та коригувати свої операційні середовища, не уповільнюючи їх через технічні деталі.

- **Керування середовищем**: За допомогою IDP розробники можуть швидко створювати та керувати своїми середовищами для кожної служби. Такий підхід самообслуговування зменшує затримки, оскільки розробникам не потрібно чекати, поки хтось інший налаштує їхні середовища за них.
- **Керування розгортанням**: IDP має покращити безперервне розгортання, пропонуючи автоматизацію, легке відкати та надійну підтримку усунення несправностей. Ці функції забезпечують плавніші, надійніші оновлення з меншим ризиком та кращий досвід порівняно з базовими інструментами розгортання. IDP допомагають керувати розгортанням, оптимізуючи процеси та розширюючи можливості команди, роблячи безперервне розгортання ефективнішим та безпечнішим.
- **Контроль доступу на основі ролей (RBAC)**: IDP використовують RBAC для визначення того, що кожен користувач може, а що не може робити на платформі, залежно від своєї ролі. Це гарантує, що члени команди мають доступ, необхідний для продуктивної роботи, зберігаючи при цьому безпеку та контроль. Наприклад, власники сервісу можуть мати розширений доступ порівняно з іншими, яким потрібен лише перегляд базової інформації про сервіс.

Хоча впровадження платформи з усіма цими компонентами вже є дуже повним IDP, є й інші аспекти, які доповнюють досвід розробників у робочих процесах. Наприклад, функція каталогу послуг може бути простою в налаштуванні та може принести велику цінність користувачам, підвищуючи видимість та знання. [2]

3.2 Впровадження та інтеграція платформи у процес розробки

Впровадження IDP – непросте завдання, оскільки потрібно враховувати багато різних варіантів використання та користувачів. Технічні вимоги до впровадження можуть сильно відрізнятися, але загалом вони є обширними.

Керування багатьма різними технологіями необхідне для об'єднання різних систем та сервісів.

Під час розробки IDP необхідно враховувати багато різних речей, щоб зробити його зручним для користувача. Перш за все, платформа має бути простою у використанні. Це включає чіткі інструкції, прості інтерфейси та дизайн, який відповідає завданням, для яких розробники використовують платформу. Це має допомагати їм виконувати свою роботу, не спричиняючи плутанини та не вимагаючи багато додаткового навчання. Іншим фактором, який слід враховувати, є призначення платформи. Вона має бути створена для завдання, для якого вона призначена. Вона повинна пропонувати інструменти та функції, які розробникам фактично потрібні для завершення їхніх проектів. Це може означати що завгодно: від спрощення ініціалізації нових проектів до інтеграції з іншими сервісами в контекст [2 , 19].

Ще одним фактором є надійність і стабільність платформи. Розробники повинні мати можливість розраховувати на те, що платформа буде доступною та працюватиме правильно, коли їм це потрібно. Якщо платформа часто не працює або працює неправильно, це може уповільнити їхню роботу та стати джерелом розчарування. Коли технології змінюються, платформа повинна мати можливість адаптуватися, не ускладнюючи життя своїм користувачам. Це означає, що вона повинна мати можливість обробляти більше користувачів або нові види проектів у міру зростання та змін організації [19].

Впровадження IDP слід починати з малого. Зазвичай починають з автоматизації робочого процесу створення проєкту, оскільки зазвичай для нього існує чіткий порядок дій, якого необхідно дотримуватися. Виявлення та автоматизація найпоширенішої частини робочого процесу розробника може забезпечити найкращу цінність для кінцевих користувачів [2] . Ці автоматизації та допоміжні функції повинні бути добре реалізовані та підтримувати певну якість, щоб забезпечити розробникам найкращий досвід.

Поширеною тенденцією є наявність повноцінної команди, яка займається розробкою та підтримкою платформи. Ця команда приймає рішення щодо всієї

платформи у співпраці з командами розробників, експертами постачальників хмарних послуг та операційними командами для створення інструментів, що використовуються в робочих процесах розробки [17] . В ідеалі команда повинна розглядати себе як постачальника послуг та зосереджуватися виключно на розробці платформи. У великих організаціях може бути кілька команд, які працюють над різними частинами платформи.

Впровадження нових технологій, як правило, відбувається повільно. Впровадження IDP також може бути повільним і стикатися з певним опором. [19] Ці проблеми можна пом'якшити, встановивши швидкий цикл зворотного зв'язку з користувачами. Швидкі цикли зворотного зв'язку з користувачами платформи можуть допомогти визначити, чи є представлені технології корисними чи ні. Зворотній зв'язок також має вирішальне значення для загальної розробки платформи, щоб зрозуміти варіанти використання та больові точки [16]. З усією платформою та новими функціями команда платформи повинна також використовувати власні інструменти.

Багато вимірних переваг IDP стосуються дій, які він може виконувати, та способів, якими він може принести користь управлінню послугами, і тому переваги є більш помітними, коли IDP використовується з мікросервісною архітектурою. Натомість, його переваги можуть бути не повністю реалізовані в монолітному застосунку, оскільки кількість дій в IDP буде менш поширеною.

Якщо організація вже встановила деякі практики IaC та використовує контейнеризоване розгортання або навіть Kubernetes, впровадження IDP буде простішим. Багато існуючих продуктів для IDP надають перевагу Kubernetes як платформі запуску, оскільки вона вже забезпечує рівень абстракції. [2 , 17]

Впровадження IDP значно варіюється від організації до організації, оскільки операційні контексти суттєво відрізняються. Таким чином, найкращі практики впровадження також дуже різняться. У наступних підрозділах представлено деякі відомі рішення.

Існує багато різних інструментів для створення IPD, наприклад, Massdriver, Cor tex, Port, Compass та Crossplane. У цьому розділі представлено

два найпоширеніші інструменти, що використовуються, – продукти Backstage та Humanitech. Інструменти, розроблені для інших суміжних технологій, також відіграють ключову роль під час впровадження. [2]

Оптимальні інструменти значною мірою залежать від існуючої інфраструктури та інструментів, які вже використовуються. Перш ніж вибирати найкращі інструменти та технології для впровадження IDP, слід провести глибоке дослідження.

Backstage - це фреймворк з відкритим кодом для створення порталів для розробників [22]. Сам по собі він не є повноцінним IDP, але його можна розширити за допомогою різноманітних плагінів для підтримки широкого спектру робочих процесів розробки. Він також часто використовується як площина керування для IDP [2]. Backstage спочатку був розроблений як внутрішній інструмент для підвищення видимості в Spotify, але пізніше був відкритим та переданий Фонду Cloud Native Computing (CNCF) [23].

Backstage включає деякі плагіни з базовою інсталяцією, такі як каталог програмного забезпечення, шаблони програмного забезпечення та плагін технічної документації. Також доступний широкий вибір плагінів для додавання інтеграцій до різних систем. Spotify також продає преміум-плагіни, такі як Soundcheck, інструмент, створений для забезпечення якості послуг та відповідності інструкціям [22].

На рисунку 3.2 показано скріншот демонстраційного середовища Backstage. Ліворуч можна побачити різні плагіни, такі як каталог, документація та технічний радар. Кожен плагін має власну сторінку, яку можна налаштувати. Основним вмістом на скріншоті є запис каталогу послуг для сервісу під назвою "www-artist". У каталозі представлена різноманітна інформація про сервіс, наприклад, власник, життєвий цикл, короткий опис, а також інтерактивна діаграма зв'язків. На діаграмі показано залежності та залежні компоненти. З цієї сторінки ми можемо швидко визначити, з ким зв'язатися та з якою критичністю, якщо у сервісу виникнуть серйозні проблеми. Під заголовком праворуч знаходиться

панель навігації для різних плагінів, пов'язаних із сервісами. Це забезпечує прямий доступ до сторінок плагінів, специфічних для сервісу. [24]

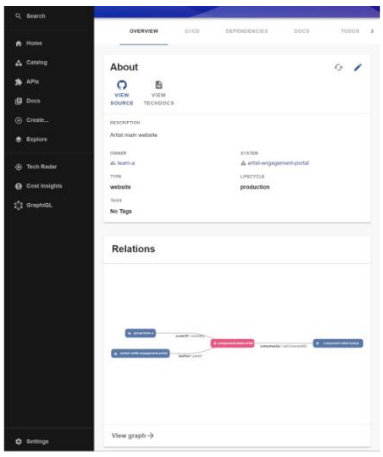


Рисунок 3.2 - Знімок екрана сторінки каталогу послуг у демонстраційному середовищі Backstage

Плагін каталогу використовує конфігурацію оголошення сутностей, описану в програмі, як джерело даних. Для найпростішої реалізації для кожної служби потрібно додати до системи такий файл. Поля можуть значно відрізнятися залежно від встановлених плагінів та конфігурації. У деяких інтеграціях частина цих полів може бути автоматично заповнена системою керування кодом.

```
1  apiVersion: backstage.io/v1alpha1
2  kind: Component
3  metadata:
4      name: www-artist
5      description: Artist main website
6  spec:
7      type: website
8      lifecycle: production
9      owner: team-a
10     system: artist-engagement-portal
11     consumesApis: ['component:artist-lookup']
```

Лістинг 3.1 - Приклад оголошення сервісу. Це той самий сервіс, що представлений на рисунку 3.2.

Проект, отриманий у попередньому розділі, цілком природно реалізується за допомогою мікросервісної архітектури. Власне, CAS базується на такому природному виборі. Більш конкретно, на рисунках На рисунках наведено представлення архітектури CAS, розділеної на сервер, який пропонує, як

мікросервіси, інструменти, представлені в розділі II-D , та клієнт, який їх використовує. Далі ми опишемо їх обидва.

Тут ми ілюструємо багаторівневу архітектуру сервера та зв'язок з клієнтом. Сервер структурований на три рівні, які ми зараз опишемо.

- Нижній рівень (операційна система). CAS може використовувати будь-який дистрибутив Linux на базі Debian. Для нашої реалізації ми використовуємо Ubuntu та Debian.
- Середній рівень. Платформа контейнеризації (наприклад, Docker) абстрагується від базової архітектури та дозволяє легко розгортати мікросервіси, що характеризують CAS, у різних середовищах. Ці мікросервіси та їхні необхідні залежності (наприклад, бази даних та веб-сервери) знаходяться у власному незалежному контейнері та розташовані в одній приватній мережі Docker, де лише вибрані точки входу дозволяють зовнішню взаємодію.
- Верхній рівень. Цей рівень складається з сервісів розробки програмного забезпечення, що відповідають інструментам, описаним у розділі II-D . Для кожного з них ми також перераховуємо необхідні сервери додатків та механізми баз даних.

Як зазначено, програмні компоненти повинні обмінюватися інформацією між собою. З іншого боку, має бути зв'язок між сервером, який пропонує мікросервіси, та різними клієнтами. У CAS таке завдання реалізується через веб-сервер Nginx. Це природний вибір, оскільки, як зазначено в розділі II-D , багато обраних програмних інструментів вже налаштовані на зв'язок через Nginx. Цей вибір також пропонує додаткові переваги з точки зору зв'язку між сервером і клієнтом, захищаючи їх.

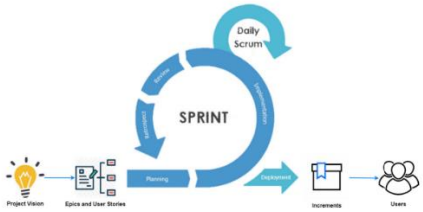


Рисунок 3.3 - Цикл розробки за методологією Scrum

Зліва направо: визначається продукт, що розробляється, а також встановлюються та набуваються необхідні для його розробки компетенції; потім продукт розробляється в ході низки ітерацій спринтів, а його версії випускаються для оцінки або остаточного впровадження.

Дійсно, Nginx - це легкий веб-сервер з відкритим кодом, який також можна використовувати як зворотний проксі, кеш HTTP та балансувальник навантаження. Він використовує порти 80 (для HTTP) та 443 (для HTTPS) через криптографічний протокол презентації TLS. Варто зазначити, що CAS може використовувати HTTPS завдяки інструменту Linux Certbot, який надає (безкоштовно) сертифікат "Let's Encrypt". Такий сертифікат потім періодично оновлюється за допомогою інструменту Linux *crontab* (див., наприклад, документацію сторінок Debian). Крім того, Nginx має приємну особливість пропонувати низьке споживання пам'яті та високий паралельний режим, оскільки він не створює новий процес для кожного веб-сайту, а використовує асинхронний подієвий підхід із запитами, що обробляються одним потоком.

Як стисло показано на рисунку 3.3 , з точки зору користувача, для доступу до платформи CAS, доступ до якої можна отримати з локальної мережі або через Інтернет, потрібен лише веб-браузер. Таким чином, користувач може налаштувати своє робоче середовище, яке можна інтегрувати з плагіном логера для відстеження продуктивності. Наразі ця функція доступна для Visual Studio Code та IntelliJ IDEA [32] .

Рисунок 3.4 - Програмні засоби для етапу планування.

Перелік засобів, які можна використовувати на цьому етапі спринту, відповідно до опису, наведеного в основному тексті.

Тепер ми обговоримо, як налаштувати CAS, як користувачі можуть отримати до нього доступ та як його можна змінити з точки зору додавання/видалення пропонованих мікросервісів, тобто програмних інструментів, які вважаються необхідними для гнучкої розробки програмного забезпечення.

Початкове налаштування платформи CAS виконується за допомогою Ansible, який в основному відповідає за налаштування та розгортання контейнерів Docker кожного мікросервісу. У найпоширенішому підході Ansible виконується на зовнішній машині, орієнтованій на сервер, де буде розміщена платформа. Перед запуском скрипта Ansible інсталятор повинен спочатку заповнити параметри, специфічні для екземпляра, у файлі конфігурації. Після запуску робочий процес Ansible (який називається *playbook*) виконує такі кроки на цільовому сервері:

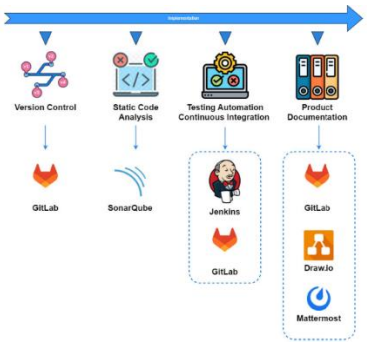


Рисунок 3.5 - Програмні засоби для етапу впровадження.

Цей етап було розбито на різні завдання відповідно до основного тексту. Для кожного з них легенда така, як на рисунку 3.4 .

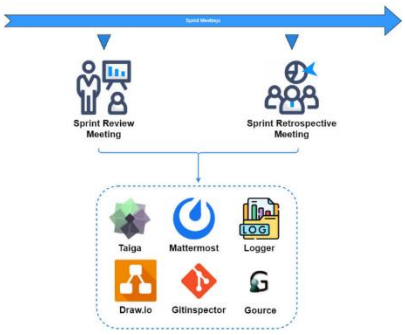


Рисунок 3.6 - Програмні засоби для проведення спринтів.

Пояснення до рисунка такі самі, як на рисунку 3.5.

1) Встановіть необхідні залежності для всієї системи.

2) Створіть приватну мережу Docker, де будуть розміщені всі контейнери.

3) Створіть та ініціалізуйте файли конфігурації Docker для кожного мікросервісу. Потім запустіть контейнери.

4) Згенеруйте конфігурацію маршрутизації для кожного мікросервісу - та запустіть контейнер Nginx . Якщо це потрібно в початковій конфігурації, на цьому етапі також запитується сертифікат для HTTPS.

Після завершення скрипта Ansible, інсталятор має виконати перший доступ у кожному мікросервісі, щоб створити користувача з правами адміністратора. Ця процедура є специфічною для кожної програми, і для отримання додаткової інформації ми відсилаємо читача до [32] .

Блоки позначають програмні засоби, а пунктирні лінії - напрямок потоку інформації між ними, відповідно до опису в основному тексті.

Як додатковий крок, після завершення початкової інсталяції, можна виконати другий посібник Ansible для налаштування єдиного входу (SSO) GitLab, який ми опишемо в наступному розділі. Цей робочий процес відповідає за створення токенів доступу GitLab та їх встановлення на інших сумісних мікросервісах.

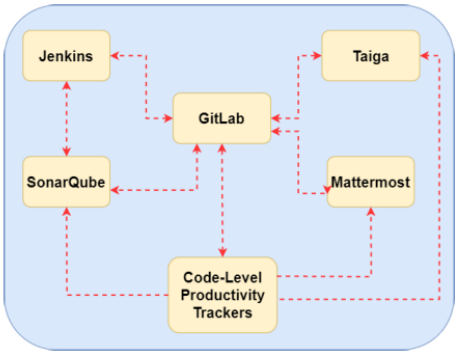


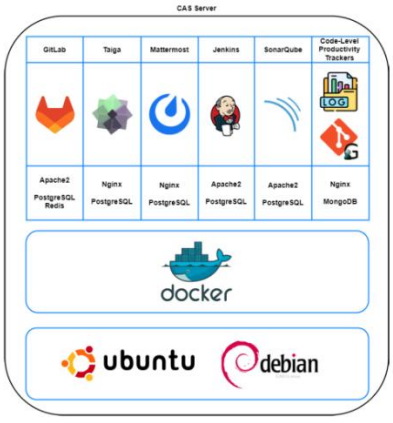
Рисунок 3.7 - Типовий сценарій взаємодії між програмними компонентами.

Оскільки GitLab відіграє центральну роль у CAS (див. рис. 3.7), природно ввімкнути автентифікацію користувачів GitLab SSO. Впровадження цього рішення для автентифікації користувачів, як загальної, так і спеціалізованої (розробницької), відображає зростаюче визнання важливості спрощеного та безпечного доступу до сервісів відповідно до принципів управління ідентифікацією та доступом (IAM) у сучасних організаціях. Прагматично кожен автентифікований користувач може отримати доступ до будь-якого з сервісів, що надаються CAS, контрольованим способом. Впровадження GitLab SSO пропонує додаткові переваги, такі як: (1) підтримка різних стандартів автентифікації та можливість інтеграції з існуючими сервісами автентифікації; (2) відповідність нормативним вимогам; (3) автоматизація створення облікових записів користувачів.

Три рівні сервера CAS. Кожен із них описано в основному тексті.

Перевагою CAS є її модульність, яка дозволяє легко додавати або видаляти сервіси залежно від потреб команди.

Ми припускаємо, що базова версія CAS увімкнена. Зміни в цій версії можуть бути як тимчасовими, так і постійними.



..

Рисунок 3.8 - Композиційна гнучка система (CAS) - загальний огляд сервера.

Видалення існуючої служби. Видалення однієї з служб, доступних у CAS, можна здійснити наступним чином.

a) Для даного сервісу відповідний контейнер має бути зупинений, використовуючи термінологію Docker. Таку операцію може виконати будь-який професійний член Scrum-команди.

b) Якщо таке видалення має бути постійним, тобто зміна забезпечує нову платформу, тоді також необхідно внести зміни до файлу конфігурації Ansible.

Вищезазначені кроки детально описані в [32], а в Додатку Наведемо приклад: видалення сервісу «Тайга».

Додавання нової послуги. Додавання послуги можна здійснити наступним чином, де для стислості ми не згадуємо налаштування параметрів, характерних для цієї послуги.

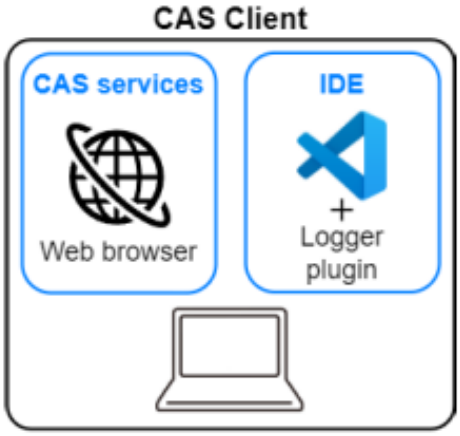


Рисунок 3.9 - Композиційна гнучка система (CAS) - загальний огляд з боку клієнта.

,

Компоненти клієнта CAS, як описано в основному тексті.

a) Спочатку потрібно перевірити, чи сумісний сервіс із технологією Docker. У такому випадку член команди, який бажає додати новий сервіс, повинен створити відповідний контейнер Docker. Сервіси, несумісні з Docker, а їх дуже мало, не можуть бути частиною CAS.

b) Новий контейнер має бути інтегрований у віртуальну приватну мережу Docker.

c) Увімкніть веб-доступ до нового сервісу через Nginx.

d) Запустіть контейнер Docker, створений на кроці a).

e) Якщо сервіс не дозволяє автентифікацію через GitLab SSO, під час кожного робочого сеансу користувач CAS повинен вводити облікові дані для входу, специфічні для цього сервісу, на додаток до тих, що вже надані для GitLab SSO.

Як уже зазначалося у вступі, CAS є *унікальним прикладом* з точки зору внутрішньопрофільних розробників (ВПР), що становлять інтерес для цього дослідження. Більше того, наскільки нам відомо, наукова література не пропонує методологій для оцінки корисності ВПР, за винятком тематичного дослідження, пов'язаного з [6] та [7]. Таким чином, враховуючи послідовну та широко прийняту процедуру, якої слід дотримуватися досі, ми спираємося на попереднє добре задокументоване та успішне використання попередньої версії CAS у стратегічному проекті. Тут ми надаємо додаткові докази цінності поточної версії CAS з точки зору самостійного розміщення на різних апаратних сценаріях, окреслюючи також варіанти використання для кожного сценарію. Нарешті, два додаткові ілюстративні приклади, знову ж таки пов'язані з її використанням у державному секторі, представлені досить детально.

CAS можна налаштувати в кількох різних системних архітектурах. Тут ми опишемо три різні конфігурації: віртуальний сервер, фізичний сервер на базі "голого металу" та конфігурацію на обмеженому обладнанні, такому як персональний комп'ютер. Кожна реалізація має свої специфічні переваги та варіанти використання, які важко порівняти, що дозволяє адаптувати CAS до різних середовищ залежно від конкретних потреб та обмежень проекту.

Для цієї конфігурації можливий варіант використання такий. Команда розробників використовує віртуальний сервер для створення приватного середовища для тестування, яке відображає виробниче середовище програмного сервісу [30] .

Реалізація віртуального сервера використовує деякі гіпервізори для створення програмного середовища, яке імітує фізичне обладнання, наприклад, у приватній хмарі. Цей підхід є гнучким, що дозволяє легко переналаштовувати

ресурси, такі як процесор, пам'ять та сховище. Ця реалізація потребує відповідної операційної системи та певної технології віртуалізації. Її перевагами є масштабованість на вимогу; ізоляція, оскільки кожен віртуальний сервер працює окремо, забезпечуючи безпеку та стабільність; ремонтопридатність, оскільки діагностика та вирішення проблем відбуваються швидше; та ефективність використання доступних ресурсів. Крім того, таке рішення спрощує аварійне відновлення, оскільки віртуалізоване середовище дозволяє автоматизувати резервне копіювання, а весь сервер, включаючи його дані та конфігурації, можна швидко реплікувати на інший хост.

Для цієї схеми можливий варіант використання такий. Команда повинна розробити продукт або послугу, зберігаючи конфіденційність усіх розроблених даних та коду, для збереження прав інтелектуальної власності або з будь-якої критичної причини. Попередня версія CAS була успішно використана в цій схемі для розробки програмного забезпечення для критичного проекту для італійської державної адміністрації [27] .

Сервер без вбудованого обладнання (Black Metal Server) – це фізична машина, призначена для одного орендаря. Такий підхід забезпечує максимальну продуктивність та повний контроль над апаратним та програмним середовищем. У цьому випадку необхідно вибрати відповідне обладнання на основі вимог до робочого навантаження (процесор, пам'ять, сховище). Переваги цього підходу полягають у продуктивності, оскільки система CAS отримує прямий доступ до апаратних ресурсів; конфіденційності, оскільки команда має повний контроль над конфігураціями апаратного та програмного забезпечення; та надійності, оскільки немає залежностей або спільних ресурсів.

Для цієї конфігурації можливий варіант використання такий. Деякі студенти використовують як допоміжний сервер старий персональний комп'ютер, на якому встановлено CAS з мінімальними ресурсами. Зазначимо, що це найпоширеніший сценарій, який використовується командами студентів-програмістів для розробки проектів.

Впровадження CAS на персональному комп'ютері передбачає використання стандартного настільного комп'ютера або ноутбука. Такий підхід ідеально підходить для невеликих команд або для цілей розробки/тестування. Перевагами є мінімізація витрат, оскільки ми використовуємо вже доступне обладнання; портативність, оскільки CAS легко налаштувати та використовувати на різних машинах з обмеженими ресурсами; гнучкість, оскільки ця конфігурація проста та ідеально підходить для невеликих команд або навіть одного розробника, який працює як розробник і тестувальник.

Як згадувалося у вступі, платформа CAS використовувалася для підтримки проектної роботи на бакалаврських курсах з програмної інженерії.

Один із проектів передбачав розробку клієнта Twitter від імені італійської державної адміністрації у сфері цивільного захисту. Метою проекту було надати студентам практичний досвід у створенні реального застосунку в державному секторі, одночасно заохочуючи креативність у розробці функцій та інтерпретації даних. Ми також наголосили на інтеграції фундаментальних гнучких принципів, таких як композиційність та адаптивність, у розробку програмного забезпечення для забезпечення модульних та розширюваних рішень.

Розробка клієнта Twitter керувалася екстремальними принципами *розробки* [31] , що означало, що команди повинні використовувати лише інструменти та бібліотеки з відкритим кодом та уникати використання комерційних хмарних середовищ для зберігання свого коду та даних. До складу команд входили 5-6 студентів, які розбили продукт на окремі, компоновані модулі, кожен з яких мав певну функціональність. До них належали: модуль збору даних для захоплення геолокалізованих твітів через Twitter API; модуль картографування для відображення твітів на інтерактивній карті; модуль аналітики для створення хмар слів, часових діаграм та результатів аналізу настроїв.

Дотримуючись принципів Scrum, команди запровадили ітеративний процес розробки. Вони почали створювати клієнт як мінімально життєздатний продукт (MVP), зосереджуючись на ключових функціях користувача, таких як отримання та відображення твітів. Ми заохочували команди бути готовими

адаптувати свої продукти до змінних вимог або нових випадків використання. Наприклад, деякі команди впровадили модульні API, що полегшило заміну Twitter API альтернативними джерелами даних (наприклад, Mastodon або інші соціальні платформи) у відповідь на обмеження Twitter API.

Керування конфігурацією CAS сприяло безперешкодній інтеграції окремих внесків в єдину систему. Для керування кодовими базами та ефективного вирішення конфліктів інтеграції використовувалися спільні репозиторії та інструменти контролю версій (наприклад, Git).

Композиційна природа CAS дозволила командам незалежно тестувати модулі, забезпечуючи їхню надійність перед інтеграцією. Модульні тести були написані для окремих компонентів, а тести на системному рівні перевіряли загальну функціональність продуктів. Деякі команди також використовували синтетичні набори даних для перевірки аналітичних функцій за різних сценаріїв, таких як симуляції катастроф або зміни аналізу настроїв.

Після завершення своїх проектів усіх студентів індивідуально попросили оцінити, серед інших аспектів, інструменти, що надає CAS. На основі цих, а також додаткових описів, ми надаємо тут короткий виклад оцінки студентів щодо програмних інструментів, що входять до CAS. З цією метою сприйнята корисність кожного інструменту вимірювалася як за стандартною п'ятибальною шкалою, так і за допомогою коментарів у вільній формі. Результати наведено далі в порядку спадання балу Лікерта, зазначаючи, що інструмент Draw.io був включений до CAS наприкінці 2023 року, і тому він не включений до цього оцінювання. Населення становить приблизно 600 студентів.

- GitLab та Taiga отримали позитивну оцінку 4,6/5 та 3,9/5 відповідно. З коментарів у вільній формі видно, що студенти оцінили загальну корисність цих двох інструментів, обидва спеціалізовані на Agile-розробках.
- SonarQube, Gitinspector та Gource отримали оцінки вище середнього – близько 3,3/5. З коментарів у вільній формі видно, що налаштування цих інструментів є складним, а статистика, яку вони надають, має бути більш цілеспрямованою та корисною.

- Mattermost отримав середній бал 2,8/5. Цікаво, що у вільних коментарях студентів перевага надавалася їхнім власним інструментам обміну миттєвими повідомленнями, не усвідомлюючи необхідності єдиного вибору, який поділяють члени команди.
- Jenkins отримав оцінку 2,33/5, що трохи нижче середнього. Судячи з коментарів студентів у вільній формі, корисність сервісу обмежена, оскільки багато сервісів, які він надає, перетинаються з тими, що пропонуються GitLab.
- Logger отримав оцінку нижче середньої – 2,11/5. Згідно з коментарями студентів у вільній формі, основна критика стосується його можливостей контролю, які в класі можуть сприйматися як порушення конфіденційності.

Платформу CAS також можна використовувати для перевірки якості програмного продукту. Дійсно, громадяни можуть мати доступ до безлічі послуг через цей додаток. Для повноти картини нагадаємо, що це власний мобільний додаток для iOS та Android, розроблений переважно на Typescript компанією PagoPA SpA та деякими волонтерами, які підтримують проект, починаючи з лютого 2017 року. У листопаді 2023 року він складався з понад 195 тисяч рядків коду та понад 5 тисяч операцій commit, які були виконані на 75 різних гілках коду.

Ми зосередилися на деяких аспектах контролю якості цього додатка за допомогою CAS, тобто на статичному аналізі якості коду та метриках продуктивності команди, як обговорюється в наступних двох підрозділах.

Зокрема, коли CAS було застосовано до AppIO, ми розглянули один знімок репозиторію проекту , датований листопадом 2023 року. Додаткові знімки були б більш інформативними, але їх важко отримати, оскільки ми не брали участі в розробці проекту. Тим не менш, інформація, яку ми повідомляємо далі, з одного боку, демонструє можливості контролю якості CAS, а з іншого, дає хорошу оцінку якості проекту. Для повноти картини зазначимо, що щодо функцій, пов'язаних з використанням CAS Logger, їх неможливо було задокументувати в AppIO, оскільки ці функції можна використовувати під час розробки проекту, а не мати єдиного знімка вихідного коду.

Ми представили проектування та реалізацію IDP, яка відповідає вимогам: (1) складається з програмного забезпечення з відкритим вихідним кодом; (2) самостійно розміщується, тобто придатна для налаштування в інтрамережевій інфраструктурі організації, яка її використовує; (3) орієнтована на підтримку методології Agile, зокрема Scrum. Ми також описали та дослідили деякі сценарії застосування, де така IDP може бути корисною, разом із робочою IDP, що втілює запропоновану методологію, а саме CAS, корисність якої була продемонстрована в різних контекстах.

Наша робота відкриває шлях до багатьох напрямків досліджень, де позитивні зрушення ще більше збагатять CAS.

- Штучний інтелект. Оскільки розробники використовують можливості, що пропонуються штучним інтелектом (ШІ), розробники штучного інтелекту (ВШІ) повинні розвиватися, щоб інтегрувати технології ШІ [98]. Досягнення включатимуть покращення автоматизації, покращення прийняття рішень та постійну адаптацію в рамках гнучких робочих процесів. Наприклад, інструменти на основі ШІ для аналізу якості коду, прогнозного управління проектами та інтелектуальної автоматизації в конвеєрах DevOps стають дедалі важливішими. Дизайн та методологія майбутніх ВШІ повинні враховувати ці інновації, гарантуючи, що команди можуть використовувати ШІ для ефективного підвищення своєї гнучкості та ефективності.
- Масштабованість. У нашому контексті цей термін може бути неоднозначним. По-перше, коли він стосується команди розробників зі зростаючою кількістю залучених людей, CAS надає користувачам масштабованість, наскільки дозволяють апаратні ресурси. З іншого боку, *масштабований Agile* стосується кількості команд та людей, залучених до проекту, і це активна область досліджень. IDP, розроблені для підтримки цих версій Agile-методологій, недоступні, і пов'язане з цим питання про те, що мається на увазі під масштабованістю IDP у цьому контексті та як її досягти, залишається відкритим. Нарешті, коли масштабованість стосується інструментів, те, як вони працюватимуть у різних масштабах, структурах команд та складностях, майже не розглядається в літературі. Коли

така інформація була доступна, ми надавали таку інформацію, а для тих, що не охоплювалися, наприклад, Gource, ми зазначаємо, що це цікава область досліджень.

- Оцінка успіху IDP Ймовірно, через свою новизну, IDP не мають гнучких моделей зрілості, подібних до тих, що встановлені, наприклад, в організаціях. Їх визначення, впровадження та подальша валідація є ще однією складною відкритою проблемою, поставленою нашим дослідженням.
- Увага до державного сектору. Хоча цифрова трансформація державного сектору є глобальним викликом, CAS здається досить ізольованою пропозицією в цій галузі. Оскільки державні установи все більше переходять на використання Scrum для своєї цифрової трансформації, наприклад, Барселона зі Scrum@IMI та його більш загальною версією Scrum@PA, розробка та реалізація IDP, що підтримують ці методології, є важливою. Зокрема, оскільки залучення користувачів до управління проектами, наприклад, громадян, є центральною частиною такої цифрової трансформації, досягнення якої може вимагати додавання нових професійних фахівців до Scrum, важливо включити до IDP інструменти для підтримки цих нових фахівців.

3.3 Аналіз ефективності використання IDP та оцінка результатів впровадження

Цей підрозділ має на меті дослідити відповідність та потенційні переваги Внутрішньої платформи розробників (IDP) в рамках однієї конкретної організації. Спостереження та висновки, представлені в цьому розділі, отримані на основі безпосереднього досвіду двох років роботи в різних відділах організації, збагаченого розмовами з п'ятьма додатковими співробітниками організації. Хоча ці спостереження походять з конкретних випадків, вони, здається, відображають ширші теми та проблеми, що загалом піднімаються в організації. Цей аналіз має на меті запропонувати загальне розуміння проблем середовища розробки та операційних практик.

Це тематичне дослідження зосереджено на великій організації в роздрібній торгівлі. Вона має власний технічний відділ, що складається приблизно з 500 співробітників. Технічний відділ має кілька підвідділів, що зосереджені на різних сферах, необхідних для роздрібної торгівлі. На організаційному рівні існує центральна команда інфраструктури, яка підтримує інші DevOps-ролі . Ця команда також керує всіма зовнішніми сервісами, що використовуються. Як і інші організації значного розміру, ця також страждає від унікальних проблем, спричинених розміром, які впливають на її операційну ефективність та динаміку команди, особливо в технічному відділі.

Технічний відділ має кілька різних підвідділів, які беруть участь у широкому спектрі різних операцій компанії. Кожен підвідділ має свої власні контекстуальні вимоги та залежності, які може бути важко врахувати під час обговорення організаційних питань у контексті всієї компанії.

Технічний ландшафт в організації є водночас багатим і складним. Використовується широкий спектр технологій. Існують деякі рекомендації щодо рекомендованих технологій, що використовуються як у фронтенді, так і в бекенді, але все ще використовується широкий спектр інших технологій. Архітектури використовують мікросервісну архітектуру.

В організації можна виділити такі проблеми:

• **Відсутність прозорості** : Постійною проблемою є відсутність прозорості між різними відділами та командами. Така ізольована природа операцій зменшує міжфункціональну співпрацю та обмін знаннями, що призводить до неефективності та дублювання зусиль у різних проектах. Відсутність прозорості можна спостерігати в багатьох різних аспектах, таких як власність на послуги, залежності від послуг та залежні від них, а також стан життєвого циклу послуг.

• **Проблеми адаптації**: Адаптація – це складне завдання в будь-якій організації, і саме такою є ця. Оскільки різні команди мають свої власні методи, новачкам та тим, хто змінює команди, необхідно ознайомитися з новими способами роботи. Більшість команд мають внутрішню документацію з адаптації на платформі внутрішніх документів компанії, але вона швидко застаріває. Крім

того, компанія використовує зовнішню робочу силу в технічному відділі, що призводить до швидшої ротації технічного персоналу.

• **Відсутність чітких пропагованих методів роботи**: Різні команди розробників продуктів в організації досить розділені та мають власні бажані рішення з багатьох різних тем, починаючи від використовуваних технологій і закінчуючи обраною структурою папок та використовуваними зовнішніми інструментами. Хоча це забезпечує більше свободи, це призводить до відмінностей між проектами та командами. Крім того, під час створення нових сервісів немає шаблонів для використання. Поширеним способом є копіювання існуючого сервісу як шаблону. Це може призвести до небажаного просування неідеальних методологій та технологій.

• **Відсутність стандартів представлення API**: Хоча багато команд мають певний спосіб подання документації API, він все ще сильно відрізняється від команди до команди. Деякі використовують документацію OpenAPI, а деякі команди публікують свою документацію API на внутрішній платформі документації. Документація API, що оновлюється вручну, має погану звичку швидко застарівати.

• **Керування дозволами для сервісів**: Зовнішні інструменти здебільшого використовують єдиний вхід (SSO) для автентифікації. Це означає, що для отримання доступу до зовнішніх сервісів потрібен сервісний квиток. Існуюча система дозволів побудована таким чином, що для кожного розробника потрібно окремо запитувати всі необхідні системи. Крім того, впровадження нових сервісів є більш масштабним, оскільки SSO необхідно впровадити, перш ніж сервіс можна буде використовувати.

• **Стан IaC** : В організації IaC та GitOps активно використовуються. Вони використовуються для більшої частини активної інфраструктури. Для цього існують деякі загальні рекомендації, яких має дотримуватися персонал DevOps. Самообслуговування рекомендовано, але розробникам часто потрібно.

підтримка з боку DevOps-персоналу. Це створює для них непотрібне робоче навантаження.

• **Моніторинг**: Стан моніторингу в організації дуже різниться. Було розпочато нові ініціативи щодо покращення моніторингу в різних проектах та командах, але загалом моніторингу та загальної видимості стану програм бракує.

• **Брак навичок**: Організація використовує зовнішню робочу силу та зазвичай вирішує наймати молодших та середніх працівників у командах. Ефективне використання методологій DevOps вимагає широкого спектру навичок, і деяким командам бракує цих навичок для забезпечення ефективних робочих процесів.

Ці проблеми разом створюють значну неефективність у щоденних операціях, що може призвести до значних втрат часу. Така неефективність не лише уповільнює робочі процеси, але й впливає на загальну продуктивність, зрештою перешкоджаючи здатності компанії досягати цілей та ефективно досягати результатів.

В організації для інфраструктури більшості сервісів, що активно розробляються, використовуються IaC та GitOps. Обраним інструментом є Terraform, і використовуються централізовано керовані користувацькі модулі Terraform. Це означає, що для більшої частини інфраструктури вже існує широко використовуваний рівень абстракції. Впровадження IDP спрощує автоматизацію інфраструктури, оскільки її можна використовувати для управління інфраструктурою.

Ще один пов'язаний інструмент, що використовується в організації, – це GitLab. Він використовується як основний інструмент для управління версіями проектів, а також для неперервної інтеграції (CI) та компакт-дисків (CD). Конфігурації Terraform використовуються для управління проектами та групами. Більшість готових інструментів для IDP сумісні з GitLab, що означає, що інтеграція в IDP для управління версіями проектів буде простішою. Крім того, автоматизація в управлінні проектами означає, що якщо для кожного репозиторію потрібні деякі файли конфігурації, їх можна організувати програмно.

Якщо в представленій організації буде впроваджено IDP, можна очікувати певних переваг. У цьому розділі потенційні переваги класифікуються на три окремі групи: підвищена видимість, встановлені золоті шляхи та загальні

переваги. Крім того, можна визначити деякі інші можливі переваги.

Масштаб та потенціал цих переваг значною мірою залежать від впровадження та рішень, прийнятих щодо процесу впровадження. Ключовим фактором успіху впровадження є кількість людей та робочої сили, виділених для роботи над процесом впровадження. Іншим важливим фактором є запланований кінцевий продукт та його характеристики.

Існує кілька різних аспектів видимості в процесі розробки, які можна покращити. Найбільше покращення можна було б зробити для збільшення видимості між командами та відділами за допомогою функції каталогу послуг. Це забезпечить спосіб представлення широкого спектру інформації про кожну послугу, такої як власність на послугу, поточний стан життєвого циклу послуги, можливі залежності та залежні елементи, а також інша загальна інформація.

Ще однією можливою перевагою видимості може бути стандартизований і навіть обов'язковий спосіб представлення та документування внутрішніх кінцевих точок API сервісів та інформації про те, як отримати доступ до цих кінцевих точок. Таким чином, пошук необхідної інформації для нових інтеграцій буде легшим. IDP також може забезпечити спосіб перевірки журналів сервісів та даних моніторингу. Це можна використовувати на вищих рівнях абстракції, наприклад, щоб побачити, чи є якісь сервіси в певному домені критичні проблеми.

Наразі в організації існують певні бюджети на витрати на хмарну інфраструктуру. Команди отримують попередження, коли вони перевищують бюджет, але немає контексту пояснень для бюджетів. IDP може забезпечити функції для покращення прозорості за допомогою аналізу витрат та відстеження бюджету. Крім того, він також може надати контекст для бюджетів.

Наявність IDP може забезпечити простий спосіб впровадження золотих шляхів в організацію. Поширеним способом впровадження золотих шляхів є створення шаблонів, що відповідають обраним найкращим практикам і технологіям. Наприклад, створення запропонованого робочого процесу для широко використовуваного мікросервісу Java працюватиме як шаблон і керівництво щодо впровадження певних функцій, таких як моніторинг і автоматизоване

тестування. Чітко встановлені золоті шляхи можуть мати багато переваг в організації. Для розробників це чітко представить найкращі мови програмування та фреймворки для використання. Створення нового проекту стане простіше, коли шаблон забезпечить CI та CD, рішення для моніторингу, і розробник зможе бути впевнений, що вибрані технології є правильними.

Золоті шляхи забезпечують стандартизацію, яка сама по собі має аналогічні переваги. Стандартні практики в різних командах спрощують зміну команд. Це також спрощує розробку та розширення фреймворку, оскільки впровадження нових інтеграцій потрібно здійснювати лише один раз .

Хоча підвищена видимість та встановлені «золоті шляхи» вже пропонують значні переваги для розробників, є й інші аспекти, на які це також може вплинути. Завдяки повноцінному IDP розробники потенційно можуть встигати більше, ніж раніше. Створення прототипів та тестування нових ідей стає швидшим для впровадження. Завдяки самообслуговуванню розробники можуть робити більше, ніж покладатися на персонал DevOps. Це також звільняє персонал DevOps, щоб він міг більше зосередитися на власних завданнях. Ці переваги більш очевидні для персоналу, який складається переважно з молодших та середніх посад.

Добре реалізований IDP може працювати як площина керування для всіх робочих процесів, пов'язаних з розробкою. Це зменшить навантаження на розробників, пов'язане з перемиканням контексту. Залежно від обмежень безпеки , IDP також може забезпечити простіший спосіб інтеграції нових сервісів для розробників.Впровадження IDP в організації буде нелегким завданням. У кожній організації такого розміру впровадження нових технологій відбувається повільно. Платформа повинна забезпечувати достатню цінність, щоб подолати тягар впровадження та модифікації існуючих робочих процесів і завдань. Для успішного впровадження IDP необхідні організаційні зміни.

Однією з головних перешкод для впровадження є вартість впровадження. Навіть якщо IDP буде впроваджено з рішеннями з відкритим кодом, знадобляться значні інвестиції в робочий час. Для повноцінного рішення в організації

такого розміру знадобиться спеціальна команда.

Оскільки немає існуючих просуваних фреймворків для послуг, створення нових може виявитися складним. Узгодження технічних та нетехнічних рішень може бути складним завданням. Фреймворки повинні бути відкритими для пропозицій та вдосконалень. В ідеалі всі команди з усіх галузей повинні погодитися щодо певних дизайнів, але цього буде важко досягти.

Наразі не існує стандартизованих способів управління моніторингом сервісів. Різні домени мають різні рішення. Впровадження моніторингу в рамках фреймворку означає або підтримку всіх використовуваних сервісів, або сприяння єдиному моніторингу в рамках фреймворку. В останньому випадку потрібні додаткові зусилля від команд розробників продуктів під час міграції на новий фреймворк.

В організації для інфраструктури використовуються три різні хмарні провайдери. Більша частина інфраструктури знаходиться в межах одного провайдера, але для підтримки всіх існуючих послуг за допомогою IDP необхідно підтримувати всіх трьох провайдерів. Крім того, також використовується деяка локальна інфраструктура. Впровадження IDP в організації мало б потенціал стати трансформаційним кроком для оптимізації та спрощення процесів розробки. У цьому підрозділі окреслено план впровадження IDP у згаданій організації з урахуванням конкретних викликів та потенційних переваг.

Процес впровадження слід спочатку розпочати з платформи MVP, яка спочатку використовується лише кількома командами в одному домені. Цей метод вже перевірено на ефективність в організації. Гарним підходом було б вибрати домен, який має як застарілі, так і новіші проекти, а також широкий спектр різноманітних проектів, щоб спочатку перевірити варіанти використання та переваги. Використання може розпочатися навіть з мінімальною підтримкою функцій, щоб отримати відгуки користувачів якомога раніше. Ширше впровадження не повинно бути нав'язаним, IDP повинна надавати користувачам достатню цінність, щоб вони хотіли використовувати її та включити у свій робочий процес.

Відправною точкою має бути пріоритетність видимості для різних команд

розробників програмного забезпечення. Існує два різні аспекти, чому це має бути першочерговим завданням. По-перше, відсутність видимості була визнана найважливішою проблемою, яка наразі ставить під загрозу ефективність розробки. Таким чином, забезпечення видимості матиме найбільший потенційний вплив. По-друге, створення структури для підвищення видимості є відносно простим і може слугувати основою для всього IDP. Навіть якщо спочатку це буде впроваджено лише для обмеженої кількості команд, покращення видимості може забезпечити значні переваги.

Ще одним важливим моментом має бути створення золотих шляхів на ранній стадії. Спочатку створення лише одного, добре продуманого шляху, який окреслює рекомендовані практики, інструменти та архітектури, може допомогти стандартизувати налаштування проектів. Першим шляхом для організації має бути бекенд-сервіс Java, що відповідає архітектурі мікросервісів, оскільки це найпоширеніший спосіб виконання бекенд-сервісів. Шлях необхідно модифікувати на основі відгуків та внесків, а кінцева мета - мати шлях, з яким можуть погодитися всі користувачі. Крім того, навіть якщо користувач створює проект, дотримуючись золотого шляху, він все одно повинен мати можливість порушити обмеження шляху, якщо це необхідно. Початок лише з одного шляху - це хороший спосіб встановити практики ітерації шляху, з яких інші шляхи для інших технологій можуть багато чого навчитися. В ідеалі повинні бути шляхи для кожного рекомендованого технологічного стеку.

Один із запропонованих способів розпочати впровадження золотих шляхів – це створити внутрішній інструмент CLI, який інші розробники можуть використовувати для створення та зміни проектів. Цей самий інструмент можна використовувати для додавання визначень каталогу послуг в інтерактивному режимі. CLI працює як хороший інтерфейс, з яким більшість розробників вже знайомі, але також має деякі проблеми, такі як необхідність підтримки широкого спектру пристроїв та операційних систем.

Навіть якщо ітеративне впровадження є чудовим способом реалізації IDP, планування перед впровадженням є необхідним. Наявність конкретного та

чіткого плану, на який можна посилатися на різних етапах впровадження, є чудовим способом не збитися з курсу. Організації необхідно виділити певну кількість робочої сили для впровадження. В ідеалі, це має бути повноцінна команда, яка б орієнтувалася на продукт і повністю відповідала за весь IDP. Початкову роботу може розпочати лише одна віддана людина.

У цьому підрозділі обговорюються обмеження, що виникли під час написання цієї роботи. Більшість обмежень пов'язані з тим, що цей термін все ще є досить новим у галузі. У розділі також пропонуються деякі ідеї для майбутньої роботи з цієї ж теми.

На цю роботу значно вплинув той факт, що тему роботи запропонував керівник, а організація, про яку йде мова, не мала внутрішнього попиту на дослідження. Ідея проведення тематичного дослідження виникла під час дослідження теми роботи та виявлення схожих проблем в організації під час роботи в ній.

Ця робота має деякі обмеження. Найбільш помітним є те, що термін «Внутрішня платформа розробника» (IDP), здається, просувається переважно однією компанією, що викликає питання щодо його визнання в галузі. Термін майже не зустрічається в рецензованих публікаціях, а два найпопулярніші онлайн-джерела управляються однією й тією ж компанією. У цій роботі джерела [2 , 16 , 25 , 26, 27] можна простежити до цієї компанії.

Незважаючи на це занепокоєння, важливо визнати, що основні концепції та методології, що приписуються внутрішньо розвиненим підприємницьким системам (ВПР), не є виключними для цього терміна чи компанії. Подібні структури та операційні стратегії очевидні в різних організаціях і обговорювалися в численних публікаціях. Це спостереження свідчить про те, що хоча конкретний термін може бути обмеженим в академічному та галузевому прийнятті, принципи та практики, які він представляє, є актуальними та вже використовуються.

Існує багато різних можливостей для майбутньої роботи з цієї теми. Щоб краще зрозуміти вплив та можливості терміну «Інтегроване розробницьке планування», слід провести кілька тематичних досліджень. Це дасть більш повну та

точнішу картину. Крім того, вивчення організацій, які впровадили подібні системи, забезпечить кращі приклади використання фактичних впроваджень.

Ще однією потенційною темою для дослідження може бути майбутній розвиток ідеології. Оскільки термін IDP все ще є досить новим у галузі розробки програмного забезпечення, буде цікаво спостерігати, як він розвиватиметься та буде прийнятий ширшою аудиторією. Ця робота зосереджена лише на терміні IDP. Майбутні дослідження можуть ширше розглянути подібні ідеології та інструменти, які вже використовуються та забезпечують аналогічні переваги. Ці теми ще не такі популярні в наукових публікаціях. Це може бути пов'язано з тим, що ці проблеми частіше зустрічаються у великих організаціях.

Внутрішня платформа розробника (IDP) інтегрує різні інструменти розробки та експлуатації в єдину систему, яка спрощує та покращує процеси розробки програмного забезпечення. Ключові переваги IDP включають підвищення ефективності робочого процесу, зменшення ручних накладних витрат та посилення співпраці між командами. Ці переваги роблять IDP потенційним рішенням для організацій, які прагнуть покращити життєвий цикл розробки програмного забезпечення та, можливо, виправити деякі недоліки, які вони можуть мати під час поточних впроваджень DevOps. Успіх IDP значною мірою залежить від її впровадження. Вона вимагає ретельного планування, налаштування відповідно до потреб організації та активного управління, щоб залишатися актуальною з урахуванням поточних технологічних змін.

У роботі також було представлено тематичне дослідження потенційного впровадження IDP у великій роздрібній організації. У ньому було висвітлено деякі існуючі проблеми в організації та те, як IDP може допомогти їх вирішити. У дослідженні зазначалося, що успішне впровадження IDP вимагає значних інвестицій ресурсів як у часі, так і в бюджеті . Також було підкреслено необхідність планування для подолання типових проблем, таких як організаційний опір та технічні складності інтеграції. Дослідження пропонує розуміння необхідної підготовки та міркувань для організацій, які розглядають можливість впровадження IDP.

У цій роботі підкреслюється роль, яку можуть відігравати платформи внутрішніх розробників (ВРД) у покращенні практики розробки програмного забезпечення. ВРД можуть призвести до ефективніших циклів розробки, швидшого виходу на ринок та покращення якості програмного забезпечення . Реалізація цих переваг значною мірою залежить від стратегічного та продуманого впровадження. Організації повинні займатися детальним плануванням, забезпечувати достатні ресурси та створювати середовище, відкрите для змін. З розвитком технологій адаптивність систем ВРД матиме вирішальне значення для підтримки ефективності та забезпечення відповідності потребам бізнесу, що постійно змінюються.

3.4 Висновки по розділу

У третьому розділі виконано практичну реалізацію внутрішньої платформи розробника та її інтеграцію у процеси розробки програмного забезпечення. Розглянуто етапи впровадження платформи, її взаємодію з існуючими інструментами та робочими процесами команд. Проведено оцінку ефективності використання IDP, яка підтвердила підвищення продуктивності, зменшення часу доставки змін та покращення узгодженості процесів розробки. Отримані результати демонструють практичну цінність внутрішніх платформ розробника у сучасній інженерії програмного забезпечення.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено техніки створення внутрішніх платформ розробника (Internal Developer Platforms), які є важливим елементом сучасного підходу до автоматизації процесів розробки, розгортання та супроводу програмного забезпечення. У роботі проаналізовано сучасні підходи до Platform Engineering та DevOps, досліджено архітектурні принципи побудови IDP. Процес дослідження охопив аналіз вимог до внутрішніх платформ, моделювання архітектури, дослідження інструментів CI/CD та оцінку ефективності впровадження IDP у сучасному середовищі розробки програмного забезпечення.

На початковому етапі було проведено аналіз предметної області та визначено основні проблеми, які виникають у процесі розробки та експлуатації програмного забезпечення в сучасних IT-системах. Зростання кількості мікросервісів, складності інфраструктури та потреби у швидкому випуску програмних продуктів створює необхідність у стандартизації процесів і централізованій автоматизації. У результаті було визначено ключові функціональні та нефункціональні вимоги до Internal Developer Platforms, серед яких автоматизація процесів CI/CD, забезпечення масштабованості, інтеграція з хмарною інфраструктурою, підтримка контейнеризації та підвищення продуктивності команд розробки.

У першому розділі роботи було досліджено теоретичні основи внутрішніх платформ розробника, концепції DevOps і Platform Engineering, а також сучасні підходи до проєктування IDP. Проведений аналіз показав, що використання внутрішніх платформ дозволяє зменшити складність взаємодії між командами розробки та операційної підтримки, забезпечити повторне використання інфраструктурних компонентів і спростити процеси розгортання програмного забезпечення.

У другому розділі було проаналізовано методи та інструменти побудови Internal Developer Platforms. Досліджено референтні архітектури IDP, основні компоненти платформи та сучасні технології, що використовуються для

автоматизації інфраструктури. У результаті було сформовано архітектурний підхід до створення внутрішньої платформи розробника, який забезпечує масштабованість, гнучкість та спрощення процесів управління інфраструктурою.

У третьому розділі було розглянуто реалізацію внутрішньої платформи розробника та оцінено ефективність її використання. Досліджено процеси впровадження платформи у цикл розробки програмного забезпечення, інтеграцію інструментів автоматизації та організацію взаємодії між компонентами системи. Крім того, впровадження внутрішньої платформи сприяє зменшенню кількості ручних операцій і підвищує ефективність роботи команд розробки.

Загалом, результати виконаної роботи підтверджують доцільність використання Internal Developer Platforms у сучасних IT-компаніях. Використання IDP забезпечує централізоване управління інфраструктурою, автоматизацію DevOps-процесів, спрощення інтеграції сервісів та підвищення продуктивності команд розробників. До переваг обраного підходу можна віднести стандартизацію середовищ розробки, підтримку масштабування, підвищення стабільності інфраструктури та скорочення часу виведення програмних продуктів на ринок.

Разом із тим, у роботі визначено певні обмеження та виклики, пов’язані зі створенням внутрішніх платформ розробника. До них належать складність початкового впровадження IDP, необхідність підтримки великої кількості інтеграцій, а також потреба у високому рівні кваліфікації спеціалістів з Platform Engineering та DevOps. У майбутньому внутрішні платформи розробника можуть бути розширені за рахунок використання технологій штучного інтелекту для автоматизації управління інфраструктурою, впровадження інтелектуального моніторингу та підтримки self-service-підходів для команд розробки.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Argo CD Documentation. (2025). argo-cd.readthedocs.io. https://argo-cd.readthedocs.io/
2. AWS DevOps Documentation. (2025). docs.aws.amazon.com. https://docs.aws.amazon.com/devops/
3. Backstage Documentation. (2025). backstage.io. https://backstage.io/docs/
4. Bass, L., Weber, I., & Zhu, L. (2022). DevOps: A Software Architect's Perspective. addison-wesley.com. https://www.informit.com/store/devops-a-software-architects-perspective-9780134049847
5. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. ACM Queue, 14(1), 70–93. https://doi.org/10.1145/2898442.2898444
6. Cloud Native Computing Foundation. (2025). CNCF Cloud Native Landscape. landscape.cncf.io. https://landscape.cncf.io/
7. Continuous Delivery Foundation. (2025). cd.foundation. https://cd.foundation/
8. Docker Documentation. (2025). docs.docker.com. https://docs.docker.com/
9. Fowler, M. (2018). Continuous Integration. martinfowler.com. https://martinfowler.com/articles/continuousIntegration.html
10. Fowler, M. (2021). Microservices Resource Guide. martinfowler.com. https://martinfowler.com/microservices/
11. GitHub Actions Documentation. (2025). docs.github.com. https://docs.github.com/en/actions
12. GitLab CI/CD Documentation. (2025). docs.gitlab.com. https://docs.gitlab.com/ee/ci/
13. Google Cloud DevOps Guide. (2025). cloud.google.com. https://cloud.google.com/devops

14. Hightower, K., Burns, B., & Beda, J. (2022). Kubernetes: Up and Running. oreilly.com. https://www.oreilly.com/library/view/kubernetes-up-and/9781098110192/

15. HashiCorp Terraform Documentation. (2025). developer.hashicorp.com. https://developer.hashicorp.com/terraform/docs

16. Humble, J., & Farley, D. (2011). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. pearson.com. https://www.pearson.com/en-us/subject-catalog/p/continuous-delivery/P200000003479

17. IBM Cloud Architecture Center. (2025). cloud.ibm.com. https://cloud.ibm.com/architecture

18. Jenkins Documentation. (2025). jenkins.io. https://www.jenkins.io/doc/

19. Kelsey, T., & Newman, S. (2023). Platform Engineering for Modern DevOps Teams. IEEE Software, 40(5), 55–63. https://doi.org/10.1109/MS.2023.3265412

20. Kubernetes Documentation. (2025). kubernetes.io. https://kubernetes.io/docs/

21. Microsoft Azure Architecture Center. (2025). learn.microsoft.com. https://learn.microsoft.com/en-us/azure/architecture/

22. Microsoft Azure DevOps Documentation. (2025). learn.microsoft.com. https://learn.microsoft.com/en-us/azure/devops/

23. Morris, K. (2020). Infrastructure as Code: Managing Servers in the Cloud. oreilly.com. https://www.oreilly.com/library/view/infrastructure-as-code/9781491924334/

24. Netflix Technology Blog. (2024). Platform Engineering at Scale. netflixtechblog.com. https://netflixtechblog.com/

25. Newman, S. (2021). Building Microservices. oreilly.com. https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/

26. Nginx Documentation. (2025). nginx.org. https://nginx.org/en/docs/

27. OpenTelemetry Documentation. (2025). opentelemetry.io. https://opentelemetry.io/docs/

28. OWASP DevSecOps Guideline. (2024). owasp.org. https://owasp.org/www-project-devsecops-guideline/

29. Platform Engineering. (2025). platformengineering.org. https://platformengineering.org/

30. Prometheus Documentation. (2025). prometheus.io. https://prometheus.io/docs/introduction/overview/

31. Pulumi Documentation. (2025). pulumi.com. https://www.pulumi.com/docs/

32. Red Hat OpenShift Documentation. (2025). docs.redhat.com. https://docs.redhat.com/en/documentation/openshift_container_platform/

33. Richardson, C. (2019). Microservices Patterns. manning.com. https://www.manning.com/books/microservices-patterns

34. Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review. IEEE Access, 5, 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629

35. Spotify Engineering Culture. (2024). engineering.atspotify.com. https://engineering.atspotify.com/

36. The Twelve-Factor App. (2025). 12factor.net. https://12factor.net/

37. Turnbull, J. (2014). The Docker Book. jamesturnbull.com. https://dockerbook.com/

38. VMware Tanzu Documentation. (2025). docs.vmware.com. https://docs.vmware.com/en/VMware-Tanzu/index.html

39. Xie, L., & Chen, Y. (2024). Scalable Internal Developer Platforms for Cloud-Native Applications. Journal of Systems and Software, 209, 111932. https://doi.org/10.1016/j.jss.2024.111932

40. Zimmermann, O. (2023). Microservices and Cloud-Native Application Architecture. IEEE Software, 40(2), 18–25. https://doi.org/10.1109/MS.2022.3229981

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: " Техніки створення внутрішніх платформ розробника (Internal Developer Platforms) "

Обсяг пояснювальної записки: 85 аркушів

Дата закінчення бакалаврської роботи 10 червня 2026р.

Підпис студента _____