ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Технології для інтеграції старих інформаційних систем із сучасними стандартами»

на здобуття освітнього ступеня бакалавра
зі спеціальності 123 Комп'ютерна інженерія
освітньо-професійної програми «Комп'ютерна інженерія»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Яна ДУБОВА
(підпис)

Виконав: здобувач вищої освіти групи КІД-41
Яна ДУБОВА

Керівник: Асєєва Л.А.
PhD, доц. каф.

Рецензент: Володимир САДОВЕНКО
к.ф.-м.н., доцент *Ім'я, ПРІЗВИЩЕ*

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерна інженерія

Ступінь вищої освіти Бакалавра

Спеціальність 123 Комп'ютерна інженерія

Освітньо-професійна програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерної інженерії
_____Наталія ЛАЩЕВСЬКА
«_____» _____2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дубова Яна Олександрівна

1. Тема кваліфікаційної роботи: «Технології для інтеграції старих інформаційних систем із сучасними стандартами»
керівник кваліфікаційної роботи Асєєва Л.А.,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» травня 2025 р. № 320.
2. Строк подання кваліфікаційної роботи «17» травня 2025 р.
3. Вихідні дані до кваліфікаційної роботи: науково-технічна література з інтеграції інформаційних систем; технічна документація на API, middleware, інструменти контейнеризації та оркестрації (Docker, Kubernetes); стандарти побудови мікросервісної архітектури; моделі адаптації застарілих (legacy) систем.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Огляд літератури та аналіз предметної області в контексті інтеграції застарілих інформаційних систем із сучасними стандартами.
2. Аналіз архітектурних підходів та технологій (REST, SOAP, gRPC, API Gateway, middleware), які застосовуються для модернізації інформаційних систем.
3. Аналіз типових рішень щодо контейнеризації, автоматизованого

розгортання (Docker, Kubernetes, Ansible) та їх ролі в адаптації legacy-систем.
4. Оцінювання ефективності реалізованого інтеграційного рішення: сумісність, продуктивність, масштабованість.

1. Перелік ілюстративного матеріалу: *презентація*
Актуальність проблеми інтеграції застарілих інформаційних систем у сучасне IT-середовище.
Схема взаємодії компонентів legacy-систем із новітніми платформами.
Огляд архітектур інтеграції: монолітна, мікросервісна, гібридна.
Порівняльна таблиця технологій: REST, SOAP, gRPC, GraphQL.
Використання API Gateway, middleware та проксі-рішень – діаграми та схеми.
Приклади інтеграцій за допомогою Docker, Kubernetes, Ansible.
Графіки навантаження, тестування швидкодії та продуктивності.
Висновки та рекомендації щодо впровадження інтеграційних рішень..

Дата видачі завдання «16» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз науково-технічної літератури з питань інтеграції застарілих інформаційних систем із сучасними стандартами	16.02–23.02.2025	
2	Дослідження сучасних архітектур інтеграції: мікросервіси, API, middleware, проксі-рішення	23.02–02.03.2025	
3	Аналіз типових технологій і протоколів взаємодії (REST, SOAP, gRPC) та середовищ розгортання	02.03–09.03.2025	
4	Розробка архітектури інтеграції та моделювання взаємодії між компонентами	09.03–19.03.2025	
5	Реалізація тестового середовища: Docker, Kubernetes, Ansible, Mock-server	19.03–29.03.2025	
6	Оформлення роботи: вступ, висновки, реферат	29.03–30.03.2025	
7	Підготовка демонстраційних матеріалів та презентації	30.03–31.03.2025	
8	Попередній захист роботи	01.04.2025	

Здобувачка вищої освіти _____ Яна ДУБОВА
(підпис)

Керівник
кваліфікаційної роботи _____
Асєєва Л.А.
(підпис)

РЕФЕРАТ

Текстова частина бакалаврської роботи: 69 сторінок, 25 рисунків, 40 джерел.

Об'єкт дослідження – інформаційні системи, що потребують інтеграції з сучасними технологічними стандартами.

Предмет дослідження – методи та технології інтеграції застарілих (legacy) інформаційних систем із сучасними платформами, включаючи використання API, сервісно-орієнтованої архітектури, контейнеризації та інструментів автоматизованого розгортання.

Мета роботи – розробка та реалізація підходу до інтеграції старих інформаційних систем із сучасними стандартами з використанням віртуалізації, мікросервісної архітектури, API-шлюзів та проміжного програмного забезпечення.

Методи дослідження:

1. Аналіз літературних джерел і нормативних документів – дослідження сучасних методів інтеграції, огляд архітектурних рішень і протоколів взаємодії (REST, SOAP, gRPC).
2. Методи системного проєктування – побудова архітектури інтеграції із використанням API-шлюзів, адаптерів, проксі-серверів та middleware.
3. Методи моделювання – створення структурних схем взаємодії між компонентами, формалізація процесу обміну даними.
4. Порівняльний аналіз – оцінка ефективності сучасних інструментів, таких як Docker, Kubernetes, Ansible, Mock-server, API Gateway.

Галузь використання – модернізація корпоративних інформаційних систем, інтеграція з хмарними сервісами, забезпечення сумісності з новими програмними продуктами, автоматизоване розгортання, розробка сервісно-орієнтованих архітектур, DevOps.

ІНТЕГРАЦІЯ ІНФОРМАЦІЙНИХ СИСТЕМ, LEGACY-СИСТЕМИ, API, MIDDLEWARE, ANSIBLE, DOCKER, KUBERNETES, GRPC, REST, SOAP,

АДАПТЕРИ, СЕРВІСНА АРХІТЕКТУРА, CI/CD

ABSTRACT

The text of the bachelor's thesis: 69 pages, 25 figures, 40 sources.

Object of research – information systems that require integration with modern technological standards.

Subject of research – methods and technologies for integrating outdated (legacy) information systems with modern platforms, including the use of APIs, service-oriented architecture, containerization, and automated deployment tools.

Purpose of the thesis – development and implementation of an approach for integrating legacy information systems with modern standards using virtualization, microservice architecture, API gateways, and middleware.

Research methods:

1. Analysis of literary sources and regulatory documents – study of modern integration methods, review of architectural solutions and interaction protocols (REST, SOAP, gRPC).
2. System design methods – designing integration architecture using API gateways, adapters, proxy servers, and middleware.
3. Modeling methods – creating structural interaction schemes between components and formalizing the data exchange process.
4. Comparative analysis – evaluating the effectiveness of modern tools such as Docker, Kubernetes, Ansible, Mock-server, and API Gateway.

Field of application – modernization of corporate information systems, integration with cloud services, ensuring compatibility with new software products, automated deployment, development of service-oriented architectures, and DevOps.

Keywords: INFORMATION SYSTEMS INTEGRATION, LEGACY SYSTEMS, API, MIDDLEWARE, ANSIBLE, DOCKER, KUBERNETES, GRPC, REST, SOAP, ADAPTERS, SERVICE-ORIENTED ARCHITECTURE, CI/CD.

ЗМІСТ

ВСТУП 10

1 ОГЛЯД ТА АНАЛІЗ СУЧАСНИХ СТАНДАРТІВ ІНТЕГРАЦІЇ ІНФОРМАЦІЙНИХ СИСТЕМ 12

1.1 Поняття та класифікація інформаційних систем 12

1.2 Проблеми застарілих ІС та виклики при їх модернізації 15

1.3 Огляд сучасних стандартів інтеграції (REST, gRPC, SOAP, GraphQL) 19

1.4 Хмарні моделі (Public, Private, Hybrid) як середовище для модернізації ІС 23

2 ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ ІНТЕГРАЦІЇ 26

2.1 Підхід «інфраструктура як код» (IaC) для автоматизації інтеграції 26

2.2 Інструменти конфігурації: Ansible, Puppet, Chef – порівняння можливостей 28

2.3 Використання контейнерів і оркестрації (Docker, Kubernetes) 30

2.4 Сервіси-посередники: API Gateway, ESB, Service Mesh 32

3 РОЗРОБКА ІНТЕГРАЦІЙНОГО РІШЕННЯ ДЛЯ СТАРИХ ІНФОРМАЦІЙНИХ СИСТЕМ 35

3.1 Розробка middleware-рівня для інтеграції спадкових систем 35

3.2 Обґрунтування вибору Kubernetes як платформи оркестрації інтеграційних сервісів 39

3.3 Впровадження внутрішнього веб-сервісу в архітектуру інтеграції 41

3.4 Розробка інтеграційних модулів за допомогою Ansible 45

3.5 Приклад роботи інтеграційного рішення на основі сервісно-орієнтованої архітектури 51

4 ТЕСТУВАННЯ ТА ОЦІНКА ІНТЕГРАЦІЙНОГО РІШЕННЯ 55

4.1 Методологія тестування сумісності при взаємодії застарілих і сучасних систем 55

4.2 Етапи розробки, впровадження та інтеграції архітектури 56

4.3 Керівництво з розробки адаптерів і сервісних компонентів 63

4.4 Рекомендації для адміністратора щодо підтримки інтеграційної інфраструктури 64

ВИСНОВКИ 68

ПЕРЕЛІК ПОСИЛАНЬ 70

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 74

ВСТУП

У сучасних умовах стрімкого розвитку інформаційних технологій багато підприємств і організацій продовжують використовувати застарілі (legacy) інформаційні системи, які були створені десятки років тому. Такі системи часто є критично важливими для функціонування бізнес-процесів, але вони не відповідають сучасним вимогам безпеки, масштабованості та сумісності з новими цифровими платформами.

Інтеграція старих систем із сучасними технологічними стандартами дозволяє зберегти накопичені дані, мінімізувати витрати на повну заміну програмного забезпечення та забезпечити поступовий перехід до нових архітектур. У цьому контексті особливого значення набувають технології мікросервісної архітектури, API-шлюзи, контейнеризація, інструменти автоматизованого розгортання, а також підхід «інфраструктура як код» (IaC).

Актуальність теми зумовлена потребою підприємств модернізувати свої IT-інфраструктури без ризиків для стабільності бізнесу, забезпечити інтеграцію з хмарними сервісами та реалізувати стратегії цифрової трансформації.

Мета роботи – розробка та реалізація підходу до інтеграції старих інформаційних систем із сучасними стандартами з використанням віртуалізації, мікросервісної архітектури, API-шлюзів та проміжного програмного забезпечення.

Об'єкт дослідження – інформаційні системи, що потребують інтеграції з сучасними технологічними стандартами.

Предмет дослідження – методи та технології інтеграції застарілих (legacy) інформаційних систем із сучасними платформами, включаючи використання API, сервісно-орієнтованої архітектури, контейнеризації та інструментів автоматизованого розгортання.

Методологія дослідження базується на комплексному підході до аналізу проблеми інтеграції застарілих інформаційних систем із сучасними технологічними стандартами. У роботі були використані такі методи:

– аналіз літературних джерел та нормативних документів, що дозволив дослідити сучасні підходи до побудови інтеграційних рішень, а також вивчити особливості архітектурних стандартів і протоколів взаємодії (REST, SOAP, gRPC).

– методи системного проєктування, які дали змогу побудувати архітектуру інтеграції на основі API-шлюзів, middleware, проксі-серверів і адаптерів.

– методи моделювання, що включали розробку структурних схем взаємодії між модулями системи, формалізацію процесу обміну даними.

– порівняльний аналіз, завдяки якому оцінено ефективність інструментів для автоматизованого розгортання та інтеграції, таких як Docker, Kubernetes, Ansible, Mock-server, API Gateway.

– емпіричні методи, які використовувались на етапі реалізації інтеграційного рішення, включаючи симуляцію обміну даними, тестування API-сумісності та оцінку надійності й масштабованості.

Практична значущість роботи полягає в розробці інтеграційної архітектури, яка може бути застосована для модернізації застарілих інформаційних систем без необхідності їх повної заміни. Запропонований підхід дозволяє забезпечити сумісність старих ІС із сучасними платформами, хмарними сервісами та програмними продуктами.

Використання таких технологій, як мікросервісна архітектура, API-шлюзи, контейнеризація та інфраструктура як код (IaC), дає змогу підвищити гнучкість і масштабованість інформаційних систем, оптимізувати процес розгортання та підтримки ПЗ, а також знизити витрати підприємств на ІТ-модернізацію.

Результати роботи можуть бути впроваджені в ІТ-відділах компаній, що стикаються з проблемою підтримки та розвитку застарілих ІС, або використані в освітніх програмах для підготовки фахівців у сфері DevOps, інтеграційних технологій та програмної інженерії.

1 ОГЛЯД ТА АНАЛІЗ СУЧАСНИХ СТАНДАРТІВ ІНТЕГРАЦІЇ ІНФОРМАЦІЙНИХ СИСТЕМ

1.1 Поняття та класифікація інформаційних систем

Інформаційна система (ІС) – це сукупність взаємопов’язаних компонентів, призначених для збору, обробки, зберігання, пошуку, розповсюдження та використання інформації з метою підтримки прийняття рішень, управління та координації діяльності в межах організації. За визначенням, наведеним у стандарті ISO/IEC 2382, інформаційна система – це організована комбінація людей, процесів, апаратних засобів, програмного забезпечення, мереж і даних, які забезпечують створення, збереження, обробку й передачу інформації [1].

Основними функціями інформаційної системи є:

– збір даних із внутрішніх або зовнішніх джерел;

– обробка інформації відповідно до заданих алгоритмів;

– зберігання результатів обробки у базах даних;

– передача інформації користувачам або іншим системам;

– забезпечення зворотного зв’язку і логування дій.

До основних компонентів інформаційної системи відносяться:

– апаратне забезпечення (hardware) – фізичні пристрої, які забезпечують роботу системи: сервери, комп’ютери, мережеве обладнання, контролери введення/виведення;

– програмне забезпечення (software) – набір програм, які виконують обробку даних, керують процесами, реалізують інтерфейси та бізнес-логіку. Це можуть бути як прикладні програми, так і системне ПЗ (операційні системи, СУБД, драйвери);

– база даних (data) – структурована сукупність взаємопов’язаних даних, що зберігаються і використовуються для підтримки функціонування ІС. Дані є центральним елементом, оскільки саме навколо них будуються інші компоненти системи;

– персонал (people) – користувачі та фахівці, які працюють із системою: адміністративний персонал, аналітики, розробники, системні адміністратори, кінцеві користувачі. Їх роль полягає у забезпеченні належної експлуатації та розвитку ІС;

– бізнес-процеси (procedures) – правила, політики, інструкції та методики, що регулюють, як і для чого використовуються інформаційні ресурси. Саме вони формують логіку дій у системі;

– мережеве забезпечення (network) – канали і протоколи передачі даних, які дозволяють компонентам системи взаємодіяти між собою та з іншими системами [2].

Інформаційні системи класифікуються за низкою ознак, залежно від функціонального призначення, архітектури побудови, рівня управління, технологічних підходів (рис. 1.1). Така класифікація дозволяє краще зрозуміти призначення системи, визначити її місце в організаційній структурі та оцінити можливості щодо інтеграції чи модернізації.



Рисунок 1.1 – Класифікація інформаційних систем

Класифікація ІС за функціональним призначенням:

– ERP-системи (Enterprise Resource Planning) – інтегровані рішення для планування ресурсів підприємства. Зазвичай охоплюють фінанси, виробництво, логістику, HR. Приклади: SAP ERP, Oracle ERP Cloud, Microsoft Dynamics [3].

– CRM-системи (Customer Relationship Management) – системи для управління взаємодією з клієнтами. Застосовуються для обліку продажів, ведення клієнтських баз, автоматизації маркетингових кампаній. Приклади: Salesforce, Zoho CRM.

– SCM-системи (Supply Chain Management) – рішення для управління ланцюгами постачання, контролю запасів, логістики, оптимізації поставок. Приклади: SAP SCM, Oracle SCM.

– HRM-системи (Human Resource Management) – автоматизують облік персоналу, нарахування заробітної плати, оцінку продуктивності тощо.

– BI-системи (Business Intelligence) – системи для аналітики, побудови звітів, візуалізації даних та прийняття управлінських рішень [4].

Класифікація ІС за рівнем управління:

– операційні системи управління – підтримують щоденну діяльність організації: облік транзакцій, обробка замовлень, випуск продукції.

– тактичні системи управління – забезпечують планування на середньостроковий період, моніторинг ефективності, контроль виконання KPI.

– стратегічні системи управління – орієнтовані на підтримку прийняття рішень топ-менеджментом, визначення довгострокових цілей та стратегії розвитку.

Класифікація ІС за архітектурою:

– монолітна архітектура – система побудована як єдиний програмний блок. Всі компоненти тісно пов’язані, що ускладнює масштабування та модернізацію. Типовий для застарілих ІС.

– клієнт-серверна архітектура – розділення функцій між клієнтською частиною (інтерфейс користувача) та серверною частиною (обробка даних). Такий підхід став основою для корпоративних систем 1990–2000-х років.

– хмарна архітектура (Cloud-based) – системи розгортаються на віддалених серверах з використанням моделей хмарних обчислень (SaaS, PaaS, IaaS). Переваги: гнучкість, масштабованість, доступність з будь-якої точки світу.

– мікросервісна архітектура – система розбита на незалежні модулі (сервіси), які взаємодіють через API. Дає можливість поступового оновлення компонентів і є ідеальною для інтеграції з legacy-системами [5].

Застарілі інформаційні системи (legacy systems) продовжують активно використовуватись у різних організаціях, незважаючи на їх обмежену сумісність із сучасними технологіями. Вони можуть бути критично важливими для бізнес-процесів, проте часто характеризуються низькою гнучкістю, складністю інтеграції та високими витратами на підтримку. У табл. 1.1 наведено основні види таких систем, приклади їх реалізації та типові сфери використання.

Таблиця 1.1 – Види застарілих інформаційних систем (legacy systems)

Вид	**Приклад**
Мейнфрейм-системи	IBM System/360, програми на COBOL, що обробляють фінансові транзакції
Монолітні прикладні програми	ERP-система SAP R/3 у своїх перших версіях
Системи з фіксованою архітектурою	Програми, що працюють на Windows NT, UNIX 90-х років
Програми, написані застарілими мовами програмування	Облікові/логістичні системи на COBOL 80–90-х років
Вбудовані системи з обмеженим оновленням програмного забезпечення	Промислові контролери без оновлення ПЗ

1.2 Проблеми застарілих ІС та виклики при їх модернізації

Серед найбільш критичних проблем, що виникають при використанні застарілих інформаційних систем, на перший план виходить їхня несумісність із сучасними технологічними стандартами. Багато таких систем були розроблені десятиліття тому на базі архітектур і платформ, які більше не підтримуються

виробниками. Вони не здатні ефективно взаємодіяти з новітніми програмними інтерфейсами (API), базами даних, мережевими протоколами або операційними системами, що унеможливлює інтеграцію з сучасним програмним забезпеченням [6]. Це створює суттєві труднощі для підприємств, які прагнуть впроваджувати цифрову трансформацію або автоматизувати процеси, адже кожна спроба об'єднати нову систему зі старою вимагає значних витрат часу, коштів і спеціалізованих знань.

Поширеною проблемою є вкрай обмежена масштабованість застарілих рішень. Більшість із них були розроблені під специфічні потреби, з урахуванням конкретного обсягу навантаження, що був актуальним на момент створення. Збільшення кількості користувачів, обсягів даних або нових функціональних вимог призводить до перевантаження таких систем або взагалі унеможливлює їх використання без суттєвої переробки. Це негативно впливає на гнучкість і адаптивність IT-інфраструктури, знижує конкурентоспроможність компанії та уповільнює її розвиток.

Окрім технічних обмежень, проблему становить відсутність повної або актуальної документації. Дуже часто застарілі системи супроводжуються лише фрагментованими описами або технічними примітками, які втратили свою актуальність через багаторазові зміни або модернізації впродовж років. У деяких випадках документація взагалі відсутня, особливо якщо розробка здійснювалась внутрішніми силами підприємства без суворого дотримання стандартів проєктування. Це ускладнює навчання нових співробітників, підвищує ризик помилок у процесі обслуговування та створює залежність від окремих фахівців, які мають практичний досвід роботи з системою [7].

Підтримка таких систем потребує значних фінансових та людських ресурсів. Часто для обслуговування застарілих платформ необхідно залучати вузькопрофільних спеціалістів, які мають знання старих мов програмування, специфічних бібліотек або апаратних компонентів. У багатьох випадках навіть апаратне забезпечення, на якому функціонує система, більше не випускається і його ремонт або заміна перетворюється на складне завдання. Усе це призводить

до високих експлуатаційних витрат та уповільнення розвитку IT-інфраструктури загалом.

З боку кібербезпеки, застарілі системи часто не підтримують сучасних засобів захисту, не отримують регулярних оновлень безпеки і не відповідають вимогам щодо збереження персональних або конфіденційних даних. Уразливості, які не усуваються, можуть бути легко використані зловмисниками для здійснення атак, викрадення даних або компрометації внутрішніх систем [8]. У контексті сучасних загроз, таких як фішинг, програмне забезпечення-вимагач, атак нульового дня, подібна ситуація є неприйнятною і вимагає оперативного реагування.

Загалом, наявність у компанії застарілих інформаційних систем створює низку стратегічних та операційних ризиків. Вона ускладнює адаптацію до ринкових змін, впровадження інновацій, порушує принципи гнучкості та безперервності бізнесу. У зв'язку з цим модернізація, часткова заміна або побудова гібридної IT-архітектури, здатної поєднати старі та нові компоненти, стає необхідною умовою цифрової трансформації організацій.

Основні проблеми застарілих інформаційних систем та їх потенційний вплив на діяльність організації наведені в табл. 1.2.

Таблиця 1.2 – Основні проблеми застарілих інформаційних систем

Проблема	**Наслідки**
Несумісність із сучасними технологіями	Ускладнення інтеграції з новими системами, ізольованість даних
Обмежена масштабованість	Неможливість розширення функціоналу, уповільнення розвитку
Відсутність або застаріла документація	Залежність від окремих працівників, ризик помилок у супроводі та модифікації системи
Високі витрати на підтримку	Потреба у вузькопрофільних спеціалістах, зростання витрат на обслуговування
Низький рівень кібербезпеки	Вразливість до атак, витоки даних, невідповідність сучасним стандартам захисту інформації

Повна заміна застарілих інформаційних систем на нові рішення на перший погляд здається оптимальним способом вирішення накопичених проблем, проте на практиці цей підхід супроводжується низкою суттєвих ризиків. Найбільшим з них є ризик переривання критичних бізнес-процесів [10]. У разі невдалої міграції або затримок у впровадженні нової системи може виникнути ситуація, коли підприємство втратить доступ до ключових даних або можливість обслуговувати клієнтів. Такий сценарій особливо небезпечний для організацій, які працюють у режимі безперервного надання послуг (банки, логістика, медицина тощо).

Ще одним ризиком є значні фінансові витрати, пов’язані з придбанням нових ліцензій, оновленням обладнання, навчанням персоналу та технічною підтримкою. Для багатьох організацій повна модернізація означає виділення великих бюджетів, які не завжди є виправданими в короткостроковій перспективі. До того ж, через відсутність повної документації або втрату експертизи частина функціоналу старої системи може бути втрачена або неправильно реалізована у новій системі, що призводить до втрати бізнес-логіки або зниження ефективності.

У зв’язку з цим, все більше організацій надають перевагу поступовій інтеграції нових рішень замість повної одномоментної заміни застарілої системи. Такий підхід дозволяє поетапно впроваджувати нові компоненти, зберігаючи при цьому роботу існуючої ІС у режимі підтримки. Інтеграція може відбуватися на рівні окремих модулів, сервісів або інтерфейсів API, що значно знижує ризики втрат і дозволяє краще контролювати хід модернізації.

Поступовий підхід сприяє збереженню експертного досвіду, адже працівники мають змогу адаптуватися до нових рішень у комфортному темпі. Це також створює умови для гнучкого тестування, виявлення та усунення недоліків без тиску дедлайнів, характерного для повного переходу.

У більшості випадків поступова інтеграція є більш безпечним, економічно доцільним і стратегічно виправданим варіантом оновлення ІТ-інфраструктури, особливо для підприємств із критичними процесами або великим обсягом застарілих компонентів.

1.3 Огляд сучасних стандартів інтеграції (REST, gRPC, SOAP, GraphQL)

У процесі модернізації застарілих інформаційних систем одним із ключових викликів є забезпечення ефективної взаємодії між старими платформами та новими цифровими рішеннями. Для цього використовуються стандартизовані протоколи інтеграції, які дозволяють обмінюватися даними між різнорідними компонентами IT-інфраструктури. Серед найпоширеніших сучасних підходів виокремлюють REST, SOAP, gRPC та GraphQL. Розглянемо принципи їх роботи, переваги та недоліки в контексті побудови гібридних систем [12].

Representational State Transfer (REST) – це архітектурний стиль для побудови веб-сервісів, що базується на протоколі HTTP. Він використовує прості HTTP-методи (GET, POST, PUT, DELETE) для виконання CRUD-операцій з ресурсами, представленими у форматі JSON або XML.

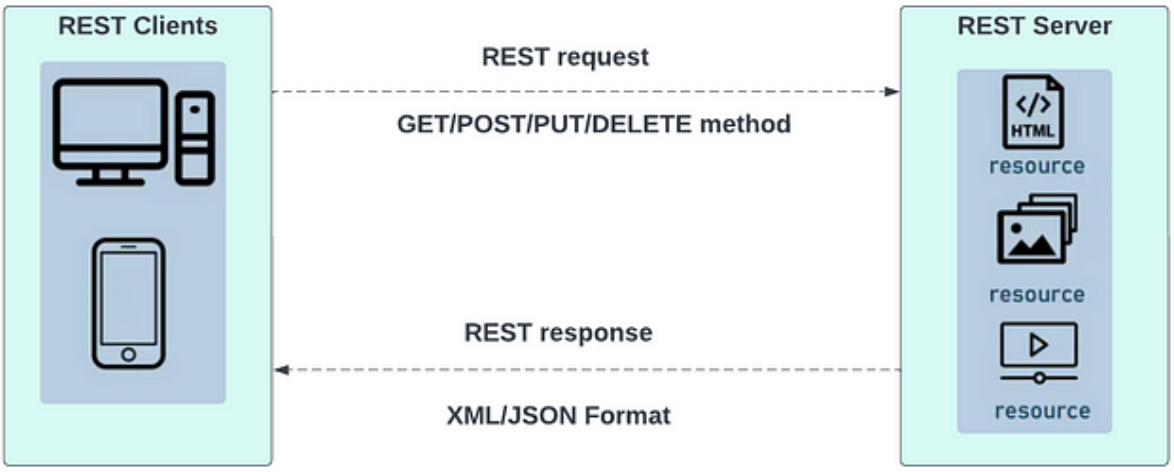


Рисунок 1.2 – Принцип роботи REST-архітектури

Переваги:

- Простота реалізації та розуміння;
- Підтримується практично всіма сучасними мовами і платформами;
- Гнучкий і легкий для налаштування;
- Висока сумісність із веб-браузерами та мобільними додатками.

Недоліки:

– Обмежена формалізація запитів, що може ускладнювати контроль структури повідомлень;

– Не підходить для високопродуктивної двосторонньої комунікації.

Доцільність використання: REST ідеально підходить для інтеграції нових веб-сервісів зі старими ІС, що можуть працювати з HTTP-протоколом і обробляти прості запити [13].

Simple Object Access Protocol (SOAP) – це протокол обміну повідомленнями, який використовує XML для структуризації запитів. Він був розроблений як стандарт для корпоративних інтеграцій із високим рівнем надійності.

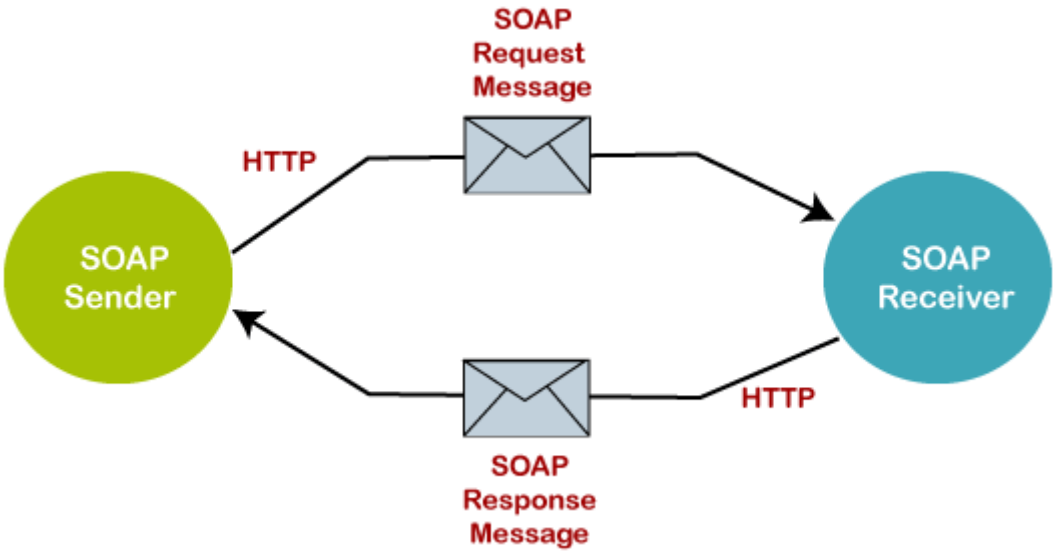


Рисунок 1.3 – Схема обміну повідомленнями в SOAP-протоколі між відправником і отримувачем

Переваги:

– Високий рівень безпеки (WS-Security);

– Підтримка транзакцій і перевірки достовірності;

– Працює не лише через HTTP, але й інші протоколи (SMTP, FTP тощо);

– Детальне описання інтерфейсів (WSDL).

Недоліки:

– Складність у реалізації;

– Надмірність через використання XML;

– Повільніше оброблення в порівнянні з REST [13].

Доцільність використання: SOAP доцільний у великих організаціях або при взаємодії з державними/банківськими системами, де потрібна висока надійність і сувора структура обміну.

Google Remote Procedure Call (gRPC) – це сучасний фреймворк для віддаленого виклику процедур, який використовує Protocol Buffers (protobuf) як формат серіалізації даних. Він забезпечує високу продуктивність та ефективну двосторонню комунікацію.

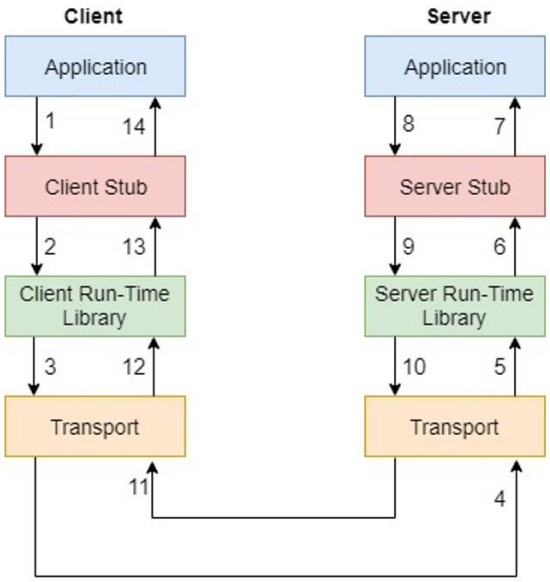


Рисунок 1.4 – Взаємодія клієнта і сервера в архітектурі RPC (Remote Procedure Call)

Переваги:

– Дуже швидка передача даних завдяки protobuf;

– Підтримка потокових запитів і двостороннього обміну;

– Автоматична генерація коду клієнтів і серверів;

– Оптимізовано для мікросервісної архітектури.

Недоліки:

– Обмежена підтримка в старих системах;

– Високий поріг входу: потрібно налаштовувати специфічне середовище;

– Менш зручний для людини формат (на відміну від JSON).

Доцільність використання: gRPC доцільний для високонавантажених внутрішніх сервісів, коли важлива швидкість і масштабованість, але малоймовірний у випадку прямої інтеграції із застарілими ІС.

GraphQL – це мова запитів до API, розроблена компанією Facebook. Вона дозволяє клієнту отримати саме ті дані, які потрібні, у потрібній структурі.

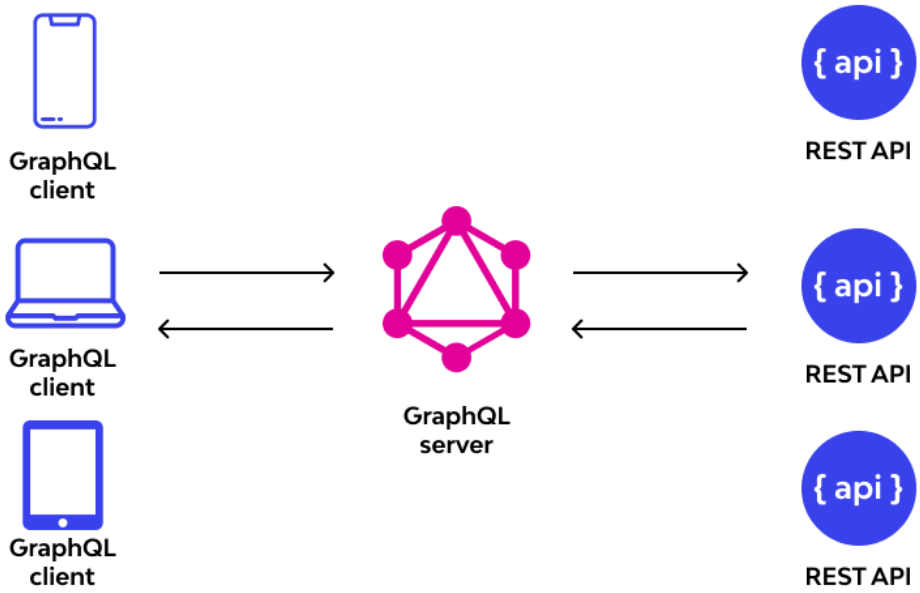


Рисунок 1.5 – Схема взаємодії GraphQL-сервера з клієнтами та REST API

Переваги:

– Ефективне отримання даних: немає надлишковості;

– Гнучкість у формуванні запитів;

– Одна кінцева точка (endpoint) для всіх запитів;

– Зменшення кількості HTTP-запитів.

Недоліки:

– Більш складна логіка реалізації на сервері;

– Слабша підтримка кешування;

– Менш придатний для трансляції складних транзакцій.

Доцільність використання: GraphQL доцільний у мобільних застосунках та клієнт-серверних рішеннях, де важлива гнучкість запитів. Але у зв'язку з вимогами до реалізації – малопридатний для взаємодії зі старими системами.

Розуміння особливостей різних стандартів інтеграції дозволяє обґрунтовано підходити до вибору технології при модернізації застарілих систем. Кожен протокол має власні переваги й обмеження, які можуть бути як критичними, так і незначними в залежності від специфіки проєкту. У контексті взаємодії старих і нових ІС важливо враховувати рівень підтримки протоколу в існуючому середовищі, вимоги до безпеки, обсяги переданих даних, складність впровадження та масштаб проєкту. Нижче у табл. 1.3 наведено порівняльну характеристику чотирьох найбільш поширених стандартів інтеграції – REST, SOAP, gRPC та GraphQL – із урахуванням їх основних властивостей та доцільності використання у різних сценаріях.

Таблиця 1.3 Порівняння поширених стандартів інтеграції

Протокол/технологія	**Формат даних**	**Швидкість**	**Складність реалізації**	**Підтримка в legacy-системах**	**Оптимально для**
REST	JSON/XML	Середня	Низька	Висока	Веб і мобільні сервіси
SOAP	XML	Низька	Висока	Висока	Корпоративні системи
gRPC	Protocol Buffers	Висока	Висока	Низька	Мікросервіси, високонавантажені системи
GraphQL	JSON	Висока	Середня	Низька	Клієнтські додатки з гнучкими запитами

1.4 Хмарні моделі (Public, Private, Hybrid) як середовище для модернізації ІС

Модернізація застарілих інформаційних систем дедалі частіше реалізується із використанням хмарних технологій, що забезпечують гнучке, масштабоване та економічно доцільне середовище для впровадження нових функцій, зберігання даних та інтеграції з зовнішніми сервісами. В залежності від архітектурних і бізнес-вимог, підприємства можуть обирати між публічною (public), приватною (private) або гібридною (hybrid) моделлю хмарної інфраструктури.

Короткий опис хмарних моделей:

– публічна хмара (Public Cloud) – модель, за якої IT-ресурси (сервери, сховища, служби) надаються стороннім постачальником через Інтернет. Приклади: Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP). Ресурси розподіляються між клієнтами за принципом мультиоренди (multi-tenancy) [14].

– приватна хмара (Private Cloud) – інфраструктура, створена спеціально для однієї організації. Вона може бути розміщена у власному дата-центрі або орендована в окремому сегменті хмарного постачальника. Такий підхід забезпечує більший контроль над безпекою та конфігурацією.

– гібридна хмара (Hybrid Cloud) – поєднує публічні та приватні ресурси, дозволяючи організації зберігати критичні дані локально, а менш чутливі процеси – виносити в публічну хмару. Це забезпечує гнучкість та оптимізацію ресурсів.

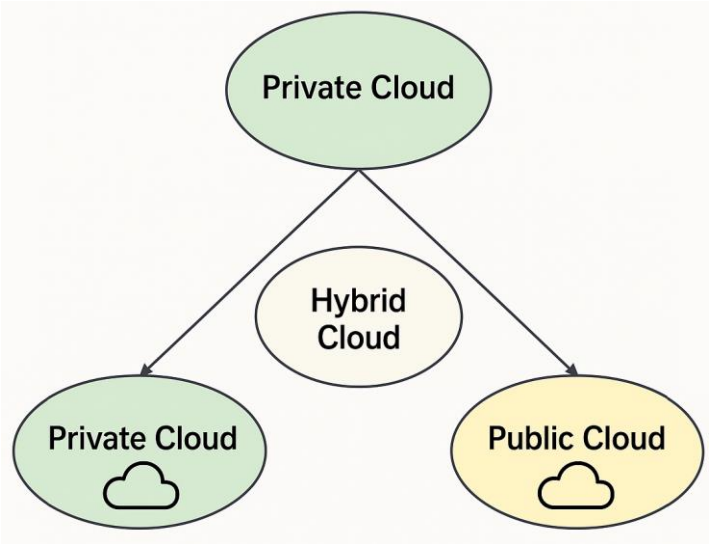


Рисунок 1.5 – Візуальне порівняння хмарних моделей: Private, Public та Hybrid Cloud

Використання хмарних платформ у процесі модернізації ІС має низку ключових переваг. Перш за все, це висока доступність – сервіси доступні з будь-якої точки світу за умови наявності Інтернет-з'єднання, що спрощує інтеграцію географічно розподілених структур або мобільних рішень. По-друге, хмара забезпечує масштабованість: організація може оперативно додавати або зменшувати ресурси відповідно до поточних потреб, не витрачаючи кошти на надлишкову інфраструктуру.

Хмарні платформи дозволяють значно скоротити витрати на розгортання та обслуговування ІТ-систем. Не потрібно інвестувати в обладнання, енергоспоживання, резервування чи фізичне обслуговування серверів – усі ці функції виконує постачальник послуг. Також більшість хмарних рішень передбачають автоматичне оновлення, захист від збоїв і вбудовані засоби безпеки, що знижує ризики.

Гібридна хмарна модель дозволяє уникнути ризиків, пов'язаних із повною заміною застарілої ІС, та поетапно переміщати частини функціоналу в хмарне середовище. Наприклад, організація може залишити критичну ERP-систему локально, а аналітику, зберігання резервних копій або фронтенд-сервіси перенести в публічну хмару.

Хмарні сервіси, такі як AWS Lambda, Azure Functions або Google Cloud Run, дозволяють реалізовувати мікросервісну архітектуру, при якій окремі частини системи можуть взаємодіяти через API або брокери повідомлень (наприклад, Apache Kafka, MQTT), навіть якщо вони розміщені в різних середовищах. Такий підхід суттєво спрощує інтеграцію, підвищує надійність та забезпечує масштабованість без жорсткої прив'язки до фізичної інфраструктури.

2 ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ ІНТЕГРАЦІЇ

2.1 Підхід «інфраструктура як код» (IaC) для автоматизації інтеграції

У процесі модернізації та інтеграції застарілих інформаційних систем особливе значення набуває підхід «інфраструктура як код» (Infrastructure as Code, IaC), який дозволяє автоматизувати розгортання, налаштування та масштабування IT-інфраструктури шляхом використання програмного коду. Цей підхід докорінно змінює традиційні уявлення про керування серверами, мережами, сховищами даних та іншими компонентами, роблячи їх частиною контрольованого, версіонованого та відтворюваного програмного середовища.

IaC — це методологія, що передбачає опис конфігурації інфраструктури за допомогою текстових конфігураційних файлів, які зберігаються в системах контролю версій (наприклад, Git). Це дозволяє застосовувати ті самі принципи, що й у розробці ПЗ: версіонування, автоматичне тестування, спільна робота над кодом, повторне використання шаблонів. Наприклад, замість ручного налаштування віртуальної машини адміністратор описує її конфігурацію в декларативному файлі, а система розгортання автоматично створює її згідно з описом [15].

Застосування IaC є надзвичайно ефективним у середовищах, де потрібно швидко та багаторазово розгортати інфраструктуру — як у хмарі, так і у локальних дата-центрах. Наприклад, під час переходу від монолітної системи до мікросервісної архітектури з використанням контейнерів (Docker) та оркестрації (Kubernetes), IaC забезпечує автоматичне створення відповідного середовища з урахуванням мережевих зв’язків, змінних середовища, правил безпеки тощо.

Головна перевага IaC полягає у повторюваності та передбачуваності. Оскільки вся інфраструктура описується у вигляді коду, будь-яке середовище — тестове, девелоперське, продакшн — може бути створене ідентично за одним шаблоном. Це зменшує кількість людських помилок, підвищує надійність і дозволяє швидко відновити працездатність системи в разі збою. Крім того,

інтеграція IaC у пайплайни CI/CD (Continuous Integration / Continuous Delivery) забезпечує автоматичне розгортання при оновленні коду — це особливо корисно при впровадженні гібридної архітектури, де нові модулі поступово додаються до старої ІС.

IaC також полегшує процес переміщення компонентів у хмарні середовища, що актуально при переході від локальних рішень до публічної або гібридної хмари. Наприклад, опис інфраструктури, створеної у локальному середовищі, може бути адаптований до хмарної платформи, такої як AWS або Azure, із мінімальними змінами конфігурації.

На сьогодні існує велика кількість інструментів для реалізації підходу IaC, серед яких найпоширенішими є:

– Terraform — декларативний інструмент від HashiCorp, який дозволяє керувати інфраструктурою у багатьох хмарах (AWS, Azure, GCP, OpenStack тощо). Підтримує модульну структуру, залежності між ресурсами, стан системи (state) і чудово інтегрується з CI/CD [16].

– Ansible — інструмент конфігураційного менеджменту, що використовує YAML-файли (playbooks) для опису бажаного стану систем. Перевагою є його агентless-модель — для виконання інструкцій не потрібно встановлювати клієнтське ПЗ на керованих вузлах.

– Puppet та Chef — інші відомі системи керування конфігурацією, які використовуються переважно в корпоративних середовищах. Вони мають потужну мову опису конфігурацій і широкі можливості автоматизації.

– CloudFormation (для AWS) та Azure Resource Manager — спеціалізовані рішення для створення інфраструктури у відповідних хмарних платформах. Вони дозволяють повністю автоматизувати розгортання ресурсів у межах екосистеми постачальника.

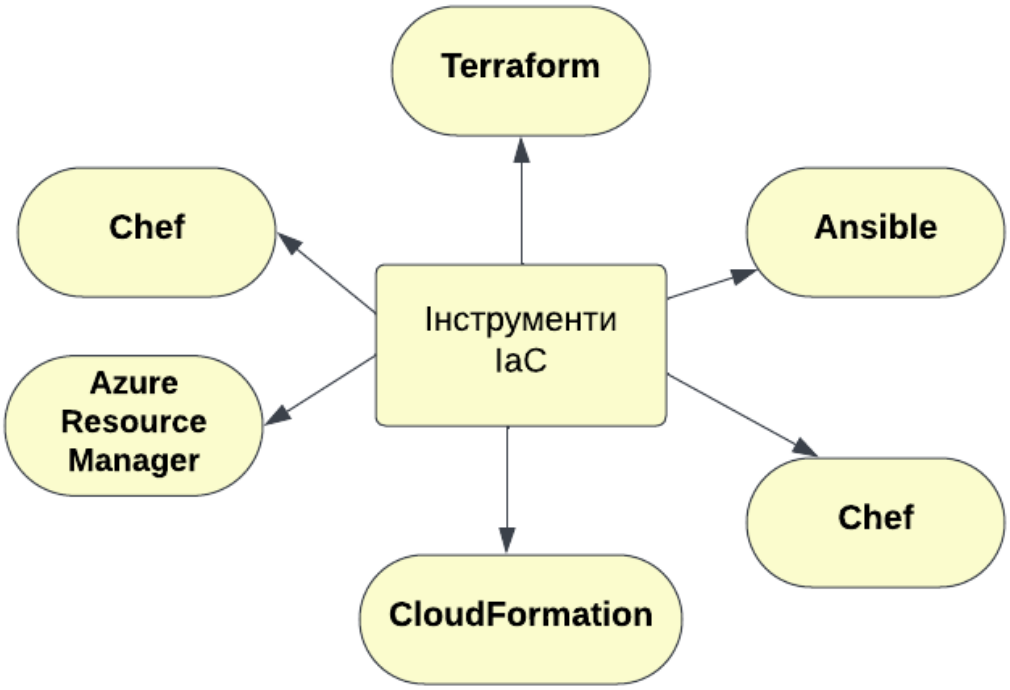


Рисунок 2.1 – Інструменти для реалізації підходу IaC

Підхід «інфраструктура як код» стає фундаментальним компонентом сучасної IT-архітектури, особливо в контексті поетапної інтеграції старих і нових інформаційних систем. Його впровадження значно знижує ризики, прискорює процес модернізації та створює умови для безперервного розгортання нових функціональних блоків у рамках гібридної IT-моделі.

2.2 Інструменти конфігурації: Ansible, Puppet, Chef – порівняння можливостей

У процесі автоматизації управління інфраструктурою та модернізації інформаційних систем важливу роль відіграють інструменти конфігураційного менеджменту. Серед найпоширеніших на ринку варто виділити Ansible, Puppet та Chef. Вони дозволяють централізовано керувати налаштуванням серверів, розгортанням програмного забезпечення, забезпеченням безперервності та відтворюваності середовищ.

Ansible — це агентless-система, яка використовує SSH для підключення до вузлів і YAML-файли (playbooks) для опису бажаного стану інфраструктури. Основна перевага Ansible — простота у використанні та низький поріг входу. Інструмент не потребує встановлення клієнта на кожну машину, а конфігураційні файли читаються як звичайні сценарії.

Puppet — це система конфігураційного менеджменту з декларативною мовою власного DSL (domain-specific language). Вона використовує модель «клієнт-сервер», де кожен вузол отримує конфігурацію з центрального сервера. Puppet є зрілим та стабільним рішенням, яке часто використовується у великих корпоративних середовищах [17].

Chef — це потужна система, яка використовує Ruby-подібну мову для написання «рецептів» (recipes), що визначають конфігурацію систем. Chef також працює за клієнт-серверною моделлю. Його гнучкість дозволяє реалізовувати складні сценарії конфігурації, проте це вимагає глибших знань програмування.

Таблиця 2.1 – Порівняльна таблиця інструментів

Критерій	**Ansible**	**Puppet**	**Chef**
Простота використання	Висока (YAML, без агентів)	Середня (власна DSL-мова)	Низька (потрібні знання Ruby)
Швидкодія	Висока в малих/середніх масштабах	Висока при великій кількості вузлів	Середня, залежить від складності рецептів
Наявність документації	Дуже повна, легка для розуміння	Повна, технічна	Повна, орієнтована на DevOps
Активність спільноти	Дуже активна, велика підтримка	Висока, корпоративна підтримка	Висока, активна в середовищі DevOps
Архітектура	Agentless (SSH), push-модель	Client-server, pull-модель	Client-server, pull-модель
Крива навчання	Плавна	Середня	Стрімка
Інтеграція з хмарами	Відмінна підтримка AWS, Azure, GCP	Підтримка AWS, Azure через модулі	Добра підтримка, особливо AWS

З урахуванням простоти реалізації, широкої документації та сумісності з

більшістю сучасних хмарних і локальних платформ, Ansible є найбільш придатним інструментом для реалізації автоматизації в межах даної роботи. Його переваги — декларативний стиль конфігурацій, швидке налаштування, відсутність потреби в агентах і широка підтримка — забезпечують ефективну інтеграцію як із застарілими системами, так і з новими хмарними або мікросервісними компонентами.

2.3 Використання контейнерів і оркестрації (Docker, Kubernetes)

Одним із ключових напрямів модернізації інформаційних систем є впровадження технологій контейнеризації та оркестрації, які забезпечують гнучкість, масштабованість та ефективне управління програмним середовищем. Застарілі системи зазвичай тісно пов’язані з конкретною операційною системою або апаратним забезпеченням, що ускладнює їх перенесення та інтеграцію з сучасними платформами. Контейнери вирішують цю проблему, забезпечуючи стандартизований та ізольований спосіб запуску програм незалежно від середовища.

Контейнеризація — це технологія віртуалізації на рівні ОС, яка дозволяє запускати додатки у вигляді незалежних контейнерів, що містять все необхідне для їх виконання: код, бібліотеки, конфігурації. Найбільш популярною платформою для контейнеризації є Docker.

Основні переваги контейнеризації:

– ізоляція: кожен контейнер працює незалежно, що зменшує ризик конфліктів між додатками;

– переносимість: контейнер можна запускати на будь-якій платформі, де встановлено Docker — незалежно від ОС або хмари;

– швидкість розгортання: створення та запуск контейнера займає лічені секунди, що особливо корисно для CI/CD;

– економія ресурсів: контейнери легші за віртуальні машини та не вимагають повноцінної ОС для кожного екземпляра;

– сумісність із DevOps: легко інтегруються з сучасними процесами автоматизації [18].

Docker є інструментом, що дозволяє створювати, управляти та запускати контейнери. Його архітектура базується на концепції образів (images) — шаблонів, що містять інструкції для створення контейнера. Образи описуються у спеціальному файлі Dockerfile, де зазначаються базовий образ, залежності, змінні середовища, команди для запуску.

Приклад простого Dockerfile:

```
FROM python:3.10
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
CMD ["python", "main.py"]
```

Команди керування:

– docker build -t myapp . — створення образу;

– docker run -d -p 5000:5000 myapp — запуск контейнера у фоновому режимі з прокиданням порту;

– docker ps, docker stop, docker logs — моніторинг та керування контейнерами [19].

Docker дозволяє розгортати навіть складні додатки за допомогою docker-compose, де можна описати декілька сервісів (наприклад, додаток, база даних, кеш) у єдиному YAML-файлі.

Для великих систем, де кількість контейнерів обчислюється десятками або сотнями, постає потреба в автоматичному управлінні цими екземплярами — для цього використовується Kubernetes.

Kubernetes (K8s) — це система оркестрації, яка автоматизує розгортання, масштабування, оновлення та моніторинг контейнеризованих додатків. Вона дозволяє:

– масштабувати додатки вгору або вниз у відповідь на навантаження;

– балансувати навантаження між контейнерами (Pods);

– забезпечувати самовідновлення: при падінні контейнера Kubernetes автоматично його перезапускає;

– керувати конфігураціями та секретами централізовано;

– впроваджувати оновлення без простоїв за допомогою rolling updates.

У типовій структурі Kubernetes додатки розміщуються в Pods, які керуються контролерами (наприклад, Deployments), а взаємодія між ними відбувається через сервіси (Services). Всі компоненти функціонують у кластері, що складається з вузлів (Nodes), якими управляє Kubernetes Master.

Використання Docker та Kubernetes дає змогу будувати гібридні архітектури, де застарілі системи працюють паралельно з новими контейнеризованими мікросервісами. Наприклад, фронтенд може бути розгорнутий у контейнері, який взаємодіє з бекендом старої системи через REST API. Kubernetes, у свою чергу, дозволяє легко масштабувати фронтенд без впливу на основну систему, поступово переводячи функціонал до нової архітектури (рис. 2.2).

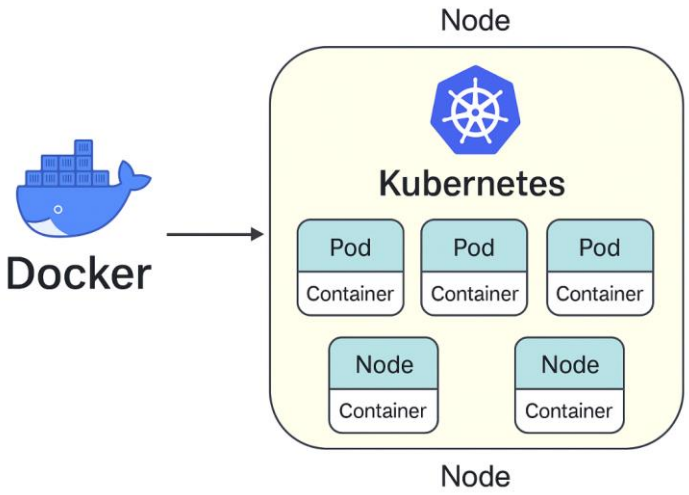


Рисунок 2.2 – Схема розгортання контейнерів за допомогою Docker та управління ними через Kubernetes-кластер

2.4 Сервіси-посередники: API Gateway, ESB, Service Mesh

У процесі побудови гібридних IT-архітектур, що поєднують старі (legacy) та нові (cloud-native, мікросервісні) компоненти, важливою складовою стають

сервіси-посередники — спеціальні інфраструктурні рішення, які забезпечують ефективну, захищену та масштабовану взаємодію між окремими сервісами, модулями або підсистемами.

До найбільш поширених моделей реалізації посередницького рівня належать API Gateway, Enterprise Service Bus (ESB) та Service Mesh. Кожен з цих підходів має власне призначення, архітектурні особливості та сферу ефективного застосування [20].

Сервіси-посередники виконують низку критично важливих функцій:

– маршрутизація запитів і трансформація даних між різними протоколами або форматами;

– забезпечення автентифікації, авторизації та шифрування комунікацій;

– балансування навантаження та керування відмовостійкістю;

– централізований моніторинг, логування та трасування запитів;

– адаптація інтерфейсів старих систем до сучасних API або мікросервісної архітектури.

Таким чином, вони виступають своєрідним «мостом» між двома світами — застарілою системою, яку не завжди можливо швидко змінити, та новими, гнучкими модулями, що потребують сучасних методів взаємодії.

Таблиця 2.2 – Порівняння API Gateway, ESB та Service Mesh

Характеристика	**API Gateway**	**ESB (Enterprise Service Bus)**	**Service Mesh**
Рівень	На межі клієнт ↔ сервіс	Центральний інтеграційний шар	Всередині мікросервісної інфраструктури

Продовження таблиці 2.2

Призначення	Прийом та обробка зовнішніх запитів до API	Інтеграція гетерогенних систем (BPM, ERP, CRM, DB)	Керування трафіком між мікросервісами
Тип взаємодії	Зовнішня	Внутрішня та зовнішня	Внутрішня (в межах кластера)
Основні функції	Аутентифікація, кешування, маршрутизація, throttle	Трансформація даних, адаптація протоколів, черги повідомлень	Безпека, трасування, retry, circuit-breaking
Приклади реалізації	Kong, AWS API Gateway, NGINX	MuleSoft, WSO2, Apache Camel	Istio, Linkerd, Consul Connect
Сумісність зі старими ІС	Висока через адаптацію REST ↔ SOAP, JSON ↔ XML	Дуже висока, підтримує legacy-протоколи	Обмежена, орієнтована на cloud-native середовища

У контексті модернізації інформаційних систем ці посередницькі сервіси дозволяють уникнути прямого переписування застарілих модулів. Наприклад:

– API Gateway може виступати фасадом для старої SOAP-системи, надаючи REST API для фронтенду або мобільного застосунку.

– ESB дозволяє інтегрувати стару систему обліку (написану на COBOL) із сучасною CRM, трансформуючи протоколи та формат даних.

– Service Mesh актуальний у випадках, коли нові мікросервіси вже розгорнуті в Kubernetes і потребують гнучкого керування трафіком, безпеки та спостереження.

Таким чином, кожен із цих підходів має власну нішу та може застосовуватись у рамках єдиної архітектури для покриття різних зон відповідальності. Їх правильна комбінація дозволяє створити масштабовану, гнучку й керовану інфраструктуру, яка здатна адаптувати старі компоненти до вимог сучасного цифрового середовища без критичних змін у вихідному коді.

3 РОЗРОБКА ІНТЕГРАЦІЙНОГО РІШЕННЯ ДЛЯ СТАРИХ ІНФОРМАЦІЙНИХ СИСТЕМ

3.1 Розробка middleware-рівня для інтеграції спадкових систем

У цьому контексті доцільно порівняти два підходи до віртуалізації – віртуальні машини та контейнери, які мають суттєві відмінності у рівні ізоляції та управлінні ресурсами. Обидва підходи дозволяють створювати логічно відокремлені середовища для розгортання програм, але реалізуються по-різному.

Віртуальні машини (VM) базуються на використанні гіпервізора – спеціалізованого програмного шару, який взаємодіє безпосередньо з апаратним забезпеченням. Завдяки цьому кожна VM отримує власну гостьову операційну систему, що забезпечує повну ізоляцію між середовищами. Адміністратор може створювати кілька незалежних віртуальних серверів, кожен із яких функціонує як повноцінна система. Такий підхід забезпечує високий рівень безпеки та сумісності з наявними компонентами спадкових ІС, але є більш ресурсозатратним у порівнянні з контейнерами [21].

На противагу, контейнери працюють безпосередньо поверх ядра операційної системи хосту, використовуючи загальну ОС для декількох ізольованих додатків. Це дозволяє запускати більше екземплярів на одному сервері за рахунок меншого споживання ресурсів, пришвидшує процес розгортання і спрощує масштабування. Контейнеризація (наприклад, через Docker) часто використовується як middleware-рішення для обгортання функціоналу спадкової системи у сучасну обгортку REST API чи сервісного адаптера.

Таким чином, вибір між VM та контейнерами залежить від технічних обмежень інтегрованої ІС, вимог до ізоляції, продуктивності та доступності ресурсів. На рисунку 3.1 подано спрощену схему архітектури розгортання сервісу за допомогою віртуальної машини.



Рисунок 3.1 – Структура взаємодії компонентів у віртуалізованому середовищі VM

Використання віртуальних машин є одним із класичних підходів до віртуалізації обчислювальних середовищ, що забезпечує стабільну ізоляцію систем і дозволяє одночасно запускати декілька операційних систем на одному фізичному сервері. Кожна віртуальна машина функціонує як незалежний об'єкт із власним ядром ОС, що дозволяє уникнути конфліктів між програмними компонентами та досягти високого рівня безпеки.

До основних переваг віртуалізованого підходу можна віднести:

– Ізоляція: кожна віртуальна машина працює у власному середовищі, тож збої або вразливості однієї системи не впливають на інші інстанції;

– Раціональне використання ресурсів: можливість запуску кількох ВМ на одному фізичному сервері дозволяє консолідувати інфраструктуру без додаткових інвестицій в обладнання;

– Гнучкість у виборі операційної системи: на одному сервері можуть одночасно працювати різні ОС (наприклад, Linux, Windows), що особливо важливо для інтеграції спадкових систем;

– Безпека: гіпервізори забезпечують логічне відокремлення машин і зменшують ризик поширення атак між системами.

Разом із тим, використання віртуальних машин має низку обмежень:

– Повільний процес ініціалізації: створення та запуск повноцінної ВМ потребує часу, що не завжди прийнятно для сценаріїв динамічного масштабування;

– Високе навантаження на ресурси: кожна ВМ містить окрему ОС, що призводить до значного споживання оперативної пам’яті, CPU і дискового простору;

– Ліцензійні витрати: кожна інстальована ОС і додатки в рамках ВМ потребують окремого ліцензування, що збільшує витрати при масштабуванні інфраструктури.

На відміну від віртуальних машин, контейнеризація пропонує більш легкий та гнучкий підхід до розгортання програмного забезпечення. Контейнери працюють поверх ядра хост-системи, використовуючи спільне середовище, що значно зменшує накладні витрати на ізоляцію. Менеджери контейнерів, такі як Docker, дозволяють швидко створювати та масштабувати ізольовані середовища для окремих компонентів системи [21].

У цьому підході контейнерна оболонка забезпечує базову ізоляцію, але всі контейнери залежать від однієї операційної системи хосту. Це дозволяє зменшити час запуску, спростити перенесення між середовищами та ефективніше використовувати ресурси. У контексті інтеграції застарілих систем, контейнеризація може застосовуватись для обгортання старих сервісів у сучасні інтерфейси – наприклад, через REST API.

Схему загальної архітектури контейнеризованого підходу наведено на рисунку 3.2.

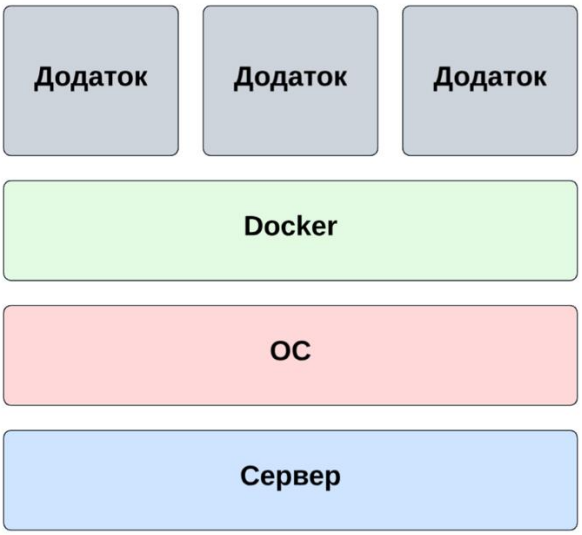


Рисунок 3.2 – Модель реалізації контейнерної ізоляції для інтеграційних сервісів

Контейнерний підхід до віртуалізації суттєво відрізняється від традиційної апаратної віртуалізації як за структурою, так і за способом організації обчислювальних процесів. Завдяки архітектурній легкості контейнери стали оптимальним рішенням для побудови middleware-рівня в інтеграційних середовищах, де критичними є швидкість розгортання, масштабованість та ефективне використання ресурсів.

До ключових переваг контейнеризації належать:

– ***Компактність:*** контейнери спільно використовують ядро операційної системи хоста, що зменшує обсяг кожного інстансу в порівнянні з віртуальними машинами, які включають повноцінну гостьову ОС. Це дозволяє розгортати значно більше контейнерів на одному сервері без перевищення ліміту ресурсів.

– ***Оперативність:*** завдяки малій вазі образів, створення, масштабування або знищення контейнерів виконується майже миттєво. Така властивість є особливо корисною для короткострокових обчислювальних завдань або систем з високою динамікою навантаження [21].

– ***Спеціалізоване середовище управління:*** платформи на кшталт Docker, RKT, Apache Mesos або Kubernetes виступають як гіпервізори

контейнерного типу, забезпечуючи централізоване управління життєвим циклом контейнерів.

– ***Незмінність середовища виконання:*** оновлення контейнерів здійснюється не шляхом внесення змін у робочі екземпляри, а шляхом генерації нових образів, які деплояться на хости. Це забезпечує відтворюваність середовища та зменшує ризик нестабільної поведінки програмного забезпечення.

Водночас, контейнеризація має й обмеження, які необхідно враховувати при проєктуванні інтеграційного рішення:

– ***Обмеження в продуктивності:*** спільне використання ядра та системних ресурсів може призводити до конфліктів між контейнерами, особливо у випадках, коли кілька компонентів потребують прямого доступу до мережі або апаратного рівня. Це створює навантаження та впливає на загальну продуктивність.

– ***Вузька сумісність:*** контейнерні образи часто жорстко прив’язані до конкретної платформи, наприклад Docker, що ускладнює їхнє перенесення в інші середовища контейнеризації або хмарні платформи без адаптації.

Узагальнюючи, контейнеризація є ефективним інструментом для реалізації middleware-рівня при інтеграції спадкових інформаційних систем. Вона забезпечує швидке розгортання, ізоляцію, масштабованість, стабільність середовища та економне використання ресурсів, що робить її оптимальною для побудови адаптивних інтеграційних платформ.

3.2 Обґрунтування вибору Kubernetes як платформи оркестрації інтеграційних сервісів

У сучасних високонавантажених IT-системах особливе значення має стабільність і передбачуваність процесів розгортання програмних компонентів. Збої на рівні окремих вузлів кластера або відмова сервера не повинні призводити до тривалого простою сервісів. Для вирішення цих завдань використовується

контейнерний підхід до побудови сервісів, що дозволяє швидко масштабувати інфраструктуру та оперативно відновлювати її працездатність у разі збою [22].

Одним із ключових інструментів middleware-рівня, що реалізує ці можливості, є Kubernetes – система автоматичного управління життєвим циклом контейнеризованих додатків. Вона виконує функції розгортання, масштабування, балансування навантаження, само-відновлення та централізованого керування конфігурацією сервісів.

Використання Kubernetes дозволяє підприємствам розгортати відмовостійкі, масштабовані додатки, які адаптуються до змін у навантаженні, кількості користувачів або інфраструктурних ресурсів. Це є особливо важливим при інтеграції спадкових ІС, коли нові сервіси обгортають старі API або виступають як адаптери між різними компонентами.

До основних можливостей Kubernetes належать:

- балансування навантаження між екземплярами додатків (pod);
- динамічне масштабування сервісів залежно від навантаження;
- автоматичне відновлення при збої компонентів;
- оркестрація зберігання даних через абстракції `Volume`;
- безперервне оновлення застосунків через механізм `Deployment`;
- гнучке управління конфігураціями і параметрами середовища.

У Kubernetes реалізовано набір абстракцій, які накладаються на фізичну або віртуальну інфраструктуру і дозволяють працювати на логічному рівні:

- Pod – базова одиниця розгортання, яка містить один або кілька пов’язаних контейнерів. Кожен pod слугує для запуску окремого екземпляра застосунку. Для масштабування кількості інстансів додаються нові pod.
- Service – абстракція, що забезпечує постійний доступ до pod'ів, незалежно від їх IP-адрес. Завдяки цьому реалізується зовнішня маршрутизація, балансування навантаження та взаємодія між мікросервісами.
- Volume – механізм збереження даних між перезапусками pod. Він дозволяє спільно використовувати сховище між кількома контейнерами та зберігати стан додатку.

- Deployment – об’єкт керування життєвим циклом pod'ів. Забезпечує автоматичне оновлення додатку, контроль над кількістю екземплярів, відкат до попередніх версій тощо.

Інтеграція Kubernetes до архітектури підприємства дозволяє ефективно використовувати доступні ресурси, зменшує залежність від фізичного обладнання та створює адаптивну платформу для побудови інтеграційних рішень. Це особливо актуально для малих та середніх компаній, які прагнуть мінімізувати витрати на підтримку власних центрів обробки даних, зберігаючи при цьому високий рівень доступності та керованості.

3.3 Впровадження внутрішнього веб-сервісу в архітектуру інтеграції

У межах формування сервісно-орієнтованої IT-інфраструктури підприємства одним із ключових компонентів виступає веб-сервіс, що забезпечує зовнішню присутність компанії, надає доступ до клієнтських функцій або взаємодіє з іншими інформаційними системами. У цьому контексті доцільно продемонструвати процес розгортання веб-сайту на прикладі популярної платформи WordPress, з використанням контейнеризованого підходу в Kubernetes-середовищі [22].

У лістингу наведено конфігурацію сервісу керування базою даних MySQL, що є необхідною частиною веб-додатку. Для забезпечення збереження даних навіть у разі видалення або збою контейнера, у конфігурації (рядки 14–26) створюється персистентне сховище обсягом 20 ГБ, яке буде підключене як зовнішній том (volume). Далі, у рядках 28–65, представлено опис `Deployment`-ресурсу, в якому вказано:

- Docker-образ бази даних (рядки 48–49);
- змінні середовища, зокрема ім’я користувача, пароль, назва БД (рядки 50–55);
- підключення до тома збереження (рядки 59–65), що гарантує сталий доступ до даних.

У лістингу наведено конфігурацію самого веб-додатку WordPress. Аналогічно, для забезпечення збереження користувацького контенту та налаштувань, у рядках 15–27 визначено volume на 20 ГБ. У розділі `Deployment` (рядки 28–67) зазначено:

- образ WordPress (рядки 48–49),
- змінні оточення, зокрема параметри підключення до бази даних (рядки 50–57),
- опис збереження даних у томі (рядки 61–67).

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
  selector:
    app: wordpress
    tier: frontend
  type: LoadBalancer
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: wp-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: frontend
    spec:
      containers:
        - image: wordpress:4.8-apache
          name: wordpress
          env:
            - name: WORDPRESS_DB_HOST
              value: wordpress-mysql
            - name: WORDPRESS_DB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql-pass
                  key: password
          ports:
            - containerPort: 80
              name: wordpress
          volumeMounts:
            - name: wordpress-persistent-storage
```

```
        mountPath: /var/www/html
    volumes:
      - name: wordpress-persistent-storage
        persistentVolumeClaim:
          claimName: wp-pv-claim
```

Після підготовки файлів конфігурації необхідно застосувати їх до Kubernetes-кластера за допомогою команди:

kubectl apply -k ./

У результаті цього процесу буде розгорнуто два основні компоненти – сервіс бази даних та веб-інтерфейс WordPress, які функціонують у контейнерах і забезпечують відмовостійке зберігання даних.

На рисунку 3.3 зображено результат перевірки працездатності сервісів у кластері після застосування конфігураційних файлів, зокрема їхній стан, доступність та IP-адреси.

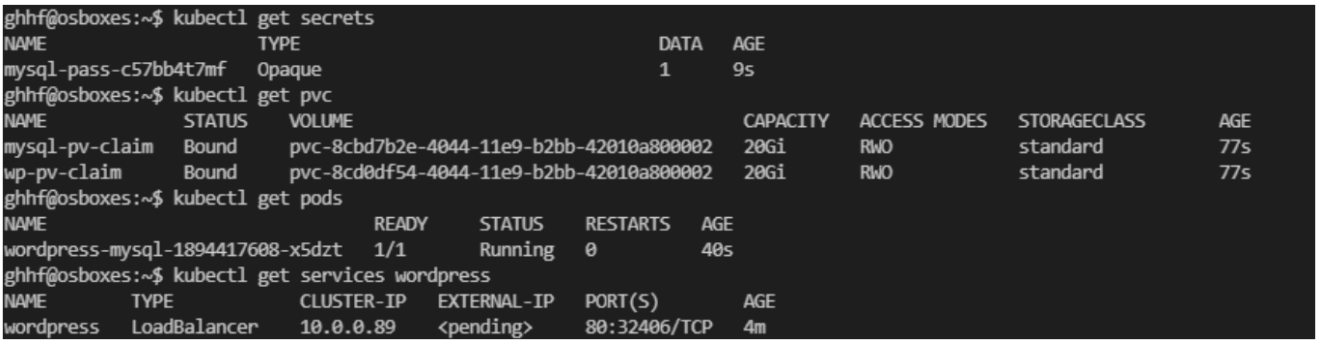


Рисунок 3.3 – Тестування доступності WordPress-сервісу через LoadBalancer

Розгортання сервісів у середовищі Kubernetes відкриває широкі можливості для забезпечення високої доступності, горизонтального масштабування та автоматичного відновлення сервісів у разі збоїв. Завдяки централізованому управлінню, система дозволяє ефективно нівелювати ключові недоліки контейнеризації, зокрема обмежену стійкість до відмов та необхідність ручного втручання при масштабуванні.

Крім того, Kubernetes спрощує створення кластерної інфраструктури через декларативне налаштування компонентів. Однак, попри значні переваги, технологія вимагає поглиблених знань від системного адміністратора, зокрема

щодо побудови кластерної топології, управління мережевими політиками, секретами, ресурсами та моніторингом. Інтерфейс налаштування не є інтуїтивно зрозумілим для новачків, що може створювати певний бар'єр на етапі впровадження в малих організаціях або проєктах із обмеженими IT-ресурсами.

3.4 Розробка інтеграційних модулів за допомогою Ansible

Серед розглянутих інструментів управління IT-інфраструктурою особливу увагу варто приділити мові опису Ansible DSL, яка базується на декларативному форматі YAML. Простота структури, читабельність та людиноорієнтований синтаксис цієї мови роблять її надзвичайно зручною для написання інструкцій з розгортання інфраструктури та автоматизації рутинних процесів. Враховуючи переваги даного підходу, його використання є доцільним при створенні засобу управління сервісно-орієнтованою IT-архітектурою.

Інструмент Ansible завдяки своїй кросплатформенності, гнучкості та модульному підходу може виступати базовою платформою для розгортання інтеграційних сервісів, оновлення конфігурацій, проведення тестування та імплементації змін в інфраструктурі без ручного втручання. Така синергія декларативного опису та модульного виконання дозволяє динамічно адаптувати середовище до змін користувацьких вимог або до оновлених версій служб.

У рамках цього підходу доцільно реалізувати власний модуль Ansible, який використовуватиме YAML-файли як інтерфейс взаємодії, дотримуючись стандартів та архітектурних принципів системи Ansible. Базовою мовою для розробки модуля обрано Python 3, оскільки:

– ця мова є рідною для Ansible та інтегрується в його плагінову архітектуру без додаткових залежностей;

– Python має вбудовану підтримку для більшості UNIX-подібних систем і входить до складу стандартних дистрибутивів Linux;

– стандартизація за допомогою PEP (Python Enhancement Proposal) забезпечує читабельність та уніфікований стиль написання коду, що полегшує підтримку;

– відкритий вихідний код та активна спільнота дозволяють швидко впроваджувати зміни та адаптувати рішення під індивідуальні потреби.

Таким чином, використання Ansible у поєднанні з Python забезпечує ефективну та масштабовану платформу для побудови гнучкого інструменту розгортання IT-інфраструктури підприємства.

Функціональна суть і призначення модулів у системі Ansible

У системі Ansible модуль є основною одиницею виконання завдань. Модуль Ansible – це багаторазовий, ізольований скрипт, який виконується локально або на віддаленій цільовій машині через інфраструктуру Ansible. Модулі дозволяють здійснювати взаємодію з різноманітними об'єктами – від файлів, конфігурацій системи та пакетів до зовнішніх API та хмарних платформ.

Типовими прикладами використання модулів є:

– оновлення конфігураційних файлів на сервері;
– зміна облікових даних у базі даних;
– запуск віртуального екземпляра у хмарі (наприклад, AWS, Azure);
– створення мережевих правил тощо.

Модулі можуть бути викликані через команду `ansible`, використовуватись у playbook або взаємодіяти з Ansible API у більш складних автоматизованих рішеннях. Кожен модуль має формалізований інтерфейс, приймає аргументи у вигляді словника (як правило, у форматі YAML), а результат його виконання повертається у форматі JSON, який обробляється ядром Ansible. Цей результат містить інформацію про статус виконання, повідомлення, змінені об’єкти, а також помилки у разі їх виникнення [23-25].

У випадках, коли вбудованих модулів недостатньо для реалізації специфічної логіки, Ansible дозволяє створювати власні користувацькі модулі, написані переважно мовою Python. Це розширює функціональні можливості платформи та забезпечує повну адаптацію до потреб конкретного середовища.

На рисунку 3.4 наведено схематичне зображення процесу взаємодії між playbook, модулями та цільовими хостами. Playbook представляє собою набір завдань, які послідовно виконуються на вказаних вузлах. У процесі виконання Ansible мапує кожне завдання до відповідного модуля, передає йому аргументи та виконує дії на віддалених системах або локально, залежно від типу задачі.

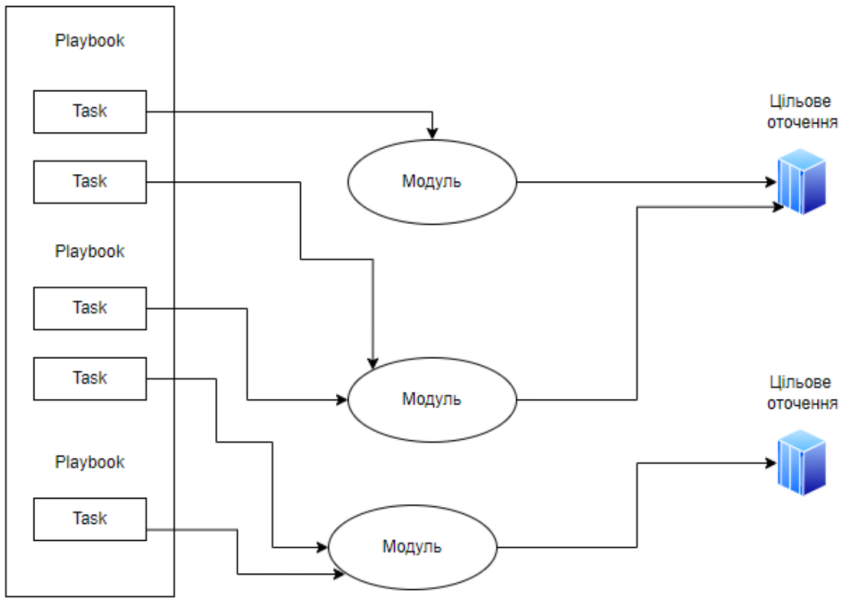


Рисунок 3.4 – Схема взаємодії модулів Ansible з playbook та цільовим середовищем

Як приклад розглянуто встановлення пакета openssh-server із використанням пакетного менеджера APT та за допомогою модуля yum у Ansible, що наведено на рисунку 3.5.

```
---
- name: Install ssh server
  apt:
    name: openssh-server
    state: latest
...
```

Рисунок 3.5 – Автоматизоване встановлення пакета через модуль apt у Ansible

Ключові параметри модуля apt, зокрема name та state, обробляються ядром Ansible і транслюються у відповідні Bash-команди, які виконуються на цільовому

віддаленому хості [24]. Після завершення виконання сформованої команди модуль отримує стандартний вивід, аналізує його та записує відповідну інформацію про результат у лог-файл системи. Механізм цієї взаємодії продемонстровано на рисунку 3.6.

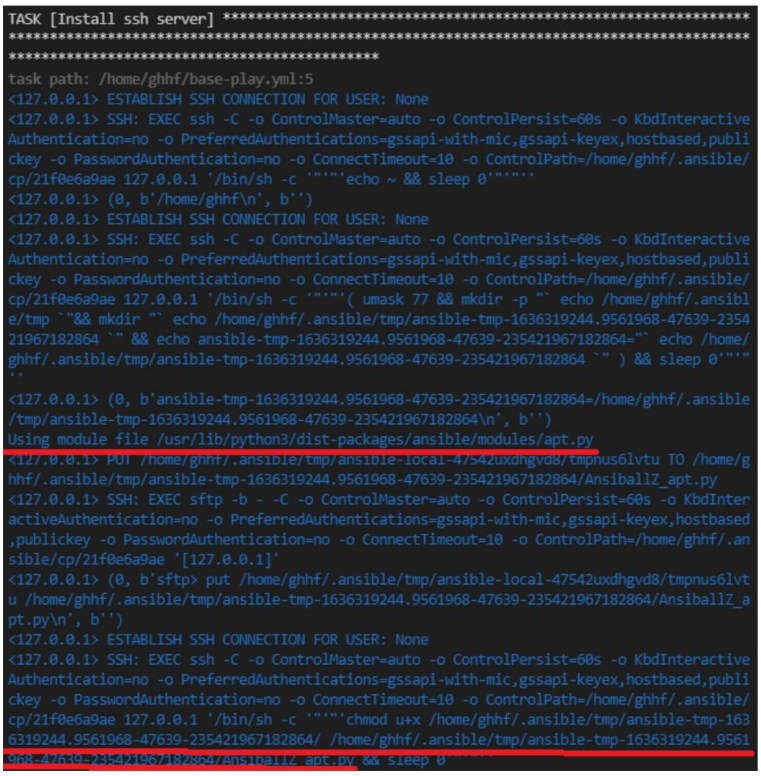


Рисунок 3.6 – Журнал виконання команди встановлення пакета openssh-server у Ansible

Для реалізації користувацького модуля було створено проєкт Ansible, структура директорій якого відображає організацію компонентів модуля. Ця структура представлена на рисунку 3.7.

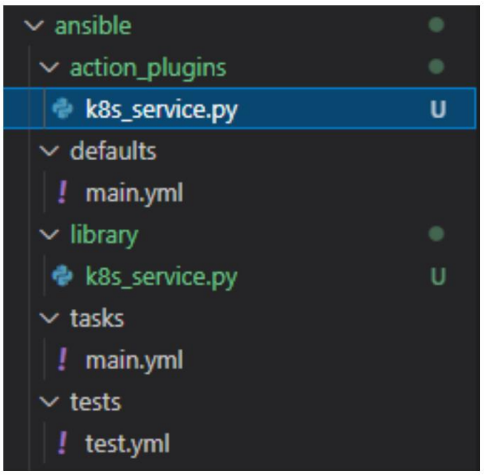


Рисунок 3.7 – Логічна структура директорій проєкту з власним модулем Ansible

Конфігурація користувацького модуля в Ansible реалізується в межах директорій action_plugins та library, які містять відповідні скрипти та логіку виконання завдань. Решта директорій проєкту – такі як roles, tasks, handlers, – відповідають за визначення структури ролей та сценаріїв (playbook), необхідних для автоматизованого розгортання сервісно-орієнтованої IT-інфраструктури.

Основна логіка функціонування модуля реалізована у вигляді набору класів, описаних у файлі k8s_service. Структуру цих класів, їхнє призначення та функціональне навантаження наведено у таблиці 3.1.

Таблиця 3.1

Склад і логіка класів інструменту k8s_service для Ansible

	Клас	Призначення
1	Service	Базовий клас сервісу, який є батьківським для всіх сервісів.
2	DocumentationTrackingService	Клас для розгортання системи документування та трекінгу на основі openproject.
3	InfrastructureDebugger	Клас для підготовки повідомлень про стан інфраструктури.
4	MailService	Клас для розгортання поштового веб-сервера на основі postfix, dovecot.

Продовження таблиці 3.1

5	WebSiteService	Клас для розгортання веб-сайту на основі стеку LAMP.
6	MoodleService	Клас для розгортання системи дистанційного навчання на основі Moodle.
7	DriveService	Клас для розгортання системи хмарного диску на основі Nextcloud.
8	UIController	Клас-контролер для керування роботою веб-застосунку інструменту.
9	UiModel	Модель, яка передається до UIController для взаємодії з інтерфейсом веб-застосунку.
10	DatabaseConnector	Клас для підключення до бази даних і виконання запитів.
11	ServicePatternWorker	Клас для створення модулів інструменту.
12	KubernetesWorker	Клас для взаємодії з кластером Kubernetes: створення, редагування, видалення ресурсів.

3.5 Приклад роботи інтеграційного рішення на основі сервісно-орієнтованої архітектури

У межах розділу 3.3 розглянуто підхід до розгортання веб-сервісів шляхом формування конфігураційних файлів Kubernetes. Незважаючи на потужність і гнучкість цієї технології, її практичне застосування потребує глибоких знань у галузі адміністрування кластерів та значного часу для вивчення специфіки синтаксису YAML, архітектури Kubernetes та засобів оркестрації [25]. Саме для спрощення цього процесу було розроблено інструмент, що автоматизує етапи конфігурування та розгортання сервісно-орієнтованої IT-інфраструктури.

Створене рішення побудоване за модульним принципом і включає три основні компоненти:

- ***Ansible-модуль*** – забезпечує автоматизоване розгортання інфраструктурних сервісів через мову конфігурації Ansible. Крім того, надає підтримку розширення функціоналу шляхом додавання власних модулів;
- ***API-модуль*** – реалізує можливість зовнішньої інтеграції з інструментом через програмний інтерфейс, відкриваючи можливості взаємодії з іншими системами;
- ***WEB-модуль*** – забезпечує зручний веб-інтерфейс для відстеження стану та керування розгорнутими сервісами. Детальна інструкція з його використання наведена у розділі 4.4.

Для використання Ansible-модуля користувачеві необхідно сформувати відповідний playbook із використанням ключових параметрів, перелік яких наведено в таблиці 2.1. Далі файл модуля додається до робочої директорії разом із playbook, після чого інструмент отримує необхідні права для виконання операцій на сервері. Описаний у сценарії сервіс автоматично транслюється у Kubernetes-конфігурацію, яка застосовується до цільового кластера.

Загальна логіка взаємодії між компонентами інструмента та послідовність виконання етапів розгортання подана на рисунку 3.8.



Рисунок 3.8 – Структурна модель роботи Ansible, API та WEB-модулів інструмента

З метою демонстрації функціональних можливостей інструмента у роботі було представлено приклад конфігураційного файлу сервісу, оформленого у форматі YAML відповідно до специфікації опису інфраструктури як коду [26]. Вміст цього конфігураційного файлу наведено на рисунку 3.9.

```
- name: Init http service
  k8s_service:
    name: Web_site
    description: this service will be configured as a website and will be used for showing company information
    state: run
    module: web-server
    module_params: []
    ports: 33820
    ha: True
    dataset:
      - /var/www/html
```

Рисунок 3.9 – Структура опису сервісу у форматі YAML для Kubernetes

Після успішного виконання playbook (див. рисунок 3.10), відбувається автоматичне формування конфігураційних об’єктів у кластері Kubernetes, зокрема створення сервісу, ресурсу Deployment та відповідного Pod. Результат застосування конфігурації відображено на рисунку 3.11.

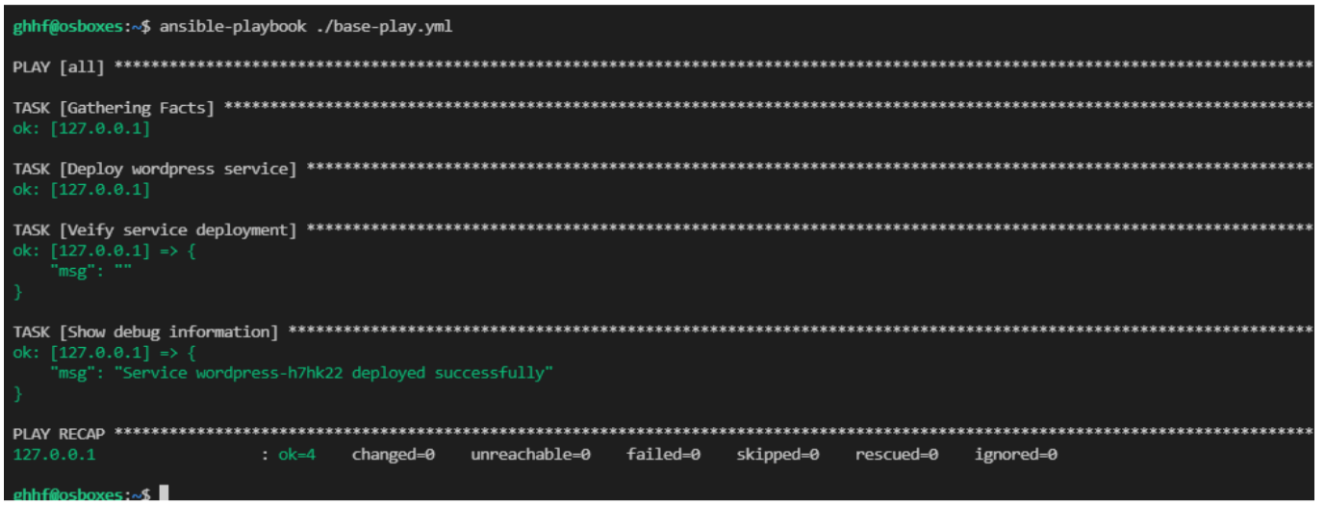


Рисунок 3.10 – Виконання сценарію Ansible для створення сервісу в кластері Kubernetes



Рисунок 3.11 – Стан розгорнутого веб-сервісу у кластерному середовищі

Під час взаємодії з API-модулем інструмента, користувач отримує можливість створення сервісу шляхом виконання REST-запиту до відповідного ендпоінта. Структура та параметри цього запиту детально описані у таблиці 3.2.

Таблиця 3.2

REST-ендпоінти для керування сервісами в інструменті

Шлях запиту (`Endpoint`)	HTTP-метод	Призначення запиту
/api/service/{id}	POST	Ініціалізація нового сервісу

Продовження таблиці 3.2

/api/service/{id}	PUT	Оновлення конфігурації вже створеного сервісу
/api/services	GET	Отримання переліку всіх зареєстрованих сервісів
/api/service/{id}	DELETE	Видалення конкретного сервісу за його ідентифікатором
/api/service/{id}	GET	Перегляд детальної інформації щодо конкретного сервісу

Запропонована структура REST-ендпоінтів забезпечує повний цикл взаємодії з сервісами – від їх створення та налаштування до моніторингу й видалення. Такий підхід дозволяє гнучко керувати елементами IT-інфраструктури в автоматизованому режимі, не потребуючи ручного втручання в конфігураційні файли. Впровадження API-модуля в інструмент розгортання значно спрощує інтеграцію з зовнішніми системами та підтримує сучасні принципи DevOps, забезпечуючи ефективність, масштабованість і прозорість процесів управління сервісами [26].

4 ТЕСТУВАННЯ ТА ОЦІНКА ІНТЕГРАЦІЙНОГО РІШЕННЯ

4.1 Методологія тестування сумісності при взаємодії застарілих і сучасних систем

У процесі впровадження інструментів для розгортання IT-інфраструктури нерідко виникають типові проблеми: порушення сумісності, збої під час запуску, нестабільна робота модулів або навіть втрата критичних даних. Щоб знизити ймовірність подібних ситуацій, сучасна розробка передбачає вбудоване тестування на всіх етапах життєвого циклу програмного забезпечення – від проєктування до виробничого розгортання.

Найбільш поширеною помилкою команд розробки є відсутність структурованого підходу до тестування: тести проводяться точково, без єдиної стратегії або системної координації. У результаті – тестування не забезпечує очікуваного рівня контролю якості, а знайдені дефекти не мають належної відстежуваності та документації.

Основне призначення тестування полягає у своєчасному виявленні помилок, перевірці відповідності функціональності заявленим вимогам, а також формуванні зворотного зв’язку для подальших ітерацій розробки. Якість і ефективність цього процесу напряму залежать від правильно обраної методології тестування.

Вибір методології обумовлюється специфікою бізнес-потреб, масштабом інструменту та критичністю його функцій. З-поміж поширених підходів можна виокремити три ключові стратегії [27-29]:

– ***Тестування на основі вимог*** – базується на формалізованій специфікації, що дозволяє будувати точні тест-кейси відповідно до функціональних очікувань.

– ***Тестування на основі ризиків*** – фокусується на критичних компонентах, які можуть спричинити найбільші наслідки у разі збою. На основі ймовірності відмови та потенційного впливу формується пріоритет перевірок.

– ***Експертне (інтуїтивне) тестування*** – передбачає використання досвіду тестувальника без жорсткої прив’язки до документації. Хоча цей метод забезпечує швидке виявлення проблем, він є менш формалізованим і може залежати від суб’єктивного фактору.

У контексті розгортання IT-інфраструктури найбільш релевантним є саме ризик-орієнтований підхід. Помилки в будь-якому компоненті архітектури можуть призвести до непередбачуваних наслідків: від зниження доступності до повної зупинки роботи системи. Тому аналіз ризиків та перевірка ключових точок відмови мають бути в основі тестової стратегії [29].

4.2 Етапи розробки, впровадження та інтеграції архітектури

Для повноцінної перевірки працездатності інструмента розгортання IT-інфраструктури необхідно враховувати ключові етапи його життєвого циклу. У практиці розробки програмного забезпечення зазвичай виокремлюють такі стадії:

- аналіз та формування вимог;
- архітектурне і технічне проєктування;
- безпосередня реалізація (розробка);
- впровадження (deployment);
- експлуатація та супровід [30].

Кожен із цих етапів повинен супроводжуватися відповідними видами тестування, що дозволяє не лише виявляти помилки на ранніх стадіях, а й запобігати появі критичних дефектів у продуктивному середовищі. Особливу увагу у випадку інструмента для розгортання IT-інфраструктури слід приділити таким стадіям, як реалізація, впровадження та експлуатація – саме вони є найбільш вразливими з точки зору впливу на стабільність та безперервність роботи системи.

На етапі реалізації важливо забезпечити якісну перевірку логіки окремих компонентів (unit-тестування), перевірити взаємодію з API та правильно обробити параметри вхідних конфігурацій.

На стадії впровадження ключовим є тестування інфраструктури: чи коректно генеруються Kubernetes-конфігурації, чи застосовуються вони у кластері, та чи не призводить конфлікт версій до помилок під час розгортання. Нарешті, під час експлуатації необхідне постійне спостереження за працездатністю сервісів, автоматичне сповіщення про відмови та перевірка сценаріїв масштабування або аварійного відновлення.

Стадія реалізації передбачає створення мінімально життєздатної версії інструмента (MVP) з фокусом на основну логіку його роботи. Цей етап має низку характерних ознак:

- зміни в програмному коді не несуть критичної загрози для продуктивного середовища;
- основна увага приділяється розробці та написанню вихідного коду;
- часові обмеження на тестування менш жорсткі, що дозволяє провести глибоку перевірку логіки;
- застосовується unit-тестування для ізольованої перевірки окремих функціональних компонентів.

Використання unit-тестів надає можливість формалізувати внутрішню архітектуру програми, відслідковувати наслідки внесення змін до коду та своєчасно виявляти відхилення в логіці роботи окремих модулів. Це особливо важливо в умовах поступового розвитку інструмента, коли модифікації можуть впливати на інші частини системи [31].

У межах тестування розробленого інструмента було створено набір unit-тестів, які охоплюють ключові класи та методи, забезпечуючи перевірку їхньої коректної роботи. Один із прикладів – клас `InfrastructureDebugger`, який відповідає за формування структурованого виводу даних про поточний стан IT-інфраструктури. Фрагмент коду, що демонструє його призначення та логіку, наведено на рисунку 4.1.

```python
import os
import kubeapi

class InfrastructureDebugger:
  #empty constructor
  def __init__(self):
    pass

  def showPodInfoLog(self, pod):
    return kubeapi.showPodLog(pod)

  def showDeploymentStatus(self, deployment):
    status = "-------Deployment: {}-------\n".format(deployment)
    status += kubeapi.showActiveDeploymentPodsCount(deployment)
    return status
```

Рисунок 4.1 – Реалізація класу InfrastructureDebugger для виводу стану інфраструктури

Для перевірки правильності функціонування класу InfrastructureDebugger було реалізовано unit-тест, приклад якого наведено на рисунку 4.2. Основна мета даного тесту – перевірити, чи відповідає фактична логіка виконання методу очікуваній, а також переконатися, що результати, які повертаються, узгоджуються з передбаченими в специфікації.

У процесі запуску тесту викликається метод setUp(), який автоматично виконується перед кожним тестовим методом. На цьому етапі створюється екземпляр класу InfrastructureDebugger, який зберігається у вигляді атрибута тестового класу. Це дозволяє забезпечити чисте середовище для кожної перевірки та унеможливити вплив попередніх викликів на результати тестування [32].

```python
import unittest
from InfrastructureDebugger import InfrastructureDebugger
#Test cases to test Calulator methods
#You always create  a child class derived from unittest.TestCase
class TestInfrastructureDebugger(unittest.TestCase):
  #setUp method is overridden from the parent class TestCase
  def setUp(self):
    self.debugger = InfrastructureDebugger()

  def test_showPodInfoLog(self):
    self.assertEqual(self.debugger.showPodLog('empty_pod'), '')

  def test_showDeploymentStatus(self):
    self.assertEqual(self.debugger.showDeploymentStatus('test_deployment'),
      '-------Deployment: test_deployment-------\n')

if __name__ == "__main__":
  unittest.main()
```

Рисунок 4.2 – Реалізація unit-тестування класу InfrastructureDebugger із використанням unittest

Усі методи тестування в модулі unittest дотримуються загальноприйнятого

правила – їх назви мають починатися з префіксу test_, що дозволяє тестовому фреймворку автоматично розпізнавати та виконувати відповідні функції [30]. У кожному з таких методів застосовується вбудована функція assertEqual(), яка виконує порівняння фактичного результату роботи коду з очікуваним. Якщо значення збігаються – тест вважається успішним; у протилежному випадку система сигналізує про збої у логіці або непередбачені зміни у поведінці програмного компоненту.

Перевага застосування unit-тестів саме на етапі розробки полягає в низькій вартості усунення помилок. Виявлення дефектів до моменту розгортання інструменту в продуктивному середовищі дозволяє швидко локалізувати проблему та внести необхідні зміни, уникнувши ризиків, пов’язаних із порушенням роботи системи на стороні кінцевого користувача.

Етап інтеграції інструменту в цільове середовище

Етап інтеграції охоплює перенесення розробленого інструмента до середовища користувача з метою його повноцінного функціонування в рамках реальної IT-інфраструктури. На цьому етапі система набуває прикладного значення, а будь-які зміни у її конфігурації чи логіці можуть мати суттєвий вплив на роботу бізнес-процесів. Характерними особливостями цього етапу є [33]:

- підвищена чутливість до помилок і потреба в детальному тестуванні взаємодії між компонентами;
- акцент на написанні та перевірці скриптів автоматичного розгортання;
- наявність обмежень за часом, але не в критичному режимі;
- усунення виявлених дефектів має найвищий пріоритет;
- доцільність застосування інтеграційного тестування як основної методології.

Інтеграційне тестування (також відоме як функціональне на рівні взаємодії компонентів) дозволяє виявити помилки, пов’язані з некоректною інтеграцією модулів, непередбаченими залежностями між підсистемами або неправильними взаємодіями з зовнішніми середовищами (операційною системою, мережею, API, хмарною платформою тощо).

Залежно від архітектурних вимог та ступеня готовності системи, інтеграційне тестування може реалізовуватись за кількома підходами:

– ***Bottom-Up Integration*** – тестування починається з найнижчого рівня модулів, які поєднуються у більші блоки. Цей підхід ефективний за умови готовності більшості базових компонентів.

– ***Top-Down Integration*** – спершу тестуються модулі верхнього рівня, тоді як компоненти нижчих рівнів імітуються тимчасовими «заглушками». У міру завершення розробки ці заглушки замінюються на реальні частини системи.

– ***Big Bang Integration*** – усі модулі інтегруються одночасно, і відбувається тестування повної системи. Хоча цей метод дозволяє швидко перейти до перевірки готового рішення, він є ризикованим, оскільки будь-яка помилка у тест-кейсах може ускладнити локалізацію проблем.

У Python інтеграційні тести також можуть реалізовуватись із використанням фреймворку `unittest`, проте, на відміну від unit-тестування, фокус зміщується на тестування процесів та залежностей між модулями, а не на окремі функції або класи. Такий підхід дозволяє перевірити реальні сценарії використання інструмента в умовах, максимально наближених до бойових.

Слід зазначити, що виправлення помилок на цьому етапі є більш витратним, ніж у фазі реалізації, оскільки кожне виправлення вимагає ретестування усієї взаємодії між компонентами та потенційно впливає на стабільність інфраструктури в цілому.

Етап супроводу та функціонального моніторингу інструмента

Фаза активного використання інструмента в реальному середовищі передбачає його безпосередню експлуатацію кінцевими користувачами та орієнтована на стабільну підтримку розгорнутого рішення. У межах цього етапу зміни до коду стають критично значущими, а час реагування на інциденти має вирішальне значення [32-34].

Ключовими ознаками цієї фази є:

- внесення змін до програмного забезпечення відбувається лише за наявності обґрунтованої необхідності, оскільки навіть незначне порушення логіки може спричинити зупинку сервісів;
- основна увага приділяється підтримці та обслуговуванню існуючого коду та інфраструктурних компонентів;
- будь-яке виявлене порушення роботи системи вимагає оперативного усунення з мінімальними затримками.

У межах експлуатації важливими завданнями є:

- забезпечення стабільної роботи сервісів;
- підтримка документації для адміністраторів та користувачів;
- розробка процедур відновлення у разі часткових або повних відмов.

Для досягнення цілей на цьому етапі доцільним є використання таких підходів тестування:

- візуальне тестування (ручна перевірка інтерфейсів, логіки запуску);
- аналіз лог-файлів (діагностика на основі журналів подій, виявлення аномалій);
- інфраструктурне тестування розгорнутої системи.

Варіанти тестування IT-інфраструктури у продуктивному середовищі поділяються на:

- перевірку окремих компонентів (наприклад, лише бази даних або лише веб-сервера);
- тестування на рівні модулів, що складаються з взаємопов’язаних елементів (наприклад, кластер MySQL + backend-сервіс);
- повну перевірку системи після повного розгортання, що передбачає наскрізне тестування.

Найбільш надійним підходом із погляду контрольованості є модульне розбиття інфраструктури з подальшою верифікацією кожного блоку незалежно. Така стратегія дозволяє швидше локалізувати джерело проблеми та перевіряти тільки змінені компоненти без необхідності перевірки всієї системи [34].

На рисунку 4.3 представлено приклад застосування підходу модульного тестування, де за допомогою директиви `block` виконано розділення на функціональні частини. Зокрема, окремо протестовані компоненти бази даних та веб-сервера, які діють автономно та можуть розгортатися незалежно один від одного.

```
---
- hosts: playground_cluster_group
  tasks:
    - name: apache server installation
      block:
        - name: apache server should be latest version
          yum:
            name: httpd
            state: latest

        - name: apache server restart
          service:
            name: httpd
            state: started

    - name: apache server verification
      script: verify.sh --apache

    - name: mysql server installation
      block:
        - name: install mysql independent
          yum:
            name: mariadb
            state: latest

    - name: mysql server verification
      script: verify.sh --mysql
```

Рисунок 4.3 – Структурований приклад перевірки компонентів інфраструктури з використанням Ansible

Такий підхід забезпечує реалізацію концепції безперервного розгортання, оскільки кожен окремий компонент інфраструктури перевіряється одразу після його створення. У випадку виявлення помилки система ініціює повторне розгортання лише конкретного модуля, а не всієї IT-інфраструктури. Це не лише оптимізує час впровадження, а й дозволяє виконувати цільове модульне налаштування за потреби [34].

4.3 Керівництво з розробки адаптерів і сервісних компонентів

Щоб реалізувати модуль для системи розгортання сервісно-орієнтованої IT-інфраструктури, необхідно підготувати середовище на базі віртуальної машини з операційною системою Ubuntu 20.04. Першим етапом є інсталяція інструменту Ansible. Після цього слід розгорнути кластер Kubernetes за допомогою команди [35]:

sudo snap install microk8s –classic

По завершенню встановлення необхідно надати поточному користувачу права доступу до групи адміністраторів Kubernetes:

sudo usermod -a -G microk8s $USER

sudo chown -f -R $USER ~/.kube

su - $USER

Після внесення змін до прав доступу варто виконати перевірку готовності кластеру до роботи, використовуючи такі команди:

microk8s status --wait-ready

microk8s dashboard-proxy

Остання команда дозволяє отримати токен доступу та посилання на веб-інтерфейс адміністратора кластеру, що відображається у браузері (див. рисунок 4.4).

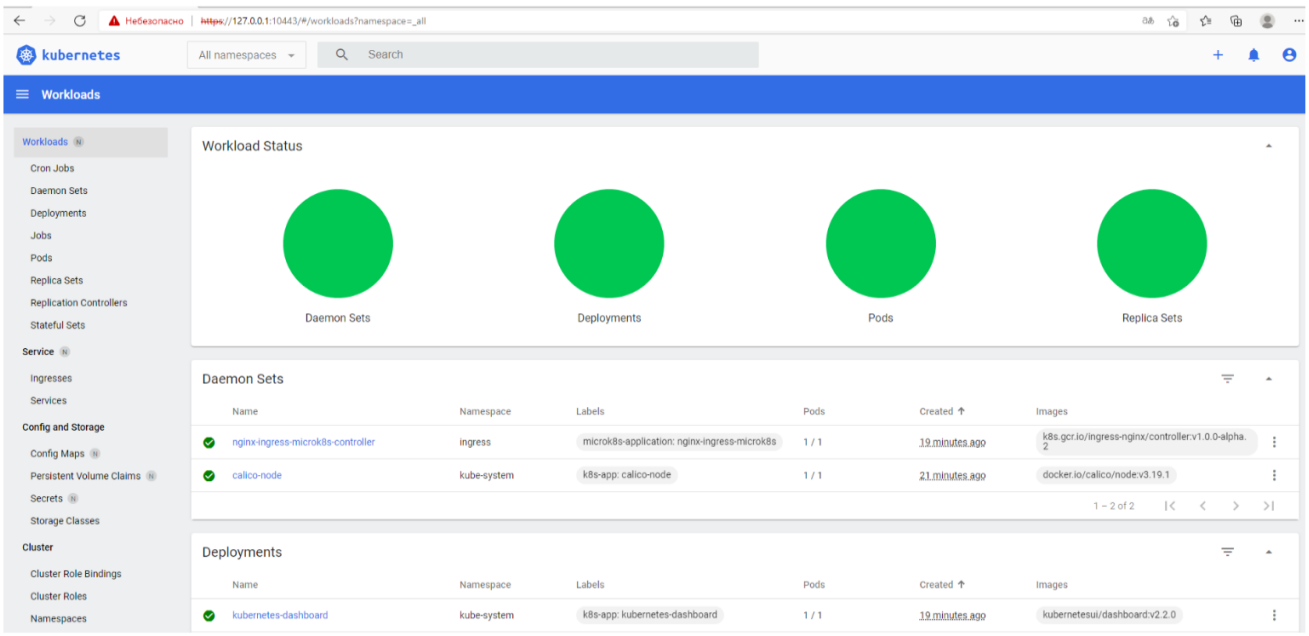


Рисунок 4.4 – Графічна панель адміністрування Kubernetes-кластеру

Щоб розпочати створення користувацького модуля для інструменту розгортання, необхідно в каталозі ansible/library створити відповідний файл модуля. Після цього слід імпортувати необхідний функціональний клас Service, що забезпечує доступ до базової логіки обробки інфраструктурних об'єктів. Після виконання цих підготовчих кроків можна переходити безпосередньо до програмування логіки модуля, враховуючи специфіку задач, які він має реалізовувати [36].

4.4 Рекомендації для адміністратора щодо підтримки інтеграційної інфраструктури

Для формування нового шаблону (патерну) сервісу необхідно перейти на головну сторінку інструмента та перейти до відповідного розділу створення патерну. У процесі налаштування користувач обирає ключові параметри, що визначають логіку розгортання.

На рисунку 4.4 представлено приклад заповнення форми для створення шаблону сервісу. Особливу увагу слід звернути на рядки 3–4:

- рядок 3 – містить ключове слово, яке буде використовуватись у YAML-описі інфраструктури для виклику конкретної дії;
- рядок 4 – визначає функцію, яка реалізує логіку відповідного елемента, написану мовою Python (див. детальніше у підрозділі 4.3).

Таким чином, за допомогою зіставлення (мапінгу) ключових слів з відповідними методами модулів, система отримує можливість інтерпретувати YAML-опис інфраструктури та коректно виконувати автоматизоване розгортання.

Щоб виконати основні дії з сервісом, користувачеві необхідно перейти на головну сторінку інструменту та відкрити розділ для створення або редагування сервісу. У цьому інтерфейсі доступні параметри конфігурації, що визначають логіку, структуру та поведінку майбутнього сервісу [38].

На рисунку 4.4 представлено приклад вибору налаштувань при конфігуруванні сервісу. Цей етап є завершальним у формуванні повноцінного шаблону, після чого обраний сервіс стає доступним для подальшого розгортання в інфраструктурі відповідно до визначеного YAML-опису.

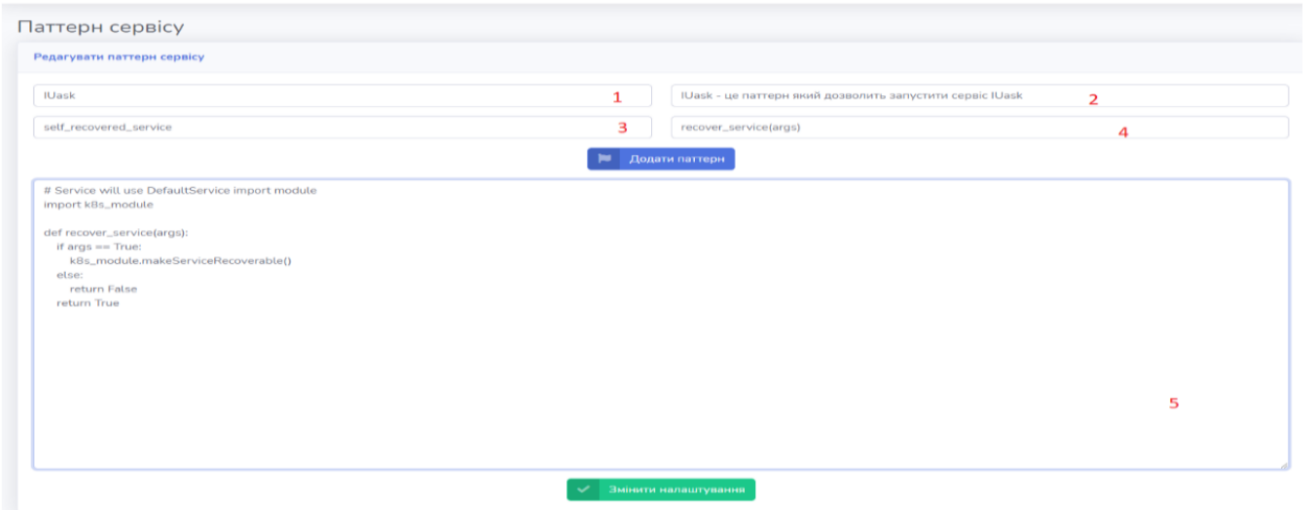


Рисунок 4.4 – Форма редагування конфігурації шаблону сервісу в інтерфейсі

На рисунку 4.5 представлено приклад заповнення форми конфігурації сервісу в межах веб-інтерфейсу інструмента. Кожен із ключових елементів форми виконує важливу роль у налаштуванні логіки роботи сервісу:

– рядок 2 – визначає цільовий стан сервісу, з яким пов’язується відповідна конфігурація; зміна цього стану автоматично ініціює виконання відповідного набору дій;
– рядок 3 – вказує на назву сервісу, який обрано як основу для створення шаблону (патерну);
– рядок 4 – містить основний YAML-опис конфігурації сервісу, включно з можливістю вбудованого використання додаткових інструкцій `ansible` у разі потреби.

Цей підхід дозволяє користувачеві гнучко задавати поведінку сервісу, розширювати її за допомогою скриптів та визначати, як саме сервіс має реагувати на зміну його стану або параметрів [38, 39].

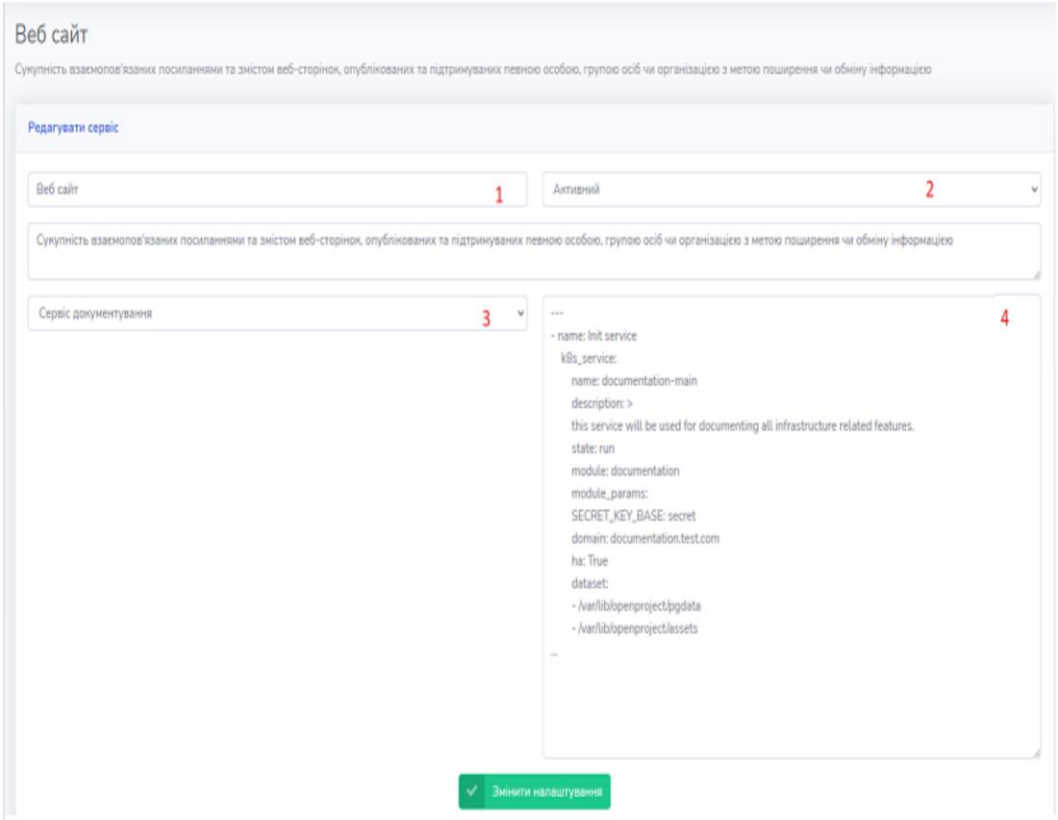


Рисунок 4.5 – Редактор властивостей сервісу через веб-інтерфейс

Після збереження змін у конфігурації, система автоматично відображає актуальну інформацію щодо стану сервісів в інфраструктурі.

Зокрема, у таблиці, представленій на рисунку 4.6, наведені ключові параметри, зокрема:

- статус поточного розгортання;
- фаза роботи сервісу (активний, помилка, оновлення тощо);
- версія використаної інфраструктури або шаблону конфігурації.

Такий підхід дозволяє оперативно здійснювати моніторинг та контроль усіх активних сервісів, забезпечуючи прозорість керування інфраструктурою в режимі реального часу [39].

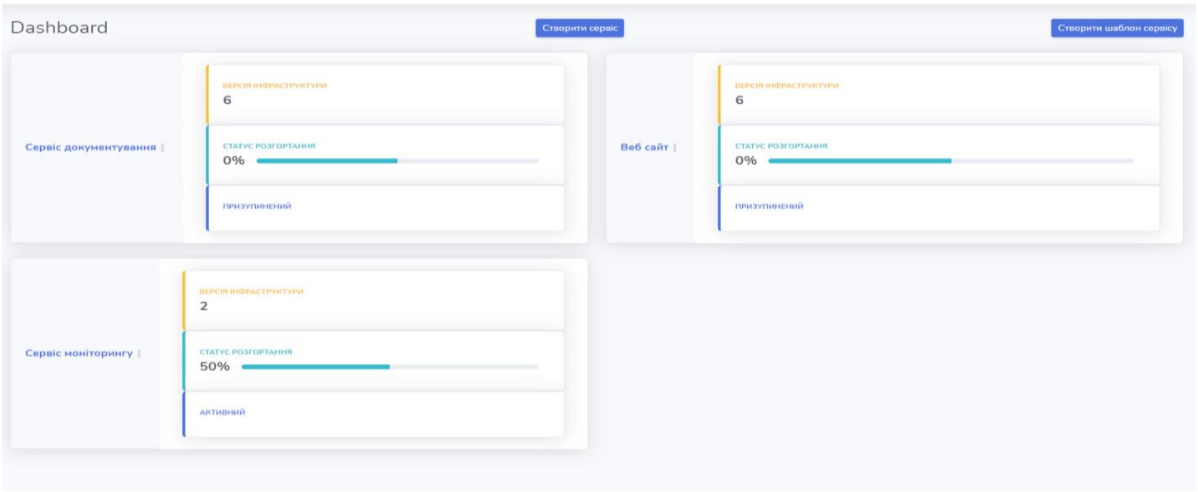


Рисунок 4.6 – Відображення поточного стану та версій компонентів системи

Після успішного створення сервісу, сформована конфігурація надсилається до оркестратора Kubernetes, де автоматично ініціюється створення об’єкта типу Deployment. У результаті цього процесу відбувається розгортання відповідних компонентів інфраструктури.

На рисунку 4.7 відображено підтвердження того, що сервіс було розгорнуто успішно – усі пов’язані Pod-и знаходяться у робочому стані. Це свідчить про активний статус сервісу, а також підтверджує правильність взаємодії між інструментом автоматизації та кластером Kubernetes [40].

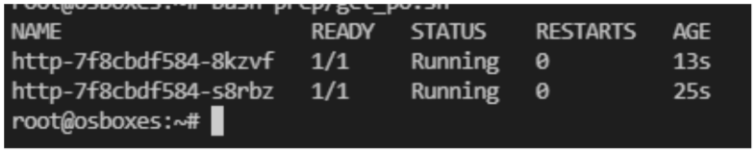


Рисунок 4.7 – Моніторинг активності сервісу через Kubernetes Dashboard

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розглянуто ключові поняття та класифікації інформаційних систем, зосереджено увагу на характерних проблемах застарілих ІС, а також окреслено виклики, пов'язані з їхньою модернізацією. Проведено детальний аналіз сучасних стандартів інтеграції, зокрема REST, SOAP, gRPC та GraphQL, з акцентом на їхню придатність до взаємодії зі спадковими системами. Окремо виокремлено роль хмарних обчислень як середовища для ефективного розгортання інтеграційних архітектур.

У другому розділі було проаналізовано сучасні технології та інструменти, що забезпечують гнучкість і автоматизацію процесів інтеграції. Зокрема, розглянуто підхід «інфраструктура як код» (IaC), можливості систем управління конфігурацією (Ansible, Puppet, Chef), а також роль контейнеризації (Docker) та оркестрації (Kubernetes). Особливу увагу приділено сервісам-посередникам, які забезпечують маршрутизацію та балансування даних між модулями.

На етапі розробки інтеграційного рішення було розроблено middleware-рівень, здатний об'єднувати старі інформаційні системи з новими платформами через оркестроване середовище. Обґрунтовано доцільність використання Kubernetes як базової інфраструктурної платформи для масштабованого розгортання сервісів. Реалізовано приклад внутрішнього веб-сервісу, що ілюструє ефективність архітектури, а також створено адаптивні модулі через механізми Ansible.

У практичній частині роботи продемонстровано приклад використання створеного інтеграційного рішення у вигляді сервісно-орієнтованої архітектури. Наведено конкретні конфігурації сервісів, способи їх розгортання, описано формат інтерфейсів взаємодії та механізми моніторингу через веб-інтерфейс. Веб-модулі, API-запити та шаблони конфігурацій довели свою ефективність у тестових сценаріях.

Для оцінки працездатності рішення було виокремлено кілька рівнів тестування: модульне, інтеграційне, а також тестування в продуктивному середовищі.

Застосовано методики перевірки сумісності, моделювання реальних навантажень та аналіз логічної взаємодії між компонентами. Результати підтвердили стійкість, гнучкість і адаптивність архітектурного рішення при роботі зі спадковими системами.

У підсумку можна стверджувати, що поставлені цілі роботи було досягнуто. Розроблене рішення має високу практичну цінність для підприємств, які мають застарілі інформаційні системи, але прагнуть інтегрувати їх у сучасну цифрову екосистему без повної заміни або кардинальної перебудови.

ПЕРЕЛІК ПОСИЛАНЬ

1. REST API Design: Resource Modeling [Електронний ресурс]. – Режим доступу: https://restfulapi.net/resource-naming/ (дата звернення: 22.05.2025).
2. XML SOAP - W3Schools [Електронний ресурс]. – Режим доступу: https://www.w3schools.com/xml/xml_soap.asp (дата звернення: 22.05.2025).
3. GraphQL – A query language for your API [Електронний ресурс]. – Режим доступу: https://graphql.org/ (дата звернення: 22.05.2025).
4. gRPC – A high-performance, open-source universal RPC framework [Електронний ресурс]. – Режим доступу: https://grpc.io/docs/ (дата звернення: 22.05.2025).
5. Terraform by HashiCorp: Documentation [Електронний ресурс]. – Режим доступу: https://developer.hashicorp.com/terraform/docs (дата звернення: 22.05.2025).
6. Ansible Documentation [Електронний ресурс]. – Режим доступу: https://docs.ansible.com/ (дата звернення: 22.05.2025).
7. Puppet Documentation [Електронний ресурс]. – Режим доступу: https://puppet.com/docs/puppet/latest/puppet_index.html (дата звернення: 22.05.2025).
8. Chef – Infrastructure Automation [Електронний ресурс]. – Режим доступу: https://docs.chef.io/ (дата звернення: 22.05.2025).
9. Docker: Overview and Getting Started [Електронний ресурс]. – Режим доступу: https://docs.docker.com/get-started/overview/ (дата звернення: 22.05.2025).
10. Kubernetes Documentation: Overview [Електронний ресурс]. – Режим доступу: https://kubernetes.io/docs/concepts/overview/ (дата звернення: 22.05.2025).
11. Kubernetes vs Docker – Red Hat [Електронний ресурс]. – Режим доступу: https://www.redhat.com/en/topics/containers/kubernetes-vs-docker (дата звернення: 22.05.2025).
12. What is a Service Mesh? – Red Hat [Електронний ресурс]. – Режим доступу: https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh (дата звернення: 22.05.2025).

13. Istio Documentation: What is Istio [Електронний ресурс]. – Режим доступу: https://istio.io/latest/docs/concepts/what-is-istio/ (дата звернення: 22.05.2025).

14. API Gateway pattern – Microsoft Azure Architecture Center [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/azure/architecture/patterns/api-gateway (дата звернення: 22.05.2025).

15. AWS API Gateway Documentation [Електронний ресурс]. – Режим доступу: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html (дата звернення: 22.05.2025).

16. What is an Enterprise Service Bus (ESB)? – MuleSoft [Електронний ресурс]. – Режим доступу: https://www.mulesoft.com/resources/esb/what-esb (дата звернення: 22.05.2025).

17. Apache Camel Documentation [Електронний ресурс]. – Режим доступу: https://camel.apache.org/manual/latest/index.html (дата звернення: 22.05.2025).

18. Hybrid Cloud – IBM Cloud Learn Hub [Електронний ресурс]. – Режим доступу: https://www.ibm.com/cloud/learn/hybrid-cloud (дата звернення: 22.05.2025).

19. Fowler M. Infrastructure as Code [Електронний ресурс] // martinfowler.com. – Режим доступу: https://martinfowler.com/bliki/InfrastructureAsCode.html (дата звернення: 22.05.2025).

20. How To Set Up a CI/CD Pipeline with Docker and Kubernetes – DigitalOcean [Електронний ресурс]. – Режим доступу: https://www.digitalocean.com/community/tutorials/how-to-set-up-ci-cd-pipeline-using-github-actions-and-docker-on-kubernetes (дата звернення: 22.05.2025).

21. An Introduction to Configuration Management with Ansible [Електронний ресурс] – Режим доступу: https://www.digitalocean.com/community/conceptual-articles/an-introduction-to-configuration-management-with-ansible

22. Overview - Kubernetes [Електронний ресурс] – Режим доступу: https://kubernetes.io/docs/concepts/overview/

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

24. What is Infrastructure as Code (IaC)? - Red Hat [Електронний ресурс] – Режим доступу: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac

25. A complete guide to REST API integration - Workato [Електронний ресурс] – Режим доступу: https://www.workato.com/the-connector/rest-api-integration

26. Introduction to gRPC [Електронний ресурс] – Режим доступу: https://grpc.io/docs/what-is-grpc/introduction/

27. What Is a SOAP API and How Does It Work? - Postman Blog [Електронний ресурс] – Режим доступу: https://blog.postman.com/soap-api-definition/

28. GraphQL | A query language for your API [Електронний ресурс] – Режим доступу: https://graphql.org/

29. Amazon API Gateway - AWS Documentation [Електронний ресурс] – Режим доступу: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

30. What is ESB? - Enterprise Service Bus Explained - AWS [Електронний ресурс] – Режим доступу: https://aws.amazon.com/what-is/enterprise-service-bus/

31. What is service mesh and why do we need it? - Dynatrace [Електронний ресурс] – Режим доступу: https://www.dynatrace.com/news/blog/what-is-a-service-mesh/

32. unittest – Unit testing framework – Python 3 documentation [Електронний ресурс] – Режим доступу: https://docs.python.org/3/library/unittest.html

33. Integration Testing Tutorial: A Comprehensive Guide With Examples [Електронний ресурс] – Режим доступу: https://www.lambdatest.com/learning-hub/integration-testing

34. Tools for Monitoring Resources - Kubernetes [Електронний ресурс] – Режим доступу: https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-usage-monitoring/

35. What Is Ansible? – Configuration Management with Ansible | NetApp [Електронний ресурс] – Режим доступу: https://www.netapp.com/hybrid-cloud/it-automation/what-is-ansible-configuration-management/

36. Configuration Management Made Easy with Puppet [Електронний ресурс] – Режим доступу: https://www.puppet.com/why-puppet/use-cases/continuous-configuration-automation

37. What is Infrastructure as Code (IaC)? - Azure DevOps | Microsoft Learn [Електронний ресурс] – Режим доступу: https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code

38. REST API Introduction | GeeksforGeeks [Електронний ресурс] – Режим доступу: https://www.geeksforgeeks.org/rest-api-introduction/

39. Core concepts, architecture and lifecycle - gRPC [Електронний ресурс] – Режим доступу: https://grpc.io/docs/what-is-grpc/core-concepts/

40. How Puppet works: Using the configuration management tool [Електронний ресурс] – Режим доступу: https://www.techtarget.com/searchitoperations/tip/What-is-the-Puppet-configuration-management-tool-and-how-does-it-work

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра Комп'ютерної інженерії



ТЕХНОЛОГІЇ ДЛЯ ІНТЕГРАЦІЇ СТАРИХ ІНФОРМАЦІЙНИХ СИСТЕМ ІЗ СУЧАСНИМИ СТАНДАРТАМИ

Виконала:
Дубова Яна Олександрівна
Науковий керівник:
Асєєва Л.А.

Київ 2025

Мета роботи:

- ***Об’єкт дослідження*** **–** інформаційні системи, що потребують інтеграції з сучасними технологічними стандартами.
- ***Предмет дослідження*** **–** методи та технології інтеграції застарілих (legacy) інформаційних систем із сучасними платформами, включаючи використання API, сервісно-орієнтованої архітектури, контейнеризації та інструментів автоматизованого розгортання.
- ***Мета роботи*** **–** розробка та реалізація підходу до інтеграції старих інформаційних систем із сучасними стандартами з використанням віртуалізації, мікросервісної архітектури, API-шлюзів та проміжного програмного забезпечення.

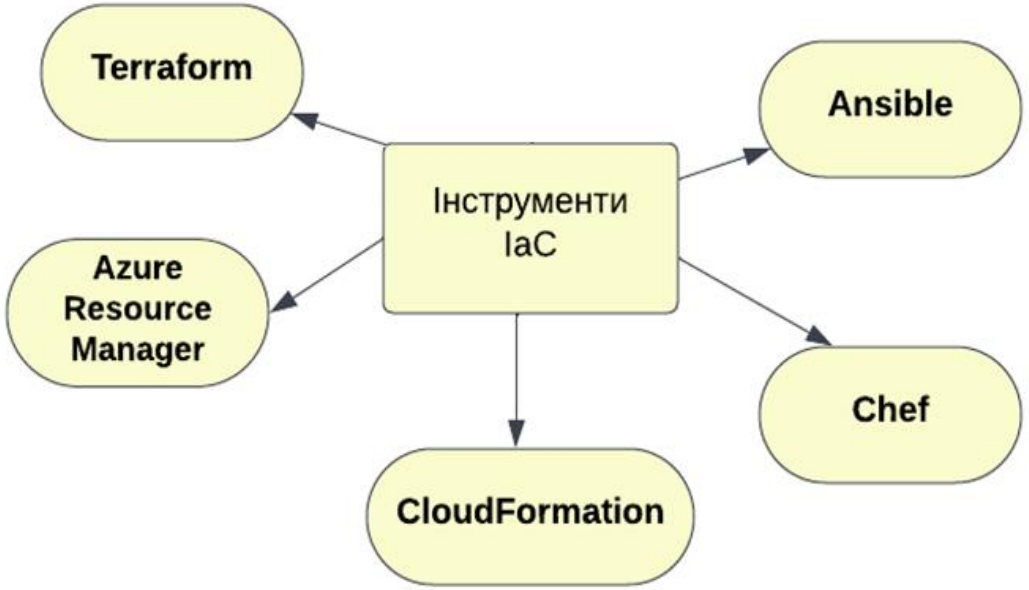
Актуальність

- Інтеграція старих систем із сучасними технологічними стандартами дозволяє зберегти накопичені дані, мінімізувати витрати на повну заміну програмного забезпечення та забезпечити поступовий перехід до нових архітектур. У цьому контексті особливого значення набувають технології мікросервісної архітектури, API-шлюзи, контейнеризація, інструменти автоматизованого розгортання, а також підхід «інфраструктура як код» (IaC).
- Актуальність теми зумовлена потребою підприємств модернізувати свої IT-інфраструктури без ризиків для стабільності бізнесу, забезпечити інтеграцію з хмарними сервісами та реалізувати стратегії цифрової трансформації.

Класифікація інформаційних систем



Інструменти для реалізації підходу IaC



Порівняння MongoDB і Oracle NoSQL



Рисунок 1. Структура взаємодії компонентів у віртуалізованому середовищі VM

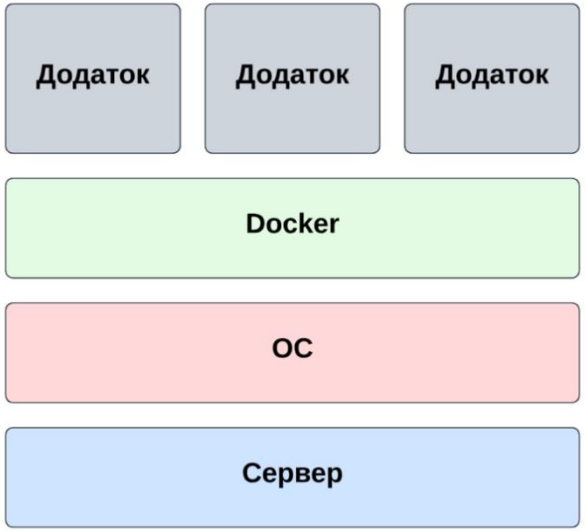


Рисунок 2. Модель реалізації контейнерної ізоляції для інтеграційних сервісів

```
ghhf@osboxes:~$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
mysql-pass-c57bb4t7mf   Opaque                                1      9s
ghhf@osboxes:~$ kubectl get pvc
NAME             STATUS    VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS       AGE
mysql-pv-claim   Bound     pvc-8cbd7b2e-4044-11e9-b2bb-42010a800002   20Gi       RWO            standard           77s
wp-pv-claim      Bound     pvc-8cd0df54-4044-11e9-b2bb-42010a800002   20Gi       RWO            standard           77s
ghhf@osboxes:~$ kubectl get pods
NAME                               READY     STATUS    RESTARTS   AGE
wordpress-mysql-1894417608-x5dzt   1/1       Running   0          40s
ghhf@osboxes:~$ kubectl get services wordpress
NAME        TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
wordpress   LoadBalancer   10.0.0.89    <pending>     80:32406/TCP   4m
```

Рисунок 3. Тестування доступності WordPress-сервісу через LoadBalancer

Автоматизація розгортання через Ansible: схема, приклад, лог виконання

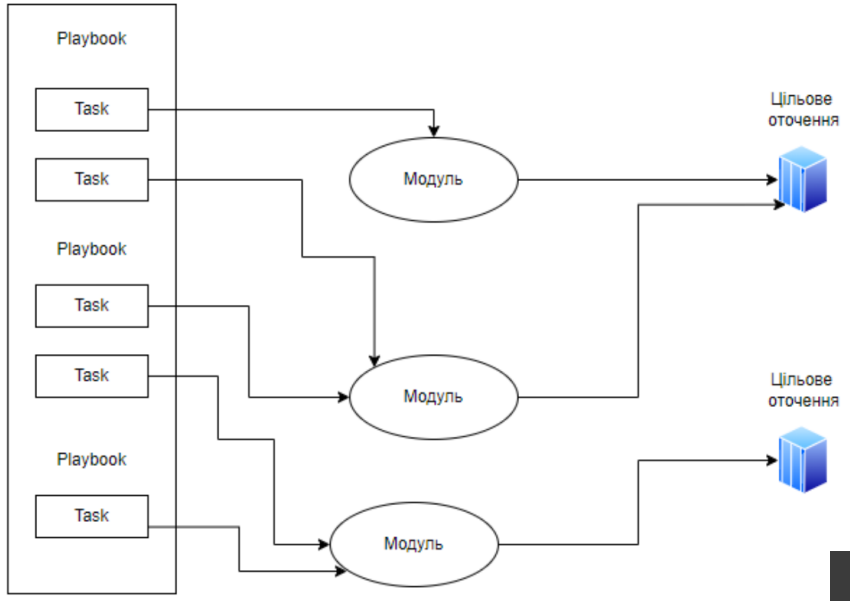


Рисунок 4. Схема взаємодії модулів Ansible з playbook та цільовим середовищем

```yaml
---
- name: Install ssh server
  apt:
    name: openssh-server
    state: latest
...
```

Рисунок 5. Автоматизоване встановлення
пакета через модуль apt у Ansible

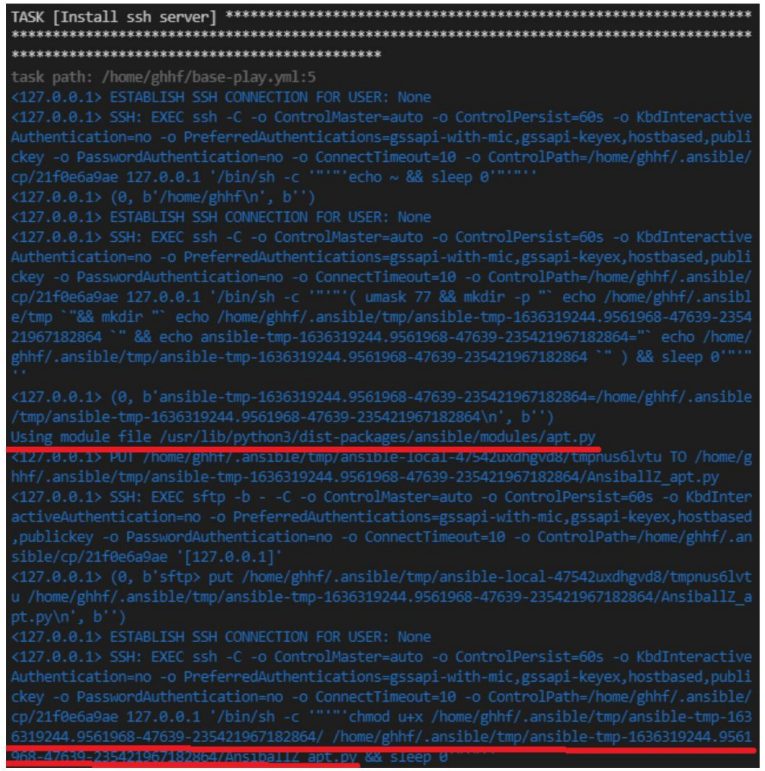


Рисунок 6. Журнал виконання команди встановлення пакета openssh-server у Ansible

Інтеграція Ansible та Kubernetes: структура моделі взаємодії та опис сервісу у YAML

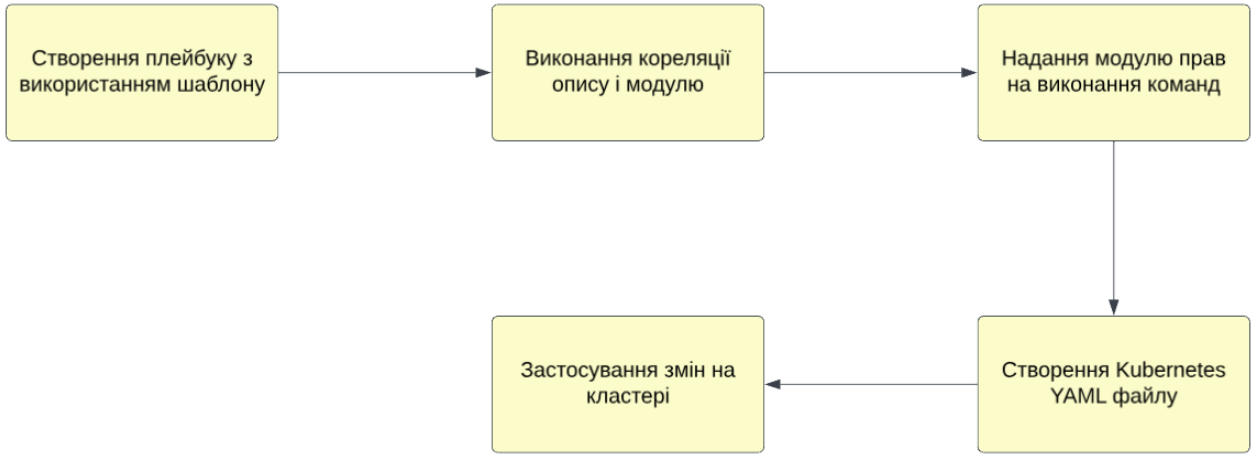


Рисунок 7. Структурна модель роботи Ansible, API та WEB-модулів інструмента

```yaml
- name: Init http service
  k8s_service:
    name: Web_site
    description: this service will be configured as a website and will be used for showing company information
    state: run
    module: web-server
    module_params:
    ports: 33820
    ha: True
    dataset:
      - /var/www/html
```

Рисунок 8. Структура опису сервісу у форматі YAML для Kubernetes

Контроль інфраструктури: Python-клас, unit-тестування та перевірка сервісів у Ansible

```python
import os
import kubeapi

class InfrastructureDebugger:
  #empty constructor
  def __init__(self):
    pass

  def showPodInfoLog(self, pod):
    return kubeapi.showPodLog(pod)

  def showDeploymentStatus(self, deployment):
    status = "-------Deployment: {}-------\n".format(deployment)
    status += kubeapi.showActiveDeploymentPodsCount(deployment)
    return status
```

Рисунок 9. Реалізація класу InfrastructureDebugger для виводу стану інфраструктури

```python
import unittest
from InfrastructureDebugger import InfrastructureDebugger
#Test cases to test Calulator methods
#You always create  a child class derived from unittest.TestCase
class TestInfrastructureDebugger(unittest.TestCase):
  #setUp method is overridden from the parent class TestCase
  def setUp(self):
    self.debugger = InfrastructureDebugger()

  def test_showPodInfoLog(self):
    self.assertEqual(self.debugger.showPodLog('empty_pod'), '')

  def test_showDeploymentStatus(self):
    self.assertEqual(self.debugger.showDeploymentStatus('test_deployment'),
     '-------Deployment: test_deployment-------\n')

if __name__ == "__main__":
  unittest.main()
```

Рисунок 11. Реалізація unit-тестування класу InfrastructureDebugger із використанням unittest

```yaml
---
- hosts: playground_cluster_group
  tasks:
    - name: apache server installation
      block:
        - name: apache server should be latest version
          yum:
            name: httpd
            state: latest

        - name: apache server restart
          service:
            name: httpd
            state: started

    - name: apache server verification
      script: verify.sh --apache

    - name: mysql server installation
      block:
        - name: install mysql independent
          yum:
            name: mariadb
            state: latest

    - name: mysql server verification
      script: verify.sh --mysql
```

Рисунок 10. Структурований приклад перевірки компонентів інфраструктури з використанням Ansible

Інтерфейси керування та конфігурації: візуальна панель Kubernetes і форма налаштування сервісів

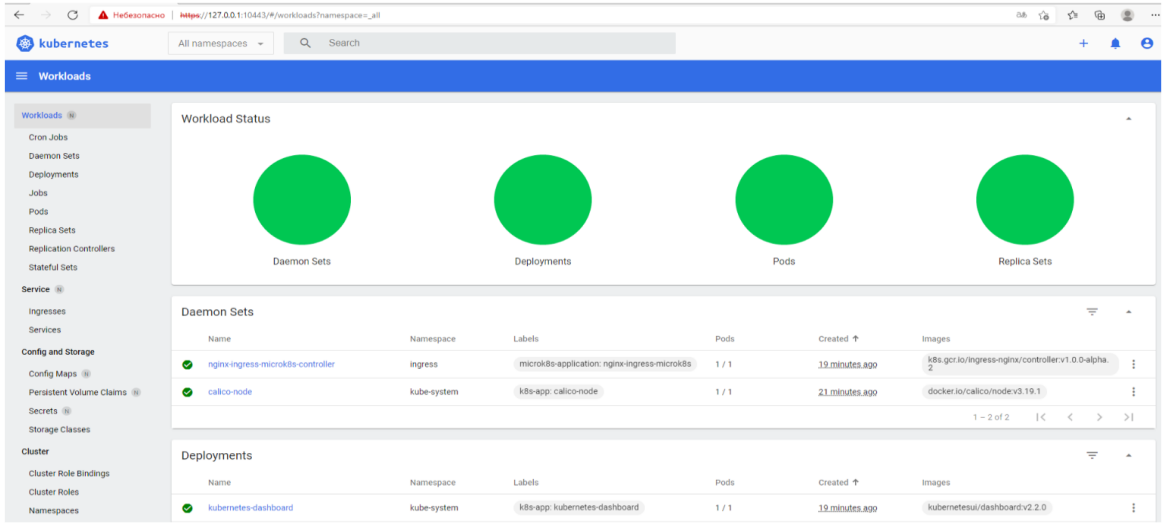


Рисунок 12. Графічна панель адміністрування Kubernetes-кластеру

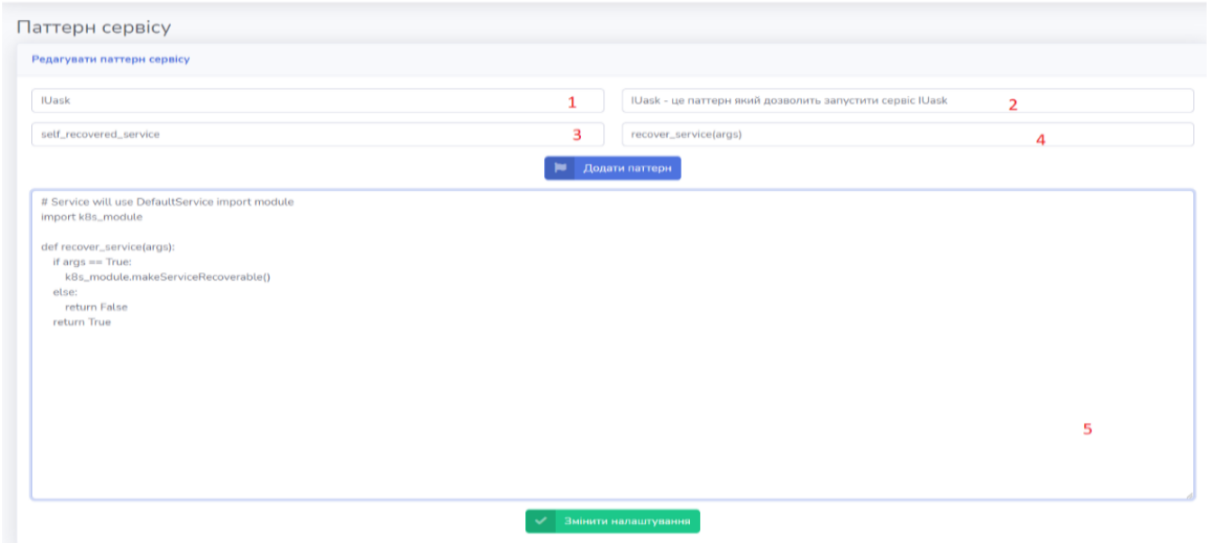


Рисунок 13. Форма редагування конфігурації шаблону сервісу в інтерфейсі

Редагування, статус і моніторинг сервісів у Kubernetes через веб-інтерфейс

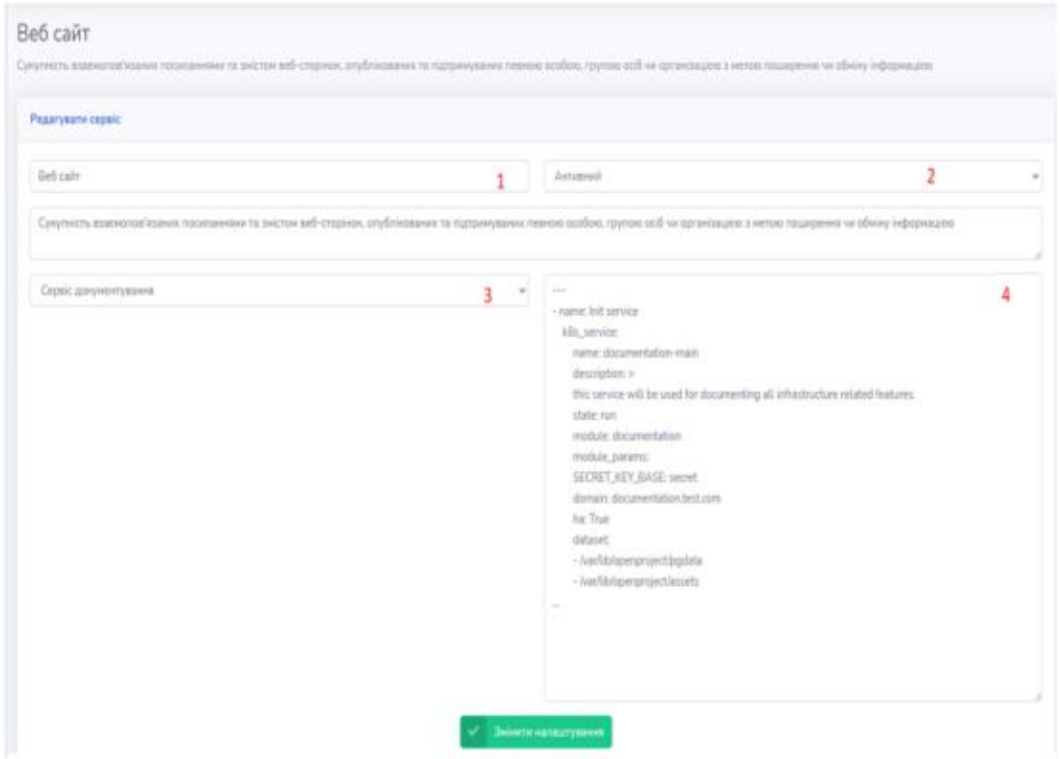


Рисунок 14. Редактор властивостей сервісу через веб-інтерфейс

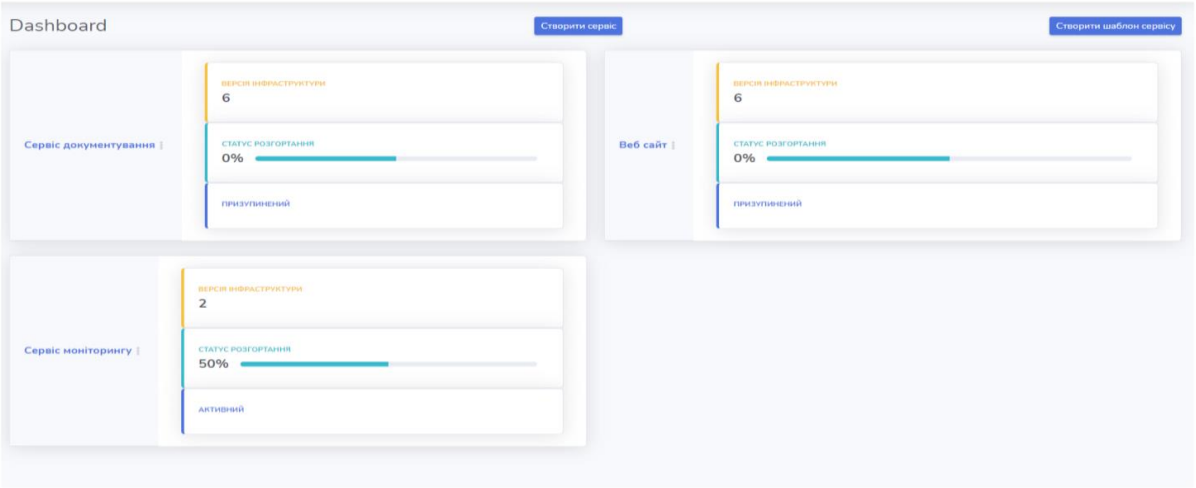


Рисунок 15. Відображення поточного стану та версій компонентів системи

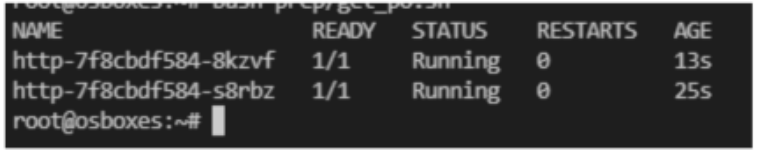


Рисунок 16. Моніторинг активності сервісу через Kubernetes Dashboard

Висновки

- У межах кваліфікаційної роботи проаналізовано проблеми застарілих інформаційних систем і сучасні стандарти їх інтеграції (REST, SOAP, gRPC, GraphQL).
- Розглянуто інструменти автоматизації — IaC, Docker, Kubernetes, Ansible. Розроблено middleware-рівень для з’єднання старих систем з новими платформами.
- Проведено аналіз веб-сервіс та адаптивні модулі, проведено тестування на різних рівнях.
- Результати підтвердили ефективність, гнучкість і сумісність запропонованої архітектури.