НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет/(ННІ) **Факультет інформаційних технологій**

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету (Директор ННІ)	**Завідувач кафедри**
інформаційних технологій .	комп'ютерних наук .
(назва факультету (ННІ))	(назва кафедри)
___________ Ігор БОЛБОТ .	_________ Белла ГОЛУБ .
(підпис) (ім'я ПРІЗВИЩЕ)	(підпис) (ім'я ПРІЗВИЩЕ)
"___" ___________ 20_ р.	"___" ___________ 20_ р.

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему

Аналіз та розробка дорадчої системи вибору оптимального набору компонентів ОС Linux

Спеціальність
121 інженерія програмного забезпечення
(код і найменування)

Освітня програма
Програмне забезпечення інформаційнх систем
(назва)

Орієнтація освітньої програми
освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми

к.ф.-м.н., доцент ______________ Віктор КИРИЧЕНКО
(науковий ступінь та вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

Керівник магістерської кваліфікаційної роботи

к.т.н., доцент ______________ Дмитро НІКОЛАЄНКО
(науковий ступінь та вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

Виконав ________________ Андрій БОНДАРЧУК
(підпис) (ім'я ПРІЗВИЩЕ здобувача)

КИЇВ – 2025

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ**

Факультет (ННІ) Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних наук

к.т.н., доцент _____ Белла ГОЛУБ

(науковий ступінь, вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

"____" ________________20_____року

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ ЗДОБУВАЧУ

Бондарчуку Андрію Сергійовичу

(прізвище, ім'я, по батькові)

Спеціальність 121 Інженерія програмного забезпечення

(код і найменування)

Освітня програма Програмне забезпечення інформаційних систем

(назва)

Орієнтація освітньої програми освітньо-професійна

(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи Аналіз та розробка дорадчої системи вибору оптимального набору компонентів ОС Linux

затверджена наказом від "01" листопада 2024р. № 1963 «С»

Термін подання завершеної роботи на кафедру 28.11.2025
(рік, місяць, число)

Вихідні дані до магістерської кваліфікаційної роботи: вимоги до серверної інфраструктури Linux, перелік типових компонентів, критерії оцінюваннязалежності та конфлікти між компонентами, сучасні DevOps-підходи.

Перелік питань, що підлягають дослідженню:

1. Проаналізувати сучасні підходи до розгортання Linux-інфраструктури та визначити ключові проблеми вибору компонентів.
2. Обґрунтувати використання графової моделі знань та розробити структуру бази знань для опису залежностей і конфліктів між компонентами.
3. Розробити та реалізувати прототип дорадчої системи (Rule Engine, Criteria Module, API-рівень), протестувати його в контейнерному середовищі та оцінити якість сформованих рекомендацій.

Перелік графічного матеріалу (за потреби) презентація, постер

Дата видачі завдання "01" листопада 2024 р.

Керівник магістерської кваліфікаційної роботи ______ Дмитро НІКОЛАЄНКО
(підпис) (ім'я ПРІЗВИЩЕ)

Завдання прийняв до виконання _______ Андрій БОНДАРЧУК

ЗМІСТ

ВСТУП 7

РОЗДІЛ 1. Теоретичні основи побудови дорадчих систем та оптимізації Linux-інфраструктури 10

1.1. Аналіз існуючих підходів до розгортання серверної інфраструктури Linux 10

1.2. Поняття та принципи роботи дорадчих систем 12

1.3. Методи прийняття рішень у системах підтримки вибору компонентів ... 14

1.4. Архітектурні моделі автоматизації DevOps-процесів 15

1.5. Аналіз існуючих рішень у сфері автоматизації конфігурацій 17

1.6. Формалізація знань у вигляді графових моделей 20

1.7. Висновки до розділу 1 23

РОЗДІЛ 2. АНАЛІЗ ЗАДАЧІ ТА ПРОЄКТУВАННЯ ПРОГРАМНОЇ ДОРАДЧОЇ СИСТЕМИ ВИБОРУ КОМПОНЕНТІВ ОС LINUX 24

2.1. Постановка задачі та визначення вимог до системи 24

2.2. Вибір методології представлення знань та алгоритмів рекомендацій...... 27

2.2.1. Критерії вибору методології 27

2.2.2. Порівняння підходів 28

2.2.3. Обґрунтування вибору графової моделі Neo4j + rule-engine........ 29

2.3. Вибір інструментальних засобів розробки 30

2.3.1. Критерії вибору технологій 30

2.3.2. Огляд популярних стеків DevOps-систем 31

2.3.3. Обґрунтування вибору Python + FastAPI + Neo4j........ 32

2.4. Архітектурне проєктування системи 33

2.4.1. Загальна структура та рівні системи 34

2.4.2. Модульна архітектура програмного комплексу 35

2.4.3. Логіка обміну даними між модулями 35

2.4.4. Архітектурні альтернативи (моноліт vs мікросервіси) 36

2.4.5. Вибір архітектури та обґрунтування рішень........ 37

2.5. Модель обміну даними між компонентами системи 38

2.5.1. Основні принципи взаємодії........ 38

2.5.2. Сценарій 1 – CLI ↔ API ↔ Neo4j 39
2.5.3. Сценарій 2 – Інтеграція з Ansible / Terraform 41
2.5.4. Сценарій 3 – Docker-середовище 41
2.5.5. Механізми кешування та логування 42
2.6. Функціональна модель системи 42
2.7. Об’єктна модель системи 45
2.8. Дорадчий модуль системи 48
2.8.1. Концепція та місце дорадчого модуля у системі 49
2.8.2. Логічна структура дорадчого модуля 50
2.8.3. Алгоритм роботи дорадчого модуля в реальному запиті 51
2.8.4. Властивість пояснюваності (Explainable Recommendations) 52
2.8.5. Особливості проєктування та новизна 53
2.9. Висновки до розділу 2 54
РОЗДІЛ 3. РОЗРОБКА ТА РЕАЛІЗАЦІЯ СИСТЕМИ 56
3.1. Загальна структура реалізації системи 56
3.2. Реалізація бази знань у Neo4j 59
3.2.1. Загальна структура графа знань 59
3.2.2. Типи зв’язків у графі знань 60
3.2.3. Приклад тестової підмножини бази знань 60
3.2.4. Заповнення бази знань 61
3.2.5. Взаємодія з Neo4j у межах системи 61
3.3. Реалізація Rule Engine та Criteria Module 62
3.3.1. Концепція Rule Engine 62
3.3.2. Структура правила 63
3.3.3. Взаємодія Rule Engine з графовою базою знань 64
3.3.4. Реалізація Criteria Module 64
3.3.5. Механізм ранжування 65
3.3.6. Результати роботи Rule Engine та Criteria Module 65
3.4. Реалізація API-рівня (FastAPI) 66
3.4.1. Основні ендпоінти 66
3.4.2. Структура запиту та відповіді 67

3.4.3. Алгоритм роботи API 68
3.4.4. Особливості використання FastAPI 68
3.5. Генерація конфігурацій (YAML/JSON) 69
3.5.1. Підтримувані формати 69
3.5.2. Структура YAML-конфігурації (опис) 69
3.5.3. Структура JSON-конфігурації (опис) 70
3.5.4. Алгоритм роботи TemplateGenerator 71
3.5.5. Приклад роботи 71
3.5.6. Переваги підходу 71
3.6. Контейнеризація системи (Docker) 71
3.6.1. Архітектура контейнерного середовища 71
3.6.2. Структура docker-compose.yml 72
3.6.3. Ініціалізація графа 72
3.6.4. Оточення API-сервісу 73
3.6.5. Переваги контейнеризації 73
3.6.6. Практичне застосування 73
3.7. Тестування роботи системи 73
3.7.1. Процедура тестування та її принципи 74
3.7.2. Аналіз роботи Rule Engine та Criteria Module 75
3.7.3. Перевірка формування конфігураційних файлів 75
3.7.4. Інтеграційне тестування в контейнерному середовищі 76
3.7.5. Аналіз стабільності та продуктивності системи 76
3.8. Висновки до розділу 3 77
РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ СИСТЕМИ 79
4.1. Методика проведення експериментів 79
4.2. Тестування API та функціональних модулів 81
4.2.1. Модульне тестування 81
4.2.2. Інтеграційне тестування API 82
4.3. Оцінка швидкодії (FastAPI, Neo4j, Rule Engine) 83
4.3.1. Час відповіді API 83

4.3.2. Продуктивність Neo4j 84
4.3.3. Продуктивність Rule Engine та Criteria Module 84
4.4. Якість рекомендацій (точність, консистентність, сумісність) 85
4.5. Оцінка масштабованості та стійкості системи 88
4.6. Порівняння з ручним підбором конфігурацій 89
4.7. Можливості оптимізації та розвитку системи 90
4.8. Висновки до розділу 4 92
ВИСНОВКИ **94**
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ **96**
ДОДАТКИ **98**
Додаток А 98
Додаток Б 99
Додаток В 100
Додаток Г 102
Додаток Ґ 103
Додаток Д 105

ВСТУП

Сучасні серверні інфраструктури стають дедалі складнішими, оскільки підприємства активно переходять до використання розподілених систем, хмарних платформ та автоматизованих підходів до управління IT-ресурсами. У таких умовах зростає потреба не лише у стандартизації процесів розгортання серверів, а й у виборі оптимального набору компонентів операційної системи Linux, що визначає продуктивність, масштабованість, безпеку та надійність усієї інфраструктури. Водночас ринок Linux-рішень пропонує значну кількість дистрибутивів, моделей пакетування, сервісів оптимізації та систем автоматизації, що суттєво ускладнює прийняття зважених технічних рішень, особливо для новачків-адміністраторів або команд, які переходять до практик DevOps.

Актуальність теми дослідження зумовлена необхідністю створення інструментів, здатних підтримувати системних адміністраторів у процесі прийняття рішень під час побудови Linux-інфраструктури. На практиці неправильний вибір компонентів – таких як сервери баз даних, веб-сервери, інструменти віртуалізації чи засоби безпеки – призводить до зниження продуктивності, появи конфліктів залежностей, зростання кількості помилок розгортання та ускладнення подальшої експлуатації. У великих компаніях ці ризики масштабуються: кожна некоректна конфігурація впливає на сотні сервісів і тисячі користувачів. Тому виникає потреба у дорадчій системі, яка б аналізувала вимоги, коректно оцінювала доступні варіанти та пропонувала оптимальний набір компонентів з урахуванням архітектури, сценарію використання та обмежень інфраструктури.

Додатковою підставою актуальності є зростання ролі автоматизації у DevOps-процесах. Автоматизовані системи розгортання, такі як Ansible, Puppet або Terraform, значно полегшують процеси конфігурації, проте сам вибір необхідних компонентів все ще залишається на відповідальності інженера. Створення рекомендаційної системи дає змогу зменшити кількість помилок,

підвищити повторюваність конфігурацій, стандартизувати підходи до планування серверної архітектури та скоротити час, необхідний на проектування інфраструктури зараз і в майбутньому.

Метою магістерської роботи є аналіз методів побудови дорадчої систем та розробка програмного комплексу для автоматизованого вибору оптимального набору компонентів ОС Linux у процесі розгортання серверної інфраструктури.

Для досягнення поставленої мети необхідно виконати такі завдання:

1. проаналізувати сучасні підходи до побудови та оптимізації Linux-інфраструктури у DevOps-середовищах;
2. дослідити методи побудови рекомендаційних та знання-орієнтованих систем;
3. сформувати структуру моделі представлення знань про серверні компоненти й типові сценарії їх використання;
4. розробити алгоритми формування рекомендацій на основі вимог користувача та характеристик інфраструктури;
5. створити прототип дорадчої системи з можливістю інтеграції з інструментами автоматизації розгортання;
6. провести тестування системи в умовах лабораторного середовища та оцінити її ефективність порівняно з традиційним ручним підходом.

Об'єктом дослідження є процес побудови та конфігурації серверної інфраструктури на базі операційної системи Linux.

Предметом дослідження є методи формування рекомендацій щодо вибору оптимального набору серверних компонентів для різних сценаріїв розгортання.

Методи дослідження ґрунтуються на аналізі наукових джерел, порівняльному аналізі програмних компонентів, методах експертних систем, графових моделях представлення знань, експериментальних дослідженнях та моделюванні сценаріїв розгортання. У роботі використано елементи структурного аналізу, логічного моделювання та практичних експериментів у лабораторному середовищі на базі віртуалізації.

Практичне значення роботи полягає у створенні програмного прототипу дорадчої системи, яка може бути використана у навчальних цілях, внутрішній автоматизації підприємств та під час проєктування DevOps-інфраструктури. Система дозволяє зменшити кількість помилок конфігурації, спростити навчання молодих адміністраторів, підвищити стандартизованість рішень та скоротити час на підготовку серверних конфігурацій.

Структура роботи складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. У першому розділі розглянуто теоретичні засади побудови серверної інфраструктури Linux та принципи створення рекомендаційних систем. У другому розділі наведено аналіз типових сценаріїв розгортання, класифікацію компонентів та вимог до них. Третій розділ присвячено розробці структури та реалізації прототипу системи. У четвертому розділі подано результати тестування, оцінку ефективності та аналіз переваг автоматизованого підходу порівняно з ручною конфігурацією.

РОЗДІЛ 1. Теоретичні основи побудови дорадчих систем та оптимізації Linux-інфраструктури

1.1. Аналіз існуючих підходів до розгортання серверної інфраструктури Linux

Операційна система Linux є однією з базових платформ для розгортання сучасних серверних рішень. Її популярність зумовлена відкритістю програмного коду, стабільністю, надійністю та можливістю гнучкого налаштування під специфічні потреби адміністратора чи користувача [1–3]. На основі Linux побудовано інфраструктури дата-центрів, хмарні сервіси, системи автоматизації виробництва, аналітичні комплекси та веб-платформи.

Залежно від вимог до продуктивності, стабільності та моделі підтримки застосовуються різні дистрибутиви Linux. Зокрема, Debian орієнтований на стабільні сервери загального призначення, Ubuntu Server часто використовується для гнучкого та відносно швидкого розгортання сервісів, CentOS Stream і Rocky Linux – у корпоративних середовищах, сумісних з екосистемою Red Hat, тоді як OpenSUSE та Fedora Server нерідко застосовуються у тестових та розробницьких середовищах [4]. Кожен із зазначених дистрибутивів має власну систему управління пакетами (apt, dnf, zypper тощо), що безпосередньо впливає на процес підтримки та оновлення програмного забезпечення.

Типовий процес розгортання Linux-сервера включає такі послідовні етапи:

– вибір дистрибутиву відповідно до вимог проєкту, терміну підтримки та наявності необхідних пакетів;

– інсталяція базових служб (SSH-сервер, служби журналювання, системні демони), налаштування мережевих інтерфейсів та механізмів логування;

– конфігурування безпеки, що охоплює створення користувачів, налаштування міжмережевих екранів (iptables, nftables), активацію та налаштування механізмів контролю доступу (SELinux або AppArmor);

– впровадження засобів моніторингу та резервного копіювання (Prometheus, Zabbix, Grafana, rsnapshot, Bacula тощо);

– розгортання прикладних сервісів, зокрема веб-серверів (Nginx, Apache), систем керування базами даних (PostgreSQL, MySQL), брокерів повідомлень (RabbitMQ, Redis) та інших компонентів.

У класичному підході ці дії виконуються вручну або за допомогою окремих Bash-скриптів. Недоліками такого підходу є низька відтворюваність конфігурацій, відмінності у налаштуваннях на формально ідентичних вузлах, складність масштабування та значний вплив людського фактору [5].

З появою та поширенням DevOps-підходу адміністрування інфраструктури трансформувалося у модель Infrastructure as Code (IaC), в якій інфраструктурні ресурси описуються декларативно у вигляді коду [6]. Для цього застосовуються такі інструменти, як Ansible, Terraform, Puppet, SaltStack, Chef. Подібні системи дають змогу повторювано відтворювати середовища, пришвидшують розгортання серверів і зменшують кількість помилок конфігурації. Водночас навіть за наявності IaC-інструментів залишається невирішеним завдання раціонального вибору компонентів: який веб-сервер або СУБД доцільніше застосувати для конкретного сценарію, які параметри кешування обрати, який тип файлової системи забезпечить оптимальне співвідношення продуктивності та надійності тощо.

Розширення спектра доступних інструментів та конфігурацій призвело до того, що адміністратор часто змушений покладатися на власний досвід або фрагментарні відомості з документації й неформальних джерел. Це, у свою чергу, спричиняє ризики несумісностей, перевитрат ресурсів та зниження стабільності серверного середовища. Отже, виникає потреба у створенні дорадчої системи, яка б сприяла обґрунтованому формуванню оптимальних конфігурацій Linux-інфраструктури з урахуванням технічних, продуктивних та експлуатаційних параметрів.

Подальший розвиток парадигми Cloud Native та впровадження контейнеризації (Docker, Kubernetes) додатково ускладнюють процес вибору компонентів, оскільки висувають вимоги до мінімальності, безпеки та портативності образів [7]. У моделях незмінної інфраструктури (Immutable

Infrastructure), реалізованих, зокрема, у Fedora CoreOS та RHEL CoreOS, конфігурація компонентів виконується на етапі створення образу, а можливості подальших змін у робочому середовищі є обмеженими. У таких умовах правильний вибір компонентів і параметрів ще до етапу розгортання стає критично важливим, що додатково актуалізує задачу побудови інструментів підтримки прийняття рішень щодо конфігурації Linux-середовищ у віртуальних, контейнерних та оркестраційних платформах.

1.2. Поняття та принципи роботи дорадчих систем

Дорадча система (Decision Support System, DSS) – це програмно-технічний комплекс, призначений для підтримки користувача під час прийняття рішень на основі структурованих знань, логічних правил чи статистичних моделей [8]. Подібні системи застосовують у фінансовій аналітиці, медицині, логістиці, промисловості та інформаційних технологіях. У сфері IT вони допомагають формувати конфігурації, оптимізувати ресурси та зменшувати кількість помилок, пов'язаних із ручним налаштуванням.

Типова дорадча система складається з трьох основних компонентів:

– **бази знань**, у якій зберігаються правила, факти й відношення між об'єктами;

– **механізму виведення**, що аналізує дані з бази знань і формує висновки;

– **інтерфейсу взаємодії з користувачем** (CLI, REST API, веб-інтерфейс), через який задають вхідні параметри та отримують рекомендації.

Узагальнену структуру дорадчої системи вибору компонентів Linux-середовища наведено на рис. 1.1.

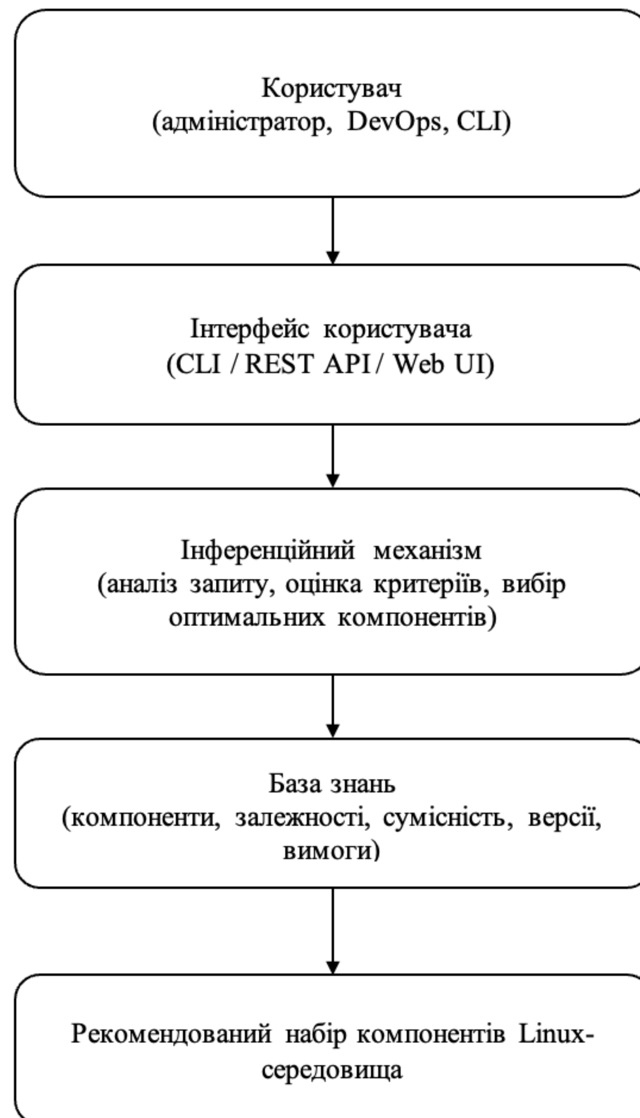


Рис. 1.1 – Узагальнена структура дорадчої системи вибору компонентів Linux-середовища

Дорадчі системи можна умовно поділити на кілька груп:

- **експертні**, що працюють за принципом правил «якщо – то»;
- **data-driven**, які використовують статистику або машинне навчання;
- **гібридні**, що поєднують обидва підходи [9].

У DevOps-середовищах найчастіше застосовують гібридний варіант. Базові правила сумісності та обмеження задаються експертно, а параметри пріоритетів та ваг критеріїв можуть уточнюватися за результатами попередніх розгортань. Подібні ідеї реалізовані у таких системах, як Red Hat Insights чи AWS Trusted Advisor, які аналізують конфігурації вузлів і пропонують рекомендації щодо продуктивності, безпеки та усунення потенційних помилок [10].

Застосування дорадчих систем у контексті Linux-інфраструктури підвищує відтворюваність конфігурацій, зменшує кількість ручних дій та дозволяє стандартизувати принципи побудови серверних середовищ.

1.3. Методи прийняття рішень у системах підтримки вибору компонентів

Формування оптимальної конфігурації серверного середовища Linux є багатокритеріальною задачею: для кожного компонента потрібно враховувати продуктивність, стабільність, ресурсоємність, сумісність, безпеку та особливості інтеграції [11]. Для прийняття рішень у таких умовах широко застосовуються методи багатокритеріального аналізу.

Найпоширеніші методи наведені у таблиці 1.1.

Таблиця 1.1

Порівняння методів оцінки альтернатив

Метод	**Принцип роботи**	**Переваги**	**Недоліки**
Зважена сума (WSM)	кожному критерію призначається вага; підсумкова оцінка – сума добутків ваг на оцінки	простота реалізації	результати залежать від вибору ваг
Аналіз ієрархій (AHP)	попарне порівняння альтернатив і критеріїв	висока точність	складність при великій кількості альтернатив
TOPSIS	вибір альтернативи, найближчої до «ідеальної»	враховує як позитивні, так і негативні фактори	складні процедури нормалізації
Правила (Rule-Based)	логічні правила «якщо – то»	прозорість логіки	складність підтримки великої бази знань

Для задачі вибору компонентів Linux-середовища найбільш практичним є гібридний підхід, що поєднує rule-based фільтрацію та метод зваженої суми.

1. Етап фільтрації (жорсткі обмеження). Використовуються логічні правила, які перевіряють сумісність версій, наявність обов’язкових залежностей та виключають конфліктні поєднання компонентів. На цьому етапі відсікаються варіанти, які гарантовано не працюватимуть.

2. Етап ранжування (м’які критерії). До залишених конфігурацій застосовується WSM із урахуванням вагових коефіцієнтів, які задаються користувачем або приймаються типовими для конкретного сценарію (наприклад, для сервера БД – продуктивність є важливішою за компактність образу). Такий підхід дозволяє одночасно враховувати як технічні обмеження (несумісність версій), так і пріоритети користувача, що робить механізм рекомендацій гнучким і наближеним до реальних умов конфігурації серверів.

1.4. Архітектурні моделі автоматизації DevOps-процесів

Розвиток DevOps-практик привів до появи різних моделей автоматизації, кожна з яких розв’язує власний клас задач і має специфічні вимоги до інфраструктури. Незалежно від реалізації, метою таких моделей є підвищення відтворюваності середовища, зменшення кількості ручних операцій і спрощення розгортання серверних систем [12].

У сучасних системах автоматизації найпоширенішими є такі архітектурні підходи.

1. Pull-модель (централізоване керування)

У цій моделі цільова конфігурація зберігається на центральному сервері, а керовані вузли періодично «підтягують» оновлення. Такий підхід реалізований у Puppet та Chef.

Переваги:

– централізований контроль конфігурацій;

– можливість масштабування великих інфраструктур.

Недоліки:

– складність початкового впровадження;

– вимога встановлення агентів на кожному вузлі.

Цю модель зазвичай використовують у великих корпоративних середовищах, де пріоритетом є контроль і аудит усіх змін.

2. Push-модель (безагентне керування)

У push-підході ініціатором змін виступає сервер автоматизації, який виконує операції на віддалених вузлах через SSH. Найвідомішим представником є Ansible.

Переваги:

– просте початкове налаштування;

– відсутність агентів на вузлах;

– зручність для невеликих і середніх інфраструктур.

Недоліки:

– затримки під час виконання операцій на великій кількості серверів;

– складніше забезпечити постійний контроль конфігурацій.

Push-модель часто застосовують у середовищах, де потрібно швидко розгортати окремі сервери або кластери без значного попереднього налаштування.

3. Інфраструктура як код (Infrastructure as Code, IaC)

IaC передбачає декларативний опис усіх інфраструктурних ресурсів – від мережевих підсистем до віртуальних машин і балансувальників. Такий підхід реалізовано у Terraform, AWS CloudFormation, OpenTofu та інших інструментах.

Переваги:

– відтворюваність середовища;

– контроль версій і простота відкату;

– інтеграція з CI/CD.

Недоліки:

– складність опису нестандартних або змішаних інфраструктур;

– залежність від провайдера (наприклад, від синтаксису Terraform чи CloudFormation).

IaC стає основою сучасної автоматизації, оскільки дозволяє перевести інфраструктуру в категорію керованих артефактів, аналогічно до програмного коду.

4. Контейнеризація та оркестрація (Docker, Kubernetes)

Контейнеризація ізолює сервіси в легкі контейнери, а оркестратори (наприклад, Kubernetes) відповідають за їх розміщення, масштабування й відновлення.

Переваги:

– портативність;

– швидке масштабування;

– можливість стандартизації сервісів у вигляді контейнерів.

Недоліки:

– складність моніторингу та діагностики;

– підвищені вимоги до безпеки;

– необхідність окремих інструментів для роботи з конфігурацією (Helm, Kustomize).

Цей підхід широко застосовується у Cloud Native-інфраструктурах і є стандартом де-факто для сучасних веб-платформ.

1.5. Аналіз існуючих рішень у сфері автоматизації конфігурацій

Щоб оцінити, наскільки автоматизованим може бути процес вибору серверних компонентів, доцільно проаналізувати інструменти, які сьогодні застосовуються для управління інфраструктурою та конфігураціями. Більшість із них зосереджені на автоматизації розгортання або зберіганні конфігурацій, однак лише частково враховують взаємозалежності між компонентами чи забезпечують механізми рекомендацій.

Однією з найбільш поширених платформ для повторного використання конфігураційних шаблонів є **Ansible Galaxy** – каталог ролей та плейбуків, що містить декларативні YAML-файли для різних сервісів і типових сценаріїв розгортання. Перевагою платформи є велика кількість доступних модулів і можливість стандартизувати конфігурації. Проте Ansible Galaxy не перевіряє сумісність ролей між собою та не пропонує рекомендацій щодо вибору оптимальних конфігурацій, що обмежує її використання для автоматичного прийняття рішень [13].

Схожу функцію виконує **Terraform Registry**, який надає каталог модулів для різних провайдерів хмарних сервісів. Модулі описані мовою HCL і дозволяють автоматизувати створення інфраструктури у вигляді коду. Однак, як і у випадку з Ansible Galaxy, платформа не містить інструментів для аналізу залежностей між модулями та не забезпечує рекомендацій щодо вибору конфігурацій з урахуванням продуктивності або сумісності [13].

Ще одним прикладом порталу конфігурацій є **LinuxServer.io**, який надає готові контейнерні образи для популярних серверних застосунків. Вони мають стандартизовані параметри та документацію, що спрощує розгортання контейнеризованих сервісів. Проте цей ресурс орієнтований переважно на Docker-середовища й не охоплює повноцінного підбору компонентів поза межами контейнерних платформ.

Для порівняння основних можливостей таких платформ наведено таблицю 1.2.

Таблиця 1.2

Порівняння відкритих платформ автоматизації

Платформа	**Призначен ня**	**Тип збережених знань**	**Наявність рекомен-дацій**	**Рівень автома-тизації**	**Обмеження**
Ansible Galaxy	каталог ролей і шаблонів Ansible	декларативні YAML-файли	відсутня	висока	не перевіряє сумісність ролей
Terraform Registry	каталог модулів IaC	конфігураційні блоки HCL	відсутня	середня	немає автомати-чної перевірки залеж-ностей
LinuxServer. io	готові контейнери Docker	метадані та опис компонентів	частково	висока	обмежено контейнер-ними середови-щами
Пропоно-вана система	автоматизо-ваний вибір компоненті в Linux	граф зв’язків і критерії	наявна	дуже висока	потребує поперед-нього навчання бази знань

Проведений аналіз показує, що існуючі рішення переважно орієнтовані на зберігання шаблонів, модулів або контейнерів, проте не забезпечують інтелектуального підбору компонентів. Вони не враховують взаємозалежності між елементами системи, несумісності або конфлікти версій, а також не здійснюють багатокритеріальну оцінку альтернатив.

Це підтверджує доцільність розробки окремої дорадчої системи, яка використовуватиме модель знань та механізм прийняття рішень для **оптимального вибору компонентів Linux-інфраструктури** з урахуванням

продуктивності, безпеки, ресурсних вимог і особливостей сценарію використання.

1.6. Формалізація знань у вигляді графових моделей

Ефективна робота дорадчої системи неможлива без чіткого формалізованого опису знань про серверні компоненти та їх взаємозалежності. У традиційних системах конфігураційні дані часто подаються у вигляді реляційних таблиць або ієрархічних структур. Проте такі моделі обмежено відображають складні зв’язки між елементами, які є характерними для Linux-інфраструктур, де компоненти можуть залежати один від одного, бути несумісними або вимагати певних параметрів оточення [14].

У реальних умовах взаємодія серверних компонентів є багаторівневою: наприклад, вибір версії ядра може впливати на сумісність СУБД, а та, у свою чергу, – на підтримувані модулі чи бібліотеки. Моделі «дерево» або «таблиця» складно масштабувати в таких сценаріях, оскільки кількість залежностей швидко зростає. Тому дедалі частіше для опису подібних предметних областей використовують графові моделі знань.

Графова модель знань (knowledge graph) подає предметну область у вигляді вузлів (entities), що відповідають об’єктам системи, та ребер (relationships), які описують відношення між ними [15]. Кожний елемент може мати набір властивостей, що характеризують його технічні параметри, обмеження чи функціональні можливості.

Використання графа для опису знань має низку переваг:

1. Природне відображення залежностей

У графі зв’язки типу «потребує» (REQUIRES), «несумісний із» (INCOMPATIBLE_WITH), «забезпечує» (PROVIDES) або «належить до» (BELONGS_TO) описуються без складних об’єднань таблиць. Це робить структуру зрозумілою та зручною для аналізу.

Наприклад, можна одразу визначити, що певний веб-сервер несумісний із версією бібліотеки libssl, а певна СУБД вимагає конкретну версію ядра Linux.

2. Ефективність обробки запитів

Реляційні бази даних потребують послідовного виконання JOIN-операцій, які ускладнюються зі збільшенням кількості залежностей. У графових базах даних, таких як Neo4j, використовується мова Cypher, яка оптимізована для обходу графів та пошуку шляхів. Це значно прискорює аналіз складних конфігурацій та виявлення непрямих залежностей.

3. Пошук прихованих зв’язків

Граф дозволяє застосовувати алгоритми DFS, BFS або пошук найкоротшого шляху, щоб знаходити не лише прямі, а й опосередковані залежності. Завдяки цьому система може виявляти складні конфлікти, що виникають лише при поєднанні кількох компонентів.

Наприклад, якщо веб-сервер залежить від певної бібліотеки, а ця бібліотека несумісна з версією СУБД, то граф дозволить виявити конфлікт ще до етапу розгортання.

4. Семантична насиченість моделі

Ребра графа можуть містити додаткові властивості – ваги, параметри продуктивності, рівень сумісності, вказівки щодо пріоритетів. Це дає можливість проводити як якісний, так і кількісний аналіз конфігурацій. Наприклад, ребро «HAS_METRIC» може містити атрибути latency, throughput чи споживання пам’яті.

Таблиця 1.4

Порівняння моделей представлення знань

Модель	**Тип зв'язків**	**Приклади реалізації**	**Переваги**	**Обмеження**
Реляційна	Фіксовані (JOIN)	MySQL, PostgreSQL	Стандартизована, зрозуміла	Складна при моделюванні глибоких залежностей
Ієрархічна	Батько–нащадок	LDAP, XML	Простота побудови	Не підтримує множинні зв'язки
Графова	Вузли та ребра	Neo4j, OrientDB	Гнучкість, швидкість обходу, природність моделювання	Потребує спеціальних алгоритмів аналізу

Використання графової моделі знань забезпечує:

- високу гнучкість та масштабованість;
- швидке формування рекомендацій;
- можливість виявлення як прямих, так і прихованих залежностей;
- точніший аналіз конфігурацій серверних компонентів;
- природний спосіб представлення складних DevOps-середовищ.

Завдяки цим властивостям графова модель виступає оптимальною основою для реалізації інтелектуальної дорадчої системи, яка має підтримувати вибір компонентів Linux-інфраструктури з урахуванням взаємозалежностей і вимог конкретного сценарію.

У наступному розділі буде розглянуто структуру бази знань, архітектуру системи та механізми реалізації алгоритмів рекомендацій.

1.7. Висновки до розділу 1

У першому розділі проведено теоретичний аналіз підходів до побудови та оптимізації серверної інфраструктури на базі операційної системи Linux.

Проаналізовано поняття дорадчих систем, їх структуру та принципи роботи. Окреслено ключові типи DSS – експертні, статистичні та гібридні – та визначено їх придатність для задач автоматизованого підбору компонентів серверної інфраструктури. Показано, що для таких сценаріїв найбільш ефективним є поєднання rule-based логіки та методів багатокритеріального оцінювання.

Розглянуто архітектурні моделі автоматизації DevOps-процесів, включаючи pull- та push-підходи, IaC-технології та системи контейнеризації й оркестрації. Встановлено, що наявні інструменти забезпечують високий рівень автоматизації, проте залишають відкритим питання раціонального вибору оптимальної конфігурації.

Проведено огляд існуючих платформ автоматизації (Ansible Galaxy, Terraform Registry, LinuxServer.io) і встановлено, що їх функціональність спрямована на зберігання та повторне використання конфігурацій, але вони не враховують взаємозалежності між компонентами та не пропонують механізмів формування рекомендацій.

Обґрунтовано доцільність використання графових моделей знань для опису залежностей між компонентами Linux-інфраструктури. Показано, що графова модель забезпечує гнучкість, масштабованість та високу ефективність пошуку взаємозв’язків, що є важливим для формування обґрунтованих рекомендацій.

Сукупність викладених теоретичних положень є основою для формування структури бази знань, побудови алгоритмів рекомендацій та реалізації прототипу дорадчої системи, що розглядається у наступних розділах роботи.

РОЗДІЛ 2. АНАЛІЗ ЗАДАЧІ ТА ПРОЄКТУВАННЯ ПРОГРАМНОЇ ДОРАДЧОЇ СИСТЕМИ ВИБОРУ КОМПОНЕНТІВ ОС LINUX

2.1. Постановка задачі та визначення вимог до системи

Зростання масштабів серверних інфраструктур, збільшення кількості сервісів та ускладнення цифрових систем призводять до необхідності більш ефективної автоматизації процесів їх розгортання та управління. У сучасних компаніях одночасно функціонують десятки компонентів – від веб-серверів і систем керування базами даних до брокерів повідомлень, сервісів моніторингу та балансувальників навантаження. Обсяги даних, які обробляються такими системами, щороку збільшуються, а швидкість розгортання нових інстансів повинна відповідати потребам безперервного розгортання та оновлення.

За даними аналітичних звітів Gartner та IDC, у корпоративних середовищах кількість віртуальних машин і контейнерів зростає на 25–30 % щороку, тоді як середній життєвий цикл окремого сервісу скоротився до менш ніж доби. Це створює запит на інструменти, здатні забезпечити швидкий і надійний вибір оптимальних конфігурацій без ручного втручання.

Попри широке використання сучасних інструментів автоматизації (Ansible, Terraform, Puppet, SaltStack), значна частина рішень щодо вибору конкретних компонентів, їх версій та параметрів сумісності залишається на розсуд адміністратора. У складних інфраструктурах це породжує типові проблеми: невідповідні конфігурації, дублювання ролей, підвищені витрати на супровід та ризики появи помилок через людський фактор. Навіть досвідчені фахівці витрачають значний час на узгодження залежностей між компонентами, оскільки відсутній централізований механізм, який би аналізував взаємозв’язки та пропонував оптимальні рішення.

Ситуацію ускладнює постійне зростання масштабів програмних репозиторіїв: у популярних дистрибутивах Linux кількість доступних пакетів становить 60–80 тисяч найменувань. Залежності між ними утворюють

багаторівневу мережу зв'язків, що робить вибір навіть базових компонентів (веб-сервер, СУБД, кеш, моніторинг) складною багатокритеріальною задачею. Окрім продуктивності, потрібно враховувати стабільність, сумісність, безпеку, ресурсоємність та можливість спільної роботи з іншими модулями системи.

Одним із підходів до розв'язання цієї проблеми є створення дорадчої системи (Decision Support System, DSS), яка на основі формалізованої бази знань та алгоритмів логічного виведення здатна автоматично формувати рекомендації щодо складу серверної інфраструктури. Така система повинна не лише накопичувати дані про компоненти, але й аналізувати залежності, виявляти конфлікти та пропонувати найбільш раціональні комбінації у конкретному контексті розгортання.

Метою цієї роботи є розробка та дослідження програмної дорадчої системи вибору оптимального набору компонентів операційної системи Linux при розгортанні серверної інфраструктури. Система повинна поєднувати знання про DevOps-підходи, типові серверні ролі (Web, DB, Cache, Monitoring) та критерії ефективності, формуючи узгоджені рекомендації для адміністратора.

Для досягнення поставленої мети необхідно виконати такі завдання:

1. Проаналізувати існуючі методи автоматизації розгортання та вибору компонентів серверних систем.

2. Обґрунтувати вибір графової моделі знань для представлення залежностей між елементами інфраструктури.

3. Розробити структуру бази знань, яка описує типи компонентів, правила сумісності та критерії відбору.

4. Реалізувати модуль логічного виведення та формування рекомендацій.

5. Створити підсистему генерації конфігураційних шаблонів (Ansible, Terraform).

6. Провести тестування працездатності системи та оцінити якість рекомендацій за визначеними критеріями.

Очікувані результати:

Розроблена система має забезпечувати виконання таких функцій:

– прийом вхідних параметрів (тип застосунку, обмеження ресурсів, пріоритети користувача);

– пошук та оцінку сумісних компонентів на основі графової моделі знань;

– формування структурованих рекомендацій у форматах JSON або YAML;

– виявлення конфліктів і несумісностей версій;

– генерацію конфігураційних шаблонів для подальшого розгортання інфраструктури.

До програмної системи висуваються такі вимоги:

– застосування графової бази знань Neo4j;

– реалізація логічного механізму мовою Python із використанням FastAPI;

– можливість розгортання у Docker-середовищі;

– підтримка REST API для інтеграції з DevOps-процесами;

– ведення журналів подій та збереження результатів у форматі JSON.

У результаті виконання роботи повинна бути створена програмна дорадча система, здатна формувати рекомендації щодо складу компонентів Linux-інфраструктури та оцінювати їх ефективність за показниками продуктивності, стабільності та сумісності.

Очікується визначення таких характеристик:

– точність рекомендацій для різних наборів критеріїв;

– середній час формування результатів;

– частка виявлених конфліктів залежностей;

– ефективність автоматизованого розгортання порівняно з ручним налаштуванням;

– вплив системи на скорочення часу підготовки серверної конфігурації.

Отримані результати можуть бути використані для побудови систем автоматичного конфігурування серверів, у навчальних DevOps-лабораторіях, а також у корпоративних середовищах, де важливо зменшити ризик помилок і підвищити ефективність розгортання.

Розробка дорадчої системи вибору оптимального набору компонентів Linux є актуальним напрямом розвитку сучасних інформаційних технологій, що

поєднує аналіз знань, автоматизацію та підвищені вимоги до якості серверних конфігурацій.

2.2. Вибір методології представлення знань та алгоритмів рекомендацій

Ключовим завданням під час проєктування дорадчої системи є вибір методології представлення знань, яка дозволить коректно моделювати структуру Linux-інфраструктури та взаємозв'язки між її компонентами. Від того, наскільки вдало обрано підхід до збереження й обробки знань, залежить швидкість формування рекомендацій, масштабованість системи та точність визначення сумісних конфігурацій.

У DevOps-середовищах знання про компоненти мають динамічний характер. Зміна версії пакета, оновлення залежностей або зміна параметрів безпеки можуть впливати на сумісність з іншими елементами системи. Тому модель подання знань повинна бути одночасно структурованою та гнучкою, забезпечувати можливість постійного розширення й адаптації відповідно до змін у програмному забезпеченні.

2.2.1. Критерії вибору методології

Для коректної роботи дорадчої системи методологія представлення знань повинна відповідати таким критеріям:

1. Природне відображення залежностей між компонентами.

Компоненти Linux-інфраструктури мають як прямі залежності (наприклад, Nginx → OpenSSL), так і опосередковані, які проявляються через сумісність версій, наявність певних бібліотек, модульну структуру чи вимоги до оточення. Обрана модель повинна підтримувати багаторівневі та нелінійні зв'язки.

2. Швидке виконання запитів.

Формування рекомендацій має відбуватися в реальному часі – без затримок, з можливістю обробки великих обсягів даних. Це особливо актуально при роботі з базами, що містять сотні тисяч вузлів.

3. Масштабованість бази знань.

Система має дозволяти додавання нових компонентів, їх властивостей і зв'язків без суттєвої зміни схеми та внутрішньої логіки.

4. Інтерпретованість результатів.

Рішення, яке пропонує система, повинне бути зрозумілим для користувача. Адміністратор має мати можливість переглянути «логіку» вибору: на основі яких зв'язків і критеріїв був обраний той чи інший компонент.

Відповідно до цих вимог було розглянуто три основні підходи до подання знань: експертні системи, rule-based моделі та графові структури.

2.2.2. Порівняння підходів

Підходи до представлення знань мають різну природу, що впливає на їх застосовність у задачах моделювання інфраструктури. Узагальнені особливості наведено у таблиці 2.1.

Таблиця 2.1

Порівняння підходів до представлення знань

Підхід	Опис	Переваги	Недоліки
Експертні системи	Знання подані у вигляді набору правил типу «якщо – то», створених фахівцем	Висока інтерпретованість; простота реалізації для вузьких задач	Низька масштабованість; складність підтримки при великій кількості правил
Rule-based моделі	Поєднання логічних правил із формальними критеріями та вагами	Можливість врахування кількох факторів; зрозумілість результатів	Недостатня ефективність для складних структур даних; обмежена взаємозалежність правил
Графові моделі	Знання подані у вигляді вузлів (об'єктів) і ребер (зв'язків між ними)	Природне моделювання нелінійних залежностей; висока швидкість обходу графів; легке розширення	Потребують спеціалізованих баз даних і алгоритмів пошуку шляхів

Графовий підхід виявився найбільш придатним, оскільки він дозволяє описувати складні нелінійні залежності, які часто присутні у Linux-середовищах. Наприклад, у випадку реляційної чи rule-based моделей важко формалізувати взаємозалежність між ядром Linux, бібліотекою glibc та контейнерним модулем, тоді як у графовій моделі такі зв'язки відображаються природно, через послідовність вузлів і ребер.

Окрім цього, графові бази даних підтримують ефективні алгоритми обходу та пошуку шляхів (DFS, BFS, shortest path), що дає змогу швидко виявляти конфлікти сумісності або знаходити альтернативні конфігурації.

2.2.3. Обґрунтування вибору графової моделі Neo4j + rule-engine

На основі проведеного аналізу для реалізації бази знань обрано графову СУБД Neo4j, яка забезпечує:

– гнучке представлення об'єктів і зв'язків у вигляді вузлів (nodes) та ребер (relationships);

– використання мови запитів Cypher, що дозволяє виконувати пошук залежностей, оцінювати вагу зв'язків та виявляти цикли;

– високу продуктивність при виконанні операцій типу «знайти всі сумісні компоненти» або «визначити конфлікти між версіями».

Для формування рекомендацій застосовується rule-engine – модуль логічного виведення, який використовує набір правил і критеріїв для оцінки можливих конфігурацій. У поєднанні з Neo4j це дає змогу реалізувати гібридну модель рекомендацій, у якій:

– Neo4j відповідає за структуру даних, побудову графа та пошук залежностей;

– rule-engine оцінює знайдені комбінації за ваговими коефіцієнтами критеріїв (продуктивність, стабільність, безпека, ресурсоємність).

Таке поєднання дозволяє враховувати як жорсткі вимоги (hard constraints), так і м'які критерії (soft metrics), забезпечуючи більш точне та адаптивне формування рекомендацій.

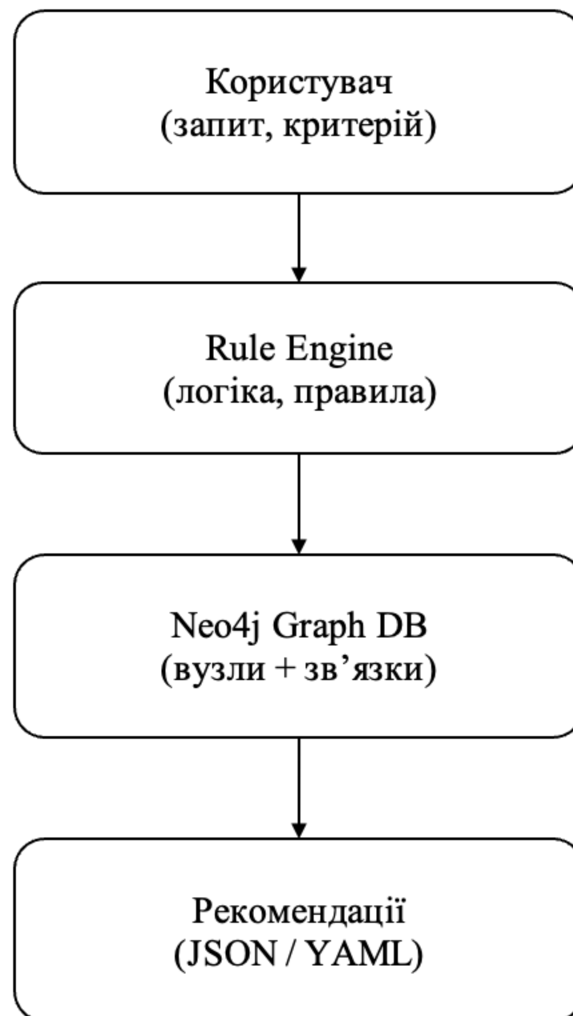


Рис. 2.1 – Узагальнена взаємодія між графовою базою знань і rule-engine.

2.3. Вибір інструментальних засобів розробки

Розроблення дорадчої системи потребує вибору технологічного стеку, який забезпечить високу продуктивність, масштабованість та сумісність із графовою базою знань. На цьому етапі важливо визначити набір інструментів, що дозволить не лише ефективно реалізувати алгоритми логічного виведення, а й забезпечити просту інтеграцію системи у DevOps-процеси з мінімальними витратами на розгортання і подальшу підтримку.

2.3.1. Критерії вибору технологій

Під час вибору технологічного стеку враховуються такі ключові критерії:

1. Сумісність із графовими СУБД

Мова програмування та фреймворк повинні мати нативні драйвери або стабільні бібліотеки для роботи з Neo4j чи іншими графовими базами даних. Це дозволяє уникнути використання проміжних адаптерів і зменшує затримки при обробці запитів.

2. Продуктивність під час роботи з великими обсягами даних

Оскільки система повинна виконувати запити до графа, який містить тисячі або сотні тисяч зв'язків, технологія має забезпечувати паралельну обробку, ефективне використання ресурсів і швидке повернення результатів.

3. Зручність розгортання у DevOps-середовищі

Обрані інструменти мають бути сумісні з Docker, Kubernetes, CI/CD-конвеєрами, підтримувати автоматизоване оновлення та мати передбачувану модель розгортання.

4. Активна спільнота та наявність бібліотек

Використання технологій із великою кількістю готових рішень, документації та прикладів значно зменшує час розробки та полегшує усунення проблем.

5. Відкритість ліцензії

Для академічних і корпоративних проєктів важливо застосовувати інструменти з відкритими ліцензіями (MIT, Apache 2.0, BSD), що дозволяє уникати додаткових витрат та обмежень.

2.3.2. Огляд популярних стеків DevOps-систем

Було проведено порівняльний аналіз найбільш поширених стеків, що використовуються у DevOps-практиках. Особливу увагу приділено поєднанню мови програмування, веб-фреймворку та типу бази даних.

Таблиця 2.2

Порівняння технологічних стеків

Стек	Мова	Переваги	Обмеження
Python + FastAPI + Neo4j	Простота, гнучкість, велика кількість бібліотек	Висока продуктивність при правильній оптимізації	Однопотоковість інтерпретатора
Go + Fiber + Neo4j	Компільований код, швидкість	Менше бібліотек для графів	Вищий поріг входу
Node.js + NestJS + ArangoDB	Асинхронність, JSON-орієнтованість	Високе навантаження на пам’ять	Обмежена підтримка складних запитів

Результати аналізу свідчать, що стек **Python + FastAPI + Neo4j** забезпечує найбільш збалансоване поєднання продуктивності, швидкості розробки та сумісності з графовими моделями.

Python має широку екосистему бібліотек для роботи з даними (pandas, networkx, numpy), що дозволяє реалізувати логічний та аналітичний рівні системи.

FastAPI забезпечує високу продуктивність асинхронного REST-API, підтримує автоматичну генерацію документації та добре інтегрується з DevOps-процесами.

2.3.3. Обґрунтування вибору Python + FastAPI + Neo4j

Обраний технологічний стек дозволяє реалізувати всі функціональні вимоги системи з мінімальними витратами на розробку й подальшу експлуатацію.

1. Зручна інтеграція з Neo4j через Py2neo та офіційні драйвери

Python має стабільні бібліотеки для роботи з Neo4j, що дозволяють виконувати запити мовою Cypher, створювати графи, оновлювати вузли та працювати зі складними структурами без необхідності низькорівневого коду.

2. Високопродуктивний API-фреймворк FastAPI

FastAPI використовує асинхронну модель обробки запитів і здатний обслуговувати великий потік звернень. Підтримка OpenAPI/Swagger полегшує інтеграцію з CI/CD та іншими сервісами.

3. Просте та швидке контейнерне розгортання

Python та FastAPI легко упаковуються у Docker-контейнери. Розгортання системи у хмарному або локальному середовищі займає мінімальний час і не потребує складних налаштувань.

4. Використання шаблонізатора Jinja2

Jinja2 дозволяє генерувати YAML- і JSON-конфігурації для Ansible та Terraform, що спрощує автоматичне формування файлів розгортання на основі рекомендацій.

5. Кросплатформеність та відкриті ліцензії

Python, FastAPI та Neo4j мають відкриті ліцензії й працюють у будь-якому середовищі (Linux, Windows, macOS), що робить стек універсальним для академічних та корпоративних проєктів.

Таким чином, Python + FastAPI + Neo4j вибрано як базовий технологічний фундамент системи. Він забезпечує сумісність із графовою моделлю знань, високу швидкість обробки запитів, зручність контейнерного розгортання та доступність готових програмних рішень.

2.4. Архітектурне проєктування системи

Архітектурне проєктування є одним із ключових етапів розроблення програмної системи, оскільки саме на цьому рівні визначаються структура компонентів, принципи їх взаємодії та підходи до забезпечення масштабованості. Для дорадчої системи вибору компонентів ОС Linux архітектура має інтегрувати модуль логічного виведення, графову базу знань і модуль генерації конфігурацій, забезпечуючи узгоджений обмін даними між ними.

Метою архітектурного проєктування є створення гнучкої моделі, здатної стабільно працювати за умов збільшення обсягів даних, кількості компонентів та зростання числа користувачів. Архітектура має забезпечити розширюваність, підтримуваність і можливість інтеграції з DevOps-процесами.

2.4.1. Загальна структура та рівні системи

Загальна архітектура системи передбачає поділ на п’ять основних рівнів, які взаємодіють між собою через визначені інтерфейси:

1. Рівень інтерфейсу користувача (User Interface Layer).

Забезпечує взаємодію користувача із системою через CLI або веб-інтерфейс. На цьому рівні задаються критерії пошуку, пріоритети та параметри інфраструктури.

2. Модуль обробки запитів (Recommendation Engine Layer).

Виконує логічне виведення, здійснює аналіз даних із графової бази та застосовує правила й вагові коефіцієнти для формування рекомендацій.

3. Модуль критеріїв і правил (Criteria Layer).

Містить формалізовані критерії оцінювання (продуктивність, стабільність, сумісність, безпека) та забезпечує їх використання у процесі ранжування компонентів.

4. База знань (Knowledge Graph Layer).

Реалізована на основі графової СУБД Neo4j. Містить вузли компонентів, дані про їх властивості та ребра, що описують залежності, конфлікти та відношення сумісності.

5. Модуль генерації шаблонів (Template Generator Layer).

Формує вихідні конфігураційні файли (Ansible, Terraform), що можуть бути використані для подальшого автоматичного розгортання інфраструктури.

Поділ системи на рівні забезпечує чітку логіку обробки запитів, спрощує відлагодження та дозволяє незалежно модернізувати окремі частини.

2.4.2. Модульна архітектура програмного комплексу

Система має модульну архітектуру, що забезпечує незалежний розвиток та тестування кожного компонента. Такий підхід дозволяє впроваджувати нові функції без істотних змін в інших частинах системи.

Основні модулі:

– InputHandler – приймає запит користувача, перевіряє коректність параметрів та передає їх далі в систему.

– RecommendationEngine – застосовує правила логічного виведення, отримує дані з графової бази та визначає перелік потенційно сумісних компонентів.

– CompatibilityChecker – перевіряє, чи не мають обрані компоненти конфліктів між собою (наприклад, несумісні версії бібліотек).

– CriteriaModule – обробляє вагові коефіцієнти критеріїв та формує підсумкову оцінку компонентів.

– TemplateGenerator – створює конфігураційні шаблони на основі обраної конфігурації.

– Logger – фіксує виконані запити, результати та службову інформацію з метою подальшого аналізу.

Модульна організація дозволяє легко розширювати можливості системи, додаючи нові критерії чи типи компонентів без зміни логіки роботи інших модулів.

2.4.3. Логіка обміну даними між модулями

Передача даних між модулями здійснюється у форматі JSON через REST API.

Типовий сценарій взаємодії включає такі кроки:

– користувач формує запит із параметрами;

– Input Handler - перетворює запит у структурований JSON-об’єкт;

– Recommendation Engine - виконує запит до Neo4j;

– Criteria Module - ранжує результати за ваговими коефіцієнтами;

Template Generator - створює конфігураційні файли;

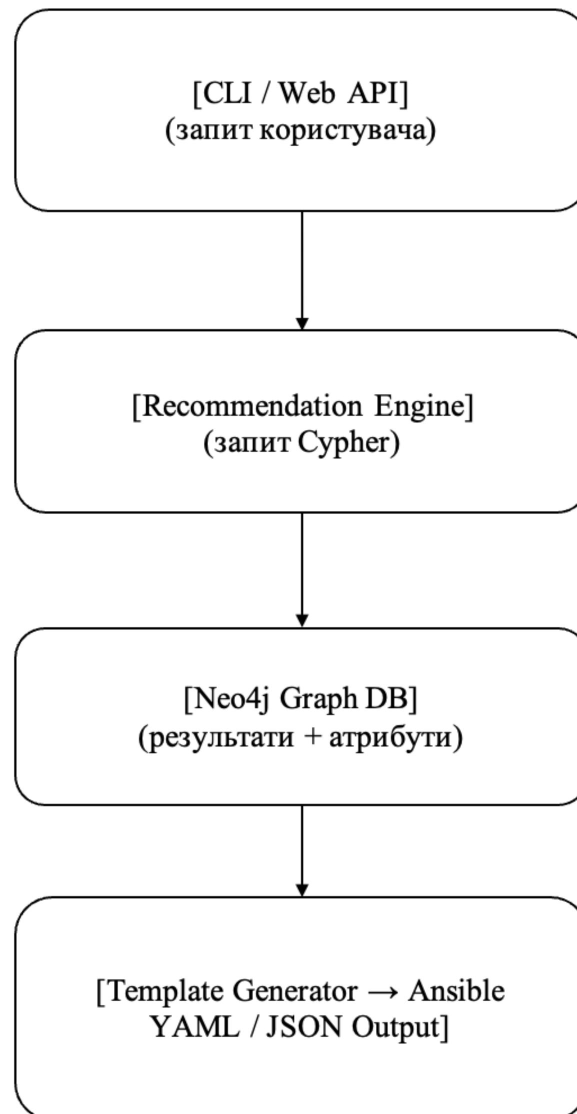


Рис. 2.2 – Логіка взаємодії між компонентами системи.

Така схема забезпечує відокремлення бізнес-логіки від рівня даних, що підвищує гнучкість і дозволяє інтегрувати систему у CI/CD-конвеєри та інші DevOps-інструменти (Jenkins, GitLab CI/CD, Terraform Cloud).

2.4.4. Архітектурні альтернативи (моноліт vs мікросервіси)

Під час проєктування було розглянуто дві архітектурні парадигми: монолітну та мікросервісну.

Таблиця 2.3

Порівняння архітектурних підходів

Критерій	**Монолітна архітектура**	**Мікросервісна архітектура**
Продуктивність	Вища при невеликій кількості користувачів	Краще масштабується при великих навантаженнях
Розгортання	Просте, однокомпонентне	Потребує контейнеризації та оркестрації
Гнучкість оновлення	Менша, зміни впливають на всю систему	Висока, оновлюється лише один сервіс
Залежності	Внутрішні, мінімальні	Можливі проблеми синхронізації
Контроль версій	Єдиний цикл релізу	Незалежні релізи для кожного сервісу

2.4.5. Вибір архітектури та обґрунтування рішень

З урахуванням вимог до продуктивності, простоти впровадження та масштабу системи обрано монолітну архітектуру з модульною внутрішньою структурою. Такий підхід забезпечує:

- швидке впровадження та налагодження;
- зручність використання в навчальних і дослідницьких середовищах;
- мінімальні вимоги до інфраструктури розгортання;
- відсутність потреби у складній оркестрації контейнерів.

Водночас модульна організація дозволяє у майбутньому перейти до мікросервісної архітектури без повного перепроєктування системи. Наприклад, окремі функціональні частини – RecommendationEngine або TemplateGenerator – можуть бути винесені у самостійні сервіси за потреби масштабування.

Таким чином, архітектура дорадчої системи є збалансованою між простотою реалізації та потенціалом подальшого розвитку, забезпечуючи

стабільну роботу системи в процесі аналізу, вибору та автоматизації розгортання компонентів ОС Linux.

2.5. Модель обміну даними між компонентами системи

Ефективна взаємодія між модулями дорадчої системи є ключовою умовою її продуктивності, стабільності та можливості масштабування. Оскільки система складається з кількох логічно відокремлених компонентів – модуля рекомендацій, графової бази знань, модуля критеріїв та генератора шаблонів – важливо забезпечити узгоджений, стандартизований та надійний обмін даними між ними.

Обрана модель взаємодії базується на принципі клієнт–серверної архітектури. Це дозволяє чітко розмежувати відповідальність між компонентами, а також спрощує розгортання і можливість подальшого розширення системи.

2.5.1. Основні принципи взаємодії

Модель обміну даними побудована на основі асинхронної передачі запитів та відповідей у стандартизованому форматі JSON.

Користувач взаємодіє із системою через CLI або веб-інтерфейс і передає початкові параметри: тип задачі, вимоги до серверної ролі, критерії оцінювання (продуктивність, безпека, стабільність). Після цього модуль рекомендацій формує запит до графової бази знань, застосовує правила логічного виведення та повертає структурований результат.

Основні принципи взаємодії:

● **Уніфікованість форматів даних.** Усі модулі використовують однакову структуру JSON, що забезпечує сумісність і спрощує інтеграцію.

● **Асинхронність.** Система підтримує паралельне обслуговування кількох користувачів, не блокуючи основний потік обробки.

● **Розмежування рівнів.** Інтерфейс, бізнес-логіка та шар даних працюють незалежно один від одного, що спрощує тестування, масштабування та внесення змін.

● **Журналювання.** Усі операції логуються для подальшого аналізу, що підвищує прозорість роботи системи та полегшує діагностику.

2.5.2. Сценарій 1 – CLI ↔ API ↔ Neo4j

У базовій конфігурації система працює за моделлю клієнт–сервер, де користувач взаємодіє з командним інтерфейсом (CLI), який викликає REST API на основі FastAPI.

Послідовність взаємодії:

1. CLI формує запит і надсилає його до REST API.
2. API передає параметри в Recommendation Engine.
3. Recommendation Engine виконує запити до Neo4j, аналізує залежності та повертає список сумісних компонентів.
4. Модуль критеріїв оцінює варіанти і формує фінальну рекомендацію.
5. Результат повертається користувачеві у форматі JSON.

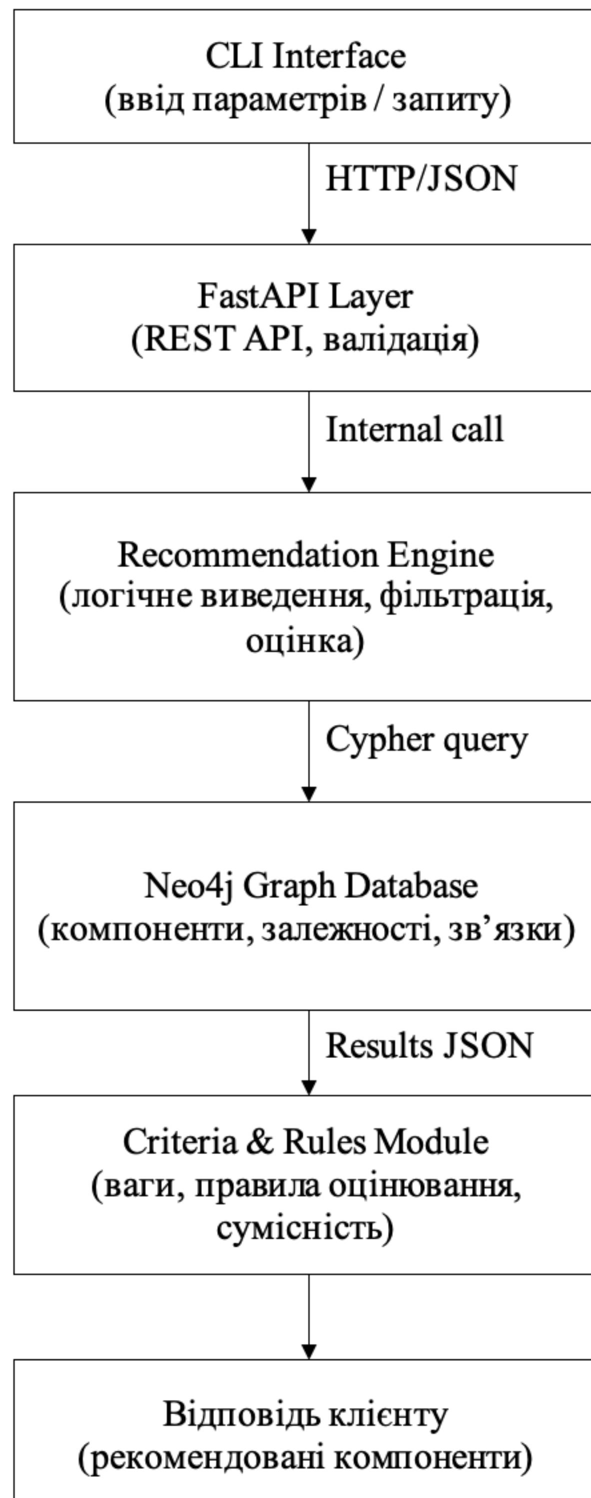


Рис. 2.3 – Модель обміну даними між CLI, API та базою знань.

Перевагою цього підходу є можливість роботи як з локальними інтерфейсами, так і з зовнішніми сервісами, що використовують REST API. Така модель природно інтегрується у CI/CD-середовища і дозволяє запускати сценарії автоматично.

2.5.3. Сценарій 2 – Інтеграція з Ansible / Terraform

Однією з важливих функцій системи є можливість інтеграції з DevOps-інструментами автоматизації. Після отримання рекомендацій Template Generator створює готові шаблони:

- для Ansible – файл playbook.yaml з секціями roles, tasks, vars;
- для Terraform – модуль .tf, який містить ресурси, змінні та провайдери.

Система виконує роль аналітичного шару перед безпосереднім розгортанням, визначаючи оптимальний набір компонентів та формуючи конфігурації, сумісні з існуючими DevOps-конвеєрами.

Цей підхід:

- зменшує час підготовки середовищ;
- усуває помилки, пов'язані з людським фактором;
- підвищує відтворюваність процесу розгортання.

2.5.4. Сценарій 3 – Docker-середовище

Для зручності розгортання всі модулі системи контейнеризовані. Використання Docker Compose дозволяє запускати систему командою docker-compose up, об'єднавши FastAPI, Neo4j, Recommendation Engine та Template Generator в єдиний стек.

Переваги контейнеризації:

- Портативність. Система може бути розгорнута на будь-якій платформі без додаткових налаштувань.
- Ізоляція. Кожен модуль працює у власному контейнері, що запобігає конфліктам версій.
- Масштабованість. Кількість контейнерів можна збільшувати залежно від навантаження (Docker Swarm, Kubernetes).
- Простота оновлень. Оновлення окремого компонента не потребує зупинки всієї системи.

Контейнерний підхід також створює можливості для майбутньої інтеграції з хмарними сервісами (AWS, GCP, Azure).

2.5.5. Механізми кешування та логування

Для підвищення продуктивності та забезпечення прозорості роботи у системі реалізовано механізми кешування та логування.

Повторні запити з однаковими параметрами обробляються через кеш-пам'ять, що:

- знижує навантаження на Neo4j,
- скорочує середній час відповіді у 2–3 рази,
- підвищує стійкість системи при пікових навантаженнях.

Кеш автоматично оновлюється після зміни структури бази знань або правил.

Усі запити, відповіді, винятки та статистика обробки записуються у журнали. Формат JSON дозволяє легко аналізувати дані за допомогою Grafana, ELK Stack або подібних систем моніторингу.

Модель обміну даними в дорадчій системі базується на принципах модульності, стандартизації та розширюваності. Вона забезпечує узгоджену роботу всіх компонентів, підтримує інтеграцію з DevOps-процесами, дозволяє працювати у контейнерних середовищах і створює передумови для масштабованості та розвитку системи у майбутньому.

2.6. Функціональна модель системи

Функціональна модель визначає послідовність взаємодій між користувачем та внутрішніми складовими дорадчої системи, а також логіку виконання основних процесів. Вона описує повний цикл роботи системи – від введення параметрів користувачем до формування готових конфігурацій для розгортання інфраструктури.

Побудова функціональної моделі дозволяє формалізувати поведінку системи, визначити інформаційні потоки, а також окреслити точки розширення, що дають змогу додавати нові можливості без зміни основної архітектури.

Основні процеси, реалізовані системою, включають такі етапи:

1. Формування запиту користувачем.

Користувач задає початкові параметри, які описують тип застосунку, вимоги до продуктивності чи безпеки, обмеження щодо ресурсів та інші критерії. Дані передаються до системи у вигляді структурованого запиту через CLI або REST API.

Стандартизований підхід до введення параметрів забезпечує коректність обробки та спрощує інтеграцію з зовнішніми сервісами.

2. Логічне виведення рекомендацій (Rule Engine)

Rule Engine здійснює початкову фільтрацію компонентів згідно з визначеними правилами. На цьому етапі формується множина допустимих кандидатів, яка враховує жорсткі вимоги – наприклад, сумісність категорії компонентів, мінімальні вимоги до ресурсів або обмеження щодо версій.

Результатом роботи цього модуля є попередній список компонентів, які відповідають базовим умовам.

3. Перевірка сумісності у графовій базі знань (Knowledge Graph)

Наступним етапом є аналіз залежностей і виявлення конфліктів у графовій базі знань Neo4j. Система перевіряє такі взаємозв’язки:

- залежності між компонентами;
- несумісність версій;
- непрямі конфлікти, що проявляються через спільні бібліотеки або налаштування;
- обмеження середовища виконання.

У результаті формується підмножина конфігурацій, які є технічно коректними та не містять суперечностей.

4. Обчислення інтегрального показника (Scoring Module)

Для кожного валідного кандидата застосовується система вагових коефіцієнтів, що враховує такі критерії:

- продуктивність;
- стабільність;

- безпека;
- ресурсоємність;
- відповідність цільовому сценарію.

Scoring Module ранжує альтернативи відповідно до заданих пріоритетів. Утворений рейтинг визначає, які компоненти є найбільш придатними для конкретної задачі розгортання.

5. Формування вихідних конфігурацій

Фінальним етапом є створення конфігураційних файлів. Template Generator перетворює отримані рекомендації у готові шаблони:

- у форматі **YAML** – для використання у Ansible;
- у форматі **JSON** або *.tf* – для Terraform і суміжних систем IaC.

Готові конфігурації можуть бути інтегровані у DevOps-конвеєри без додаткового ручного доопрацювання.

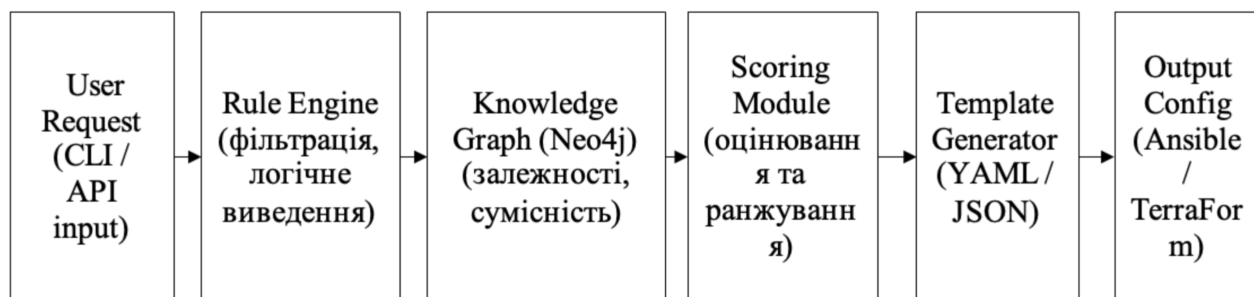


Рис. 2.4 – Узагальнена функціональна модель системи

Система є розширюваною: для додавання нових типів компонентів або критеріїв немає потреби змінювати основні модулі. Достатньо оновити структуру графової бази знань або скоригувати вагові коефіцієнти в модулі критеріїв.

Такий підхід забезпечує гнучкість та адаптивність системи, дозволяє ефективно впроваджувати нові можливості без істотних змін у коді і відповідає потребам сучасних DevOps-інфраструктур.

2.7. Об’єктна модель системи

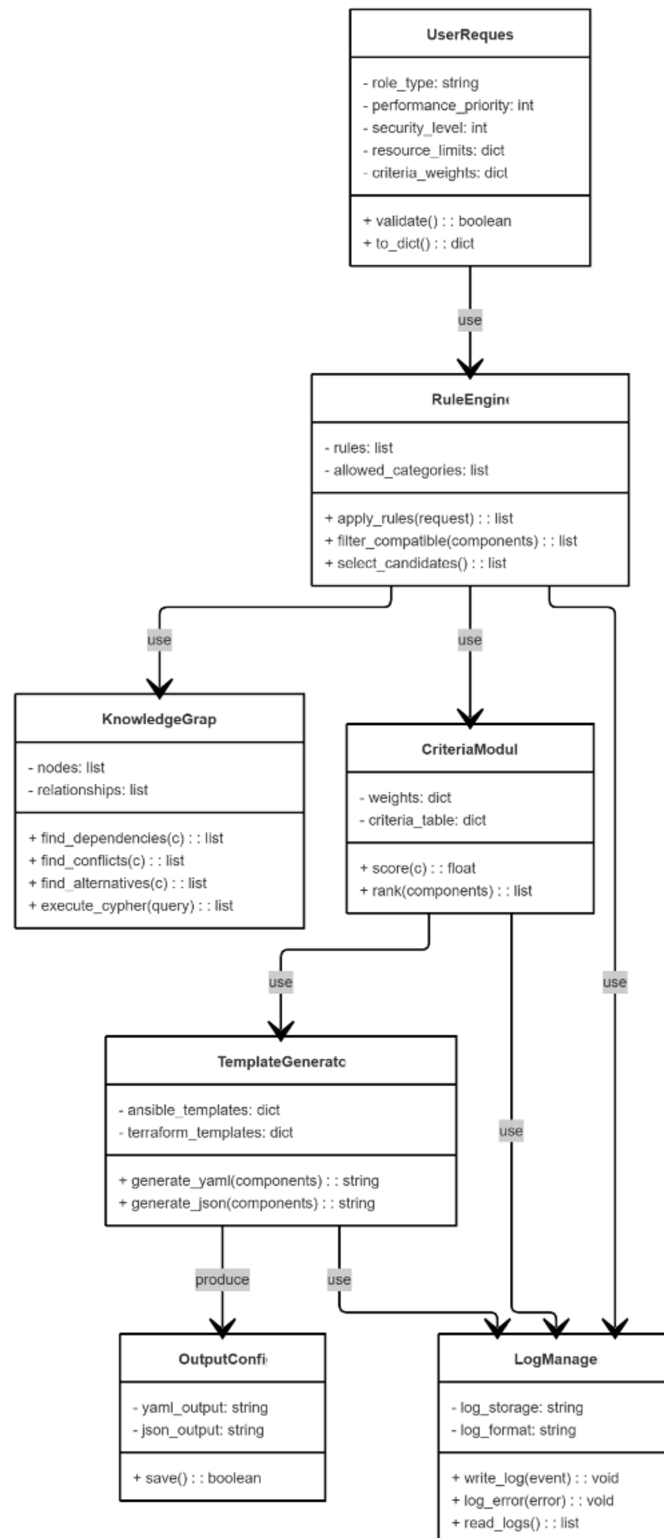


Рис. 2.5 Об’єктно-орієнтована модель

Об’єктна модель визначає ключові сутності дорадчої системи, їх властивості та взаємозв’язки. На відміну від функціональної моделі, яка описує

динаміку процесів, об’єктна модель зосереджується на статичній структурі системи та логічній організації її внутрішніх компонентів.

Побудова об’єктної моделі дозволяє чітко формалізувати дані, що використовуються системою, забезпечити узгодженість між модулями та створити основу для подальшого розширення функціональності. Модель розроблено з урахуванням принципів об’єктно-орієнтованого аналізу (ООА).

Нижче наведено ключові сутності системи.

1. UserRequest (Запит користувача)

Описує початкові параметри, які вводить користувач для формування рекомендацій.

Основні атрибути:

- role_type – тип серверної ролі (web, db, cache тощо);
- performance_priority – пріоритет продуктивності;
- security_level – вимоги до безпеки;
- resource_limits – обмеження за ресурсами (CPU/RAM);
- criteria_weights – вагові коефіцієнти критеріїв.

Основні методи:

- валідація параметрів;
- формування структури запиту до RecommendationEngine.

2. Component (Компонент системи)

Базова сутність графа знань, яка відображає окремий елемент інфраструктури.

Атрибути:

- name – назва компонента (Nginx, PostgreSQL тощо);
- category – тип (web, db, monitoring, security);
- version – версія;
- requirements – вимоги до середовища;
- compatibility_flags – інформація про сумісності й конфлікти.

3. KnowledgeGraph (Граф знань)

Визначає структуру графа, у якому зберігаються всі компоненти та зв’язки між ними.

Основні методи:

- find_dependencies(component) – пошук залежностей;
- find_conflicts(component) – визначення несумісностей;
- find_alternatives(component) – пошук альтернативних варіантів;
- execute_cypher(query) – виконання запиту до Neo4j мовою Cypher.

4. RuleEngine (Модуль логічного виведення)

Відповідає за застосування правил, перевірку відповідності критеріїв та формування початкового набору кандидатів.

Атрибути:

- набір правил (policy set);
- перелік дозволених категорій компонентів.

Методи:

- apply_rules(UserRequest) – застосування правил;
- filter_compatible(components) – фільтрація несумісних компонентів;
- select_candidates() – формування первинного набору кандидатів.

5. CriteriaModule (Модуль критеріїв)

Оцінює компоненти за визначеними критеріями та формує підсумковий рейтинг.

Атрибути:

- criteria_weights – вагові коефіцієнти;
- criteria_table – опис критеріїв оцінювання.

Методи:

- score(component) – обчислення інтегральної оцінки;
- rank(components) – ранжування кандидатів.

6. TemplateGenerator (Генератор шаблонів)

Створює вихідні конфігураційні файли, готові до використання у DevOps-інструментах.

Атрибути:

- шаблони для Ansible;
- шаблони для Terraform.

Методи:

- generate_yaml(components) – формування YAML-конфігурації;
- generate_json(components) – генерація JSON-файлів рекомендацій.

7. LogManager (Журналювання)

Фіксує хід виконання запитів, помилки та підсумкові результати.

Атрибути:

- log_storage – механізм зберігання логів (файли, база, stdout);
- log_format – формат запису (JSON або текстовий).

Методи:

- write_log(event) – запис події;
- read_logs() – доступ до збережених записів.

Об’єктна модель системи формує основу її внутрішньої структури і визначає взаємодію між даними, правилами та механізмами обробки. Завдяки чітко визначеним сутностям і методам система є розширюваною, легко масштабованою та адаптованою до потреб DevOps-процесів. Модель слугує фундаментом для реалізації алгоритмів рекомендацій та забезпечує цілісність роботи всіх модулів.

2.8. Дорадчий модуль системи

Дорадчий модуль є ядром запропонованої системи та ключовим інтелектуальним компонентом, що забезпечує формування обґрунтованих рекомендацій щодо вибору оптимального набору компонентів ОС Linux. На відміну від класичних інструментів автоматизації, які лише відтворюють заздалегідь описані сценарії, дорадчий модуль виконує динамічне прийняття рішень. Це робить його не просто частиною програмної системи, а фактично

експертною підсистемою, що моделює процес мислення інженера DevOps, але у стандартизованому і відтворюваному вигляді.

2.8.1. Концепція та місце дорадчого модуля у системі

У реальному DevOps-середовищі підбір компонентів Linux-інфраструктури – це завдання, яке ніколи не є "механічним". Адміністратор аналізує:

- вимоги до продуктивності,
- обмеження ресурсів,
- ризики безпеки,
- залежності між сервісами,
- можливі конфлікти версій,
- особливості конкретного оточення.

Це означає, що проста вибірка зі списку компонентів не дає коректного результату.

Дорадчий модуль системи відтворює саме евристичне міркування, яке в реальному житті виконує фахівець:

- "якщо ролі потрібна висока продуктивність – пріоритет Nginx;
- якщо сувора безпека – PostgreSQL;
- якщо малоресурсне середовище – опускаємо Apache;
- якщо включаю cache – враховуємо залежність від RAM;
- якщо одночасно web і db – перевіряємо конфлікти портів, вимоги TLS, потреби у сховищі…"

Таким чином, дорадчий модуль – це механізм прийняття рішень (decision-making engine), який ґрунтується на формальних правилах, але працює як інженер, який:

1. інтерпретує вимоги,
2. шукає всі допустимі варіанти,
3. виявляє несумісності,
4. оцінює альтернативи,
5. формує найкращий набір компонентів.

2.8.2. Логічна структура дорадчого модуля

Дорадчий модуль складається з трьох взаємопов'язаних підмодулів:

1. Інтерпретація вимог користувача

На цьому етапі система «перекладає» запит людини у формальний вигляд. Наприклад:

● "хочу web-сервер з мінімальною затримкою" → пріоритет Performance: високий, Security: середній.

● "будь-яка СУБД, але без лишнього навантаження на RAM" → ваги Resource Usage: високі.

Фактично формується профіль очікуваної конфігурації, який стає входом для подальшого аналізу.

2. Підсистема логічного виведення (Rule Engine)

Це серце дорадчого модуля. Rule Engine реалізує набір взаємопов'язаних логічних правил:

● Рольові правила

Визначають, що взагалі можна використовувати для даної ролі.

role = "web" → {Nginx, Apache, Caddy}

role = "db" → {PostgreSQL, MySQL}

● Правила залежностей

Виявляють усе, без чого компонент не працюватиме.

Nginx → requires TLS support

PostgreSQL → requires Persistent storage

● Правила конфліктів

Вони критично важливі. Система повинна гарантувати, що рекомендації не містять взаємно несумісних компонентів.

Nginx conflicts with Apache

MySQL 5.7 conflicts with glibc 2.39

Це один із найскладніших етапів, оскільки конфлікти можуть бути:

- прямими (A забороняє B),
- непрямими (A → requires X, але X → conflicts_with B),

- o контекстними (конфлікт виникає лише за певної ролі).

- Функціональні правила

Вони визначають пріоритети, базуючись на вимогах користувача. Приклад: якщо користувач ставить високу вагу продуктивності:

Performance weight = high → Nginx > Apache

Якщо важлива безпека:

Security weight = high → PostgreSQL > MySQL

3. Модуль оцінювання (Criteria Module)

Після застосування логічних правил залишається кілька можливих конфігурацій. Система повинна обрати найкращу. Тому застосовується багатокритеріальна модель оцінювання:

Score = Wp·Performance + Ws·Stability + Wsec·Security + Wr·ResourceUsage

Це дозволяє зробити:

- конфігурацію передбачуваною,
- результати відтворюваними,
- логіку прозорою.

2.8.3. Алгоритм роботи дорадчого модуля в реальному запиті

Розглянемо реальний сценарій.

Запит:

- Роль: web
- Пріоритет: висока продуктивність
- RAM: обмежено
- Безпека: середня

1. Формування кандидатів:
   - o {Nginx, Apache}
2. Перевірка залежностей через Neo4j:
   - o Nginx → requires openssl
   - o Apache → requires mod_http2
3. Перевірка конфліктів:
   - o Немає

4. Оцінювання за критеріями:

При RAM обмежено:

- ResourceUsage: Nginx > Apache
- Performance: Nginx > Apache

5. Фінальна рекомендація:

- Nginx

Пояснення: найвищий баланс продуктивність/ресурси

2.8.4. Властивість пояснюваності (Explainable Recommendations)

Приклад пояснення:

Nginx рекомендовано тому, що:

- має вищий показник Performance (0.92) порівняно з Apache (0.73);
- споживає менше RAM, що відповідає обмеженням;
- не має конфліктів із іншими вибраними компонентами;
- забезпечує підтримку TLS, згідно з вимогами.

Так працюють сучасні DSS у промисловості.

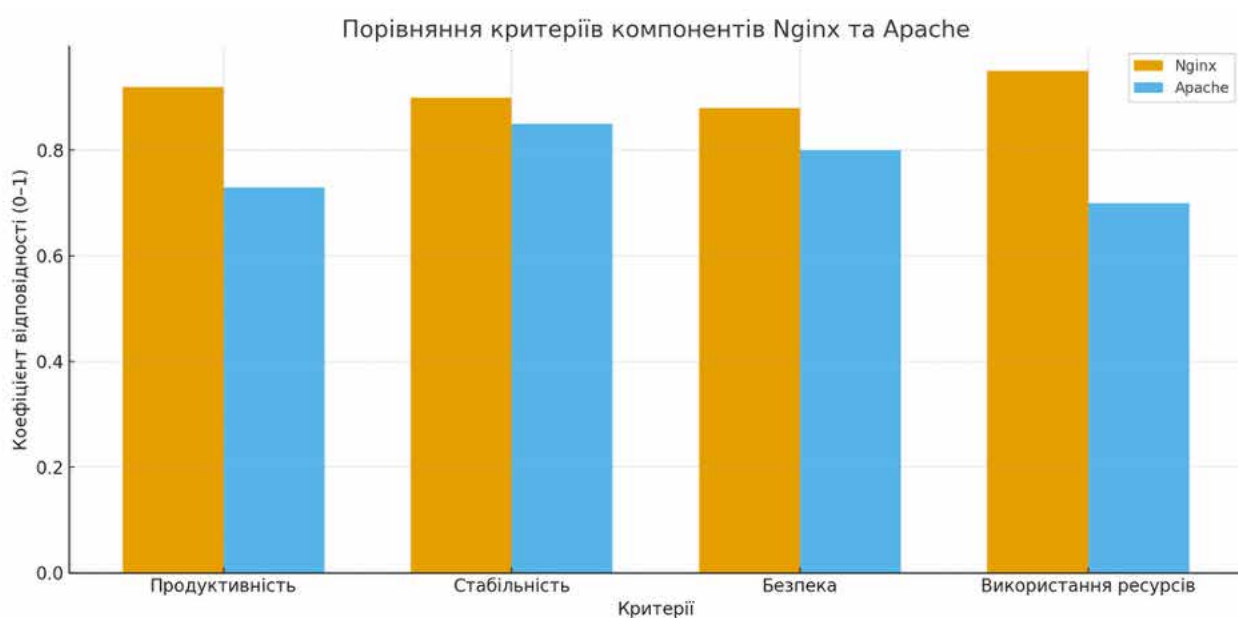


Рис. 2.6 – Порівняння коефіцієнтів відповідності компонентів Nginx та Apache за ключовими критеріями

Графік відображає порівняння двох альтернативних веб-серверів – Nginx та Apache – за чотирма критеріями, що використовуються в моделі

багатокритеріального оцінювання: продуктивність, стабільність, безпека та використання ресурсів.

Показники подано у вигляді коефіцієнтів відповідності (у діапазоні 0–1), що є нормованими величинами, отриманими з бази знань та модулів критеріїв.

З рисунка видно, що Nginx має вищі значення за більшістю критеріїв, зокрема:

- продуктивність – 0.92 проти 0.73 у Apache;
- споживання ресурсів – найвищий коефіцієнт серед альтернатив;
- безпека та стабільність – дещо вищі порівняно з Apache.

Це графічно підтверджує висновок, сформований дорадчим модулем: Nginx є більш оптимальною альтернативою для заданих умов розгортання, а пояснюваність у вигляді порівняльного графіка дозволяє легко інтерпретувати рішення системи та підвищує її прозорість.

2.8.5. Особливості проєктування та новизна

У цьому модулі поєднано:

- rule-based (експертний) підхід,
- графове моделювання залежностей,
- багатокритеріальне оцінювання,
- пояснюваність рішень.

Це дає кілька унікальних властивостей:

- модуль зростає разом із графом знань (не потребує переписування логіки);
- система здатна обробляти конфлікти, які важко помітити вручну;
- підтримує адаптацію до різних типів користувачів (оптимізація “під роль”);
- працює достатньо швидко для інтерактивного використання.

2.9. Висновки до розділу 2

У другому розділі було сформовано теоретичні та проєктні основи програмної дорадчої системи вибору компонентів ОС Linux. На початку розділу визначено постановку задачі, сформульовано вимоги до системи та обґрунтовано необхідність автоматизації процесу вибору конфігурацій у сучасних DevOps-середовищах. Показано, що традиційні підходи до підбору програмних компонентів не враховують складні взаємозалежності, що призводить до помилок конфігурації та збільшення витрат на розгортання інфраструктури.

На основі аналізу методологій представлення знань розглянуто експертні, rule-based та графові моделі. Порівняння показало, що саме графова модель знань у поєднанні з rule-engine є найбільш придатною для опису структурної складності Linux-інфраструктур та реалізації механізму рекомендацій. Обґрунтовано вибір графової СУБД Neo4j та rule-engine як основи для реалізації гібридного підходу до формування конфігурацій.

Проведено вибір інструментальних засобів та сформовано технологічний стек системи. Визначено, що комбінація Python, FastAPI та Neo4j забезпечує оптимальний баланс між продуктивністю, швидкістю розробки, доступністю бібліотек і зручністю розгортання в контейнерних та DevOps-середовищах.

Запропоновано архітектурну модель системи, що складається з кількох рівнів та модулів, які забезпечують обробку запитів, аналіз залежностей, логічне виведення, оцінювання альтернатив і формування готових конфігурацій. Окремо розглянуто модель обміну даними між компонентами на основі REST API, а також механізми кешування, журналювання та інтеграції з Ansible і Terraform.

Об’єктна модель системи описує ключові сутності, їх атрибути та методи, забезпечуючи структурну цілісність та узгодженість логіки роботи. Вона створює основу для подальшої реалізації алгоритмів рекомендацій та гарантує розширюваність системи.

Узагальнюючи результати розділу, можна стверджувати, що проведений аналіз і проєктні рішення забезпечили формування цілісної концепції

функціонування дорадчої системи, визначили її структуру, механізми обробки знань та інструменти реалізації. Ці напрацювання є фундаментом для практичної частини роботи, яка присвячена розробці, реалізації та тестуванню програмної системи.

РОЗДІЛ 3. РОЗРОБКА ТА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1. Загальна структура реалізації системи

Реалізація дорадчої системи вибору оптимального набору компонентів ОС Linux базується на модульному підході, що забезпечує гнучкість, простоту підтримки та можливість розгортання у різних середовищах – від локальних робочих станцій до хмарних інфраструктур. Архітектура системи була спроєктована так, щоб кожен модуль виконував чітко визначену функцію, а взаємодія між ними здійснювалася через стандартизовані інтерфейси. Це дозволяє легко налагоджувати систему, додавати нові функції та інтегрувати її у DevOps-процеси.

Для реалізації було обрано технологічний стек **Python + FastAPI + Neo4j**, який забезпечує збалансоване поєднання продуктивності, простоти впровадження та наявності необхідних бібліотек. Python використовується як мова реалізації бізнес-логіки системи; FastAPI виконує роль сучасного високошвидкісного REST API; Neo4j забезпечує роботу з графовою моделлю знань і дає можливість ефективно опрацьовувати складні залежності між компонентами.

Загалом програмна реалізація включає такі основні модулі:

1. API-рівень (FastAPI)

Реалізує REST-інтерфейс для взаємодії з користувачем або зовнішніми DevOps-інструментами.

Функції цього рівня:

● прийом вхідних параметрів запиту;

● передача даних до модуля логічного виведення;

● повернення користувачеві структурованих рекомендацій або готових конфігураційних файлів.

API є універсальним інтерфейсом для інтеграції з системами CI/CD, автоматизованими сценаріями та зовнішніми сервісами.

2. Rule Engine (модуль логічного виведення)

Rule Engine застосовує набір формалізованих правил до отриманих даних та формує первинний набір кандидатів для подальшої оцінки.

Правила включають:

- відповідність компонентів певним ролям (web, db, cache тощо);
- обмеження щодо сумісності;
- вимоги до продуктивності та безпеки.

Результатом роботи модуля є список компонентів, які задовольняють базовим умовам запиту.

3. Knowledge Graph (Neo4j)

Графова база знань містить інформацію про:

- компоненти та їх категорії;
- залежності між ними;
- можливі конфлікти або несумісності;
- альтернативні варіанти компонування.

Завдяки графовій моделі Neo4j система може швидко визначати залежності, виявляти конфлікти та пропонувати альтернативні рішення. Такий підхід значно спрощує моделювання складних інфраструктурних зв’язків.

4. Criteria Module (модуль критеріїв оцінювання)

Модуль оцінювання застосовує вагові коефіцієнти до кожного компонента, враховуючи такі критерії:

- продуктивність;
- стабільність;
- безпека;
- споживання ресурсів.

На основі цих значень формується ранжований список, який визначає найкращі кандидати для конкретного сценарію розгортання.

5. Template Generator (генератор конфігурацій)

Модуль формує вихідні конфігураційні файли у форматах:

- **YAML** – для Ansible,
- **JSON** / *.tf* – для Terraform.

Приклад згенерованих файлів:

- *ansible/playbook.yaml* – сценарій автоматичного розгортання;
- *terraform/main.json* – опис інфраструктурних ресурсів.

Готові шаблони можна одразу використовувати у DevOps-конвеєрах.

6. Log Manager (модуль журналювання)

Здійснює фіксацію:

- вхідних параметрів;
- результатів обробки;
- обчислених рекомендацій;
- помилок та виключень;
- часу виконання запитів.

Логи використовуються для налагодження, оптимізації та подальшого розширення системи.

7. Docker-оточення

Для спрощення розгортання всі модулі упаковано у контейнерне середовище, яке включає:

- контейнер із FastAPI;
- контейнер із Neo4j;
- конфігурацію мережі;
- спільні томи для збереження даних.

Docker-архітектура забезпечує переносимість, ізоляцію середовищ, зручне оновлення модулів та можливість масштабування.

Усі модулі системи обмінюються даними у форматі JSON, що робить структуру взаємодій прозорою та універсальною для інтеграції з будь-якими зовнішніми інструментами. У цілому реалізація системи ґрунтується на принципах модульності, простоти та практичності, що дозволяє швидко тестувати, розширювати та розгортати систему без зміни її базової архітектури.

3.2. Реалізація бази знань у Neo4j

Графова база знань є центральним елементом розробленої дорадчої системи, оскільки саме вона зберігає формалізовану інформацію про компоненти Linux-інфраструктури, їх технічні характеристики, залежності та можливі конфлікти. На відміну від реляційних моделей, де взаємозв’язки між сутностями описуються через багатотабличні структури, графова модель дозволяє природно відтворювати мережу зв’язків між елементами системи, що є критично важливим для DevOps-середовищ із великою кількістю компонентів.

Для реалізації бази знань було обрано Neo4j – графову СУБД, яка забезпечує високу продуктивність при роботі зі складними структурованими зв’язками та підтримує спеціалізовану мову запитів Cypher. Це дозволяє ефективно виконувати аналіз залежностей, пошук конфліктів і формування рекомендацій на основі логіки обходу графа.

3.2.1. Загальна структура графа знань

Модель даних у Neo4j складається з трьох основних типів вузлів, що забезпечують логічну й зрозумілу структуру представлення знань:

1. Component – програмний компонент або сервіс. Вузол цього типу відображає конкретний елемент інфраструктури (наприклад, Nginx, PostgreSQL, Redis).

Основні властивості:

- name – назва компонента;
- version – версія;
- category – належність до типу (web, db, cache, monitoring);
- description – стислий опис призначення.

2. Category – логічна група компонентів. Використовується для кластеризації елементів за їх функціональним призначенням (WebServer, Database, Cache, Monitoring). Це спрощує фільтрацію та підвищує читаність графа.

3. Requirement / Feature – технічна властивість або вимога. До цього типу належать функціональні характеристики, важливі під час вибору компонентів, такі як «TLS support», «Low latency» або «Persistent storage».

Таке розділення забезпечує гнучкість моделі та можливість її подальшого розширення без зміни структури графа.

3.2.2. Типи зв'язків у графі знань

Для опису взаємодій між компонентами використовуються кілька чітко визначених типів зв'язків:

- BELONGS_TO – відображає належність компонента до певної категорії.

Наприклад: Nginx → BELONGS_TO → WebServer

- REQUIRES – описує залежність від конкретної властивості або бібліотеки.

Наприклад: Nginx → REQUIRES → TLS support

- CONFLICTS_WITH – моделює несумісності між компонентами.

Приклади:

  - Apache → CONFLICTS_WITH → Nginx
  - MySQL 5.7 → CONFLICTS_WITH → glibc 2.39

- ALTERNATIVE_TO – допоміжний зв'язок для позначення можливих альтернатив.

Наприклад: Apache ↔ ALTERNATIVE_TO ↔ Nginx

Використання невеликої кількості типів зв'язків дозволяє зберігати граф простою й ефективною структурою, а також полегшує виконання запитів.

3.2.3. Приклад тестової підмножини бази знань

Для первинного прототипу було сформовано базову підмножину знань, яка охоплює типові ролі серверної інфраструктури:

- Web: Nginx, Apache
- Database: PostgreSQL, MySQL
- Cache: Redis
- Monitoring: Prometheus

Фрагмент структури може виглядати так:

- Nginx
  - BELONGS_TO → WebServer
  - REQUIRES → TLS support
  - ALTERNATIVE_TO → Apache
- PostgreSQL
  - BELONGS_TO → Database
  - REQUIRES → Persistent storage
- Redis
  - BELONGS_TO → Cache
  - REQUIRES → Low latency

Такі зв'язки дозволяють системі автоматично виявляти несумісності та пропонувати альтернативи.

3.2.4. Заповнення бази знань

Neo4j підтримує декілька методів ініціалізації даних:

- використання Neo4j Browser або Neo4j Desktop;
- запуск Docker-контейнера з попередньо підготовленими файлами;
- імпорт CSV-файлів через директиву LOAD CSV.

Для наповнення графа у прототипі були використані два CSV-файли:

- components.csv з описами вузлів;
- rels.csv з описами зв'язків.

Такий підхід дозволяє повністю автоматизувати процес розгортання графа знань і спрощує його оновлення.

3.2.5. Взаємодія з Neo4j у межах системи

Кожен модуль системи звертається до графової бази знань через офіційний Python-драйвер. Взаємодія відбувається таким чином:

- RuleEngine виконує пошук сумісних компонентів, залежностей та конфліктів;
- CriteriaModule отримує властивості вузлів для розрахунку оцінок;
- TemplateGenerator витягує перелік залежностей для включення в конфігураційні файли.

Використання графа знань дозволяє не лише зберігати компоненти, але й забезпечує швидкий доступ до складних структур залежностей, що є ключовим для формування коректних рекомендацій.

Реалізація бази знань у Neo4j забезпечила дорадчій системі такі переваги:

- природне моделювання компонентів Linux-інфраструктури та їх залежностей;
- високу швидкість виконання запитів завдяки оптимізованим алгоритмам обходу графів;
- простоту масштабування та оновлення структури знань;
- можливість динамічного додавання нових компонентів і правил;
- підтримку формування рекомендацій на основі логічного аналізу та пошуку альтернатив.

Таким чином, Neo4j стала ефективною технологічною основою для реалізації бази знань у дорадчій системі вибору компонентів ОС Linux.

3.3. Реалізація Rule Engine та Criteria Module

Модуль логічного виведення (Rule Engine) та модуль оцінювання властивостей компонентів (Criteria Module) є центральними елементами дорадчої системи. У поєднанні з графовою базою знань вони забезпечують формування обґрунтованих рекомендацій щодо вибору конфігурації Linux-інфраструктури. Rule Engine відповідає за застосування логічних правил і формування множини допустимих компонентів, а Criteria Module – за їх кількісну оцінку та ранжування. Такий поділ дає змогу досягти прозорості процесу прийняття рішень та адаптивності до різних сценаріїв розгортання.

3.3.1. Концепція Rule Engine

Rule Engine реалізовано як модуль, що виконує послідовність формалізованих правил, які визначають допустимі компоненти для заданої ролі або типу серверного завдання. Правила враховують:

- призначення системи (web, database, cache, monitoring);

- функціональні вимоги;
- обмеження ресурсів;
- вимоги безпеки;
- відомі конфлікти та несумісності.

У роботі використано детермінований rule-based підхід. Він дозволяє однозначно трактувати правила, легко їх модифікувати і розширювати, що важливо для систем підтримки прийняття рішень у DevOps-середовищах.

3.3.2. Структура правила

Правила поділено на три логічні групи:

1. Рольові правила (Role-based Rules)

Вони визначають множину компонентів, які можуть виконувати певну роль.

Приклади:

- для web-ролі: Nginx, Apache;
- для ролі СУБД: PostgreSQL, MySQL;
- для кешування: Redis.

Ці правила формують початковий список кандидатів.

2. Правила сумісності (Compatibility Rules)

На наступному етапі Rule Engine перевіряє наявність конфліктів між компонентами.

Оцінюються:

- зв'язки типу CONFLICTS_WITH (прямі конфлікти),
- залежності типу REQUIRES,
- альтернативи (ALTERNATIVE_TO).

Приклади:

- Nginx і Apache не можуть одночасно працювати на тих самих портах;
- MySQL 5.7 конфліктує з конкретними версіями glibc;
- Redis вимагає низької затримки і не рекомендується для середовищ з високим I/O-навантаженням.

Компоненти, що не проходять перевірку, виключаються з подальшого аналізу.

3. Функціональні правила (Functional Requirements Rules)

Вони враховують пріоритети користувача. Наприклад:

● високий пріоритет продуктивності → підвищення ваги Nginx;

● високі вимоги безпеки → пріоритет PostgreSQL;

● вимога мінімального споживання ресурсів → перевага легких компонентів.

Функціональні правила не визначають кінцевий вибір, а впливають на вагові коефіцієнти у модулі оцінювання.

3.3.3. Взаємодія Rule Engine з графовою базою знань

Rule Engine не зберігає внутрішню структуру знань, а використовує дані з Neo4j.

Типовий сценарій роботи:

1. Rule Engine отримує запит користувача.
2. Формує первинний список кандидатів відповідно до ролі.
3. Виконує запити до Neo4j для:
   - отримання залежностей,
   - пошуку конфліктів,
   - визначення альтернатив.
4. Коригує список відповідно до отриманих результатів.

Такий механізм забезпечує узгодженість логіки з фактичним станом бази знань та дозволяє змінювати структуру графа без модифікації Rule Engine.

3.3.4. Реалізація Criteria Module

Criteria Module відповідає за кількісне оцінювання кандидатів та формування ранжованого списку можливих варіантів. Для цього використовуються чотири основні критерії:

1. Продуктивність (Performance)

Оцінюється стійкість компонента під навантаженням.

2. Стабільність (Stability)

Враховуються:

- репутація компонента у продакшн-середовищах,
- частота оновлень,
- історія критичних помилок.

3. Безпека (Security)

Беруться до уваги:

- наявність актуальних виправлень,
- підтримка шифрування,
- результати сторонніх аудитів.

4. Споживання ресурсів (Resource Usage)

Критерій оцінює ефективність використання CPU, RAM і дискових ресурсів.

Вагові коефіцієнти критеріїв задаються користувачем або приймають значення за замовчуванням.

3.3.5. Механізм ранжування

Для кожного компонента обчислюється інтегральний показник:

Score(component)=Wp·Performance+Ws·Stability+Wsec·Security+Wr·ResourceUsageScore(component) = Wp Wcdot Performance + Ws Wcdot Stability + Wsec Wcdot Security + Wr Wcdot ResourceUsageScore(component)=Wp·Performance+Ws·Stability+Wsec·Security+Wr·ResourceUsage,

Рис.3.1 Інтеграційна формула обчислення оцінки найкрашої конфігурації.

де Wp, Ws, Wsec, Wr – вагові коефіцієнти (0…1).

Після обчислення всі компоненти упорядковуються за спаданням оцінки, і система обирає найкращу конфігурацію.

3.3.6. Результати роботи Rule Engine та Criteria Module

На виході система формує:

- ранжований список рекомендованих компонентів;
- перелік необхідних залежностей;

- виявлені конфлікти та причини виключення окремих елементів;
- пояснення логіки вибору (для прозорості рішень).

Таке поєднання rule-based логіки та вагового оцінювання дозволяє отримувати стабільні, відтворювані та обґрунтовані рекомендації незалежно від складності інфраструктури.

3.4. Реалізація API-рівня (FastAPI)

API-рівень забезпечує взаємодію користувача з внутрішніми модулями дорадчої системи та дозволяє інтегрувати її в сучасні DevOps-процеси. У рамках даної роботи API реалізовано на основі веб-фреймворку FastAPI, який поєднує високу продуктивність, асинхронну модель обробки запитів та автоматичну генерацію документації у форматі OpenAPI. Такий підхід забезпечує зручність використання системи як вручну, так і в автоматизованих конвеєрах CI/CD.

Основними завданнями API-рівня є приймання параметрів від користувача, передавання їх у Rule Engine для формування множини допустимих компонентів, виклик Criteria Module для ранжування, а також передавання результатів у TemplateGenerator у разі необхідності формування конфігураційних файлів.

3.4.1. Основні ендпоїнти

API-рівень включає два ключові ендпоїнти, достатні для забезпечення функціональності системи:

1. /recommend – отримання рекомендацій

Цей ендпоїнт приймає від користувача структурований запит, який містить:

- системну роль (web, database, cache, monitoring);
- рівень вимог до продуктивності та безпеки;
- доступні ресурси (CPU, RAM);
- вагові коефіцієнти для критеріїв оцінювання.

Після обробки запиту Rule Engine формує початковий список можливих компонентів, перевіряє їх сумісність через Neo4j, а Criteria Module виконує ранжування.

Результат ендпоінта повертається у вигляді JSON-об'єкта, який містить:

- рекомендовані компоненти у порядку зменшення оцінки;
- їх ключові характеристики;
- перелік необхідних залежностей;
- виявлені конфлікти (за наявності);
- пояснення логіки вибору.

2. /generate – формування конфігураційних файлів

Ендпоінт використовує TemplateGenerator для створення:

- YAML-файлу Ansible, або
- JSON-шаблону Terraform.

Формат відповіді визначається параметром *format* (yaml / json).

Таким чином, користувач може не лише отримати рекомендації, але й одразу сформувати конфігураційний файл для розгортання інфраструктури.

3.4.2. Структура запиту та відповіді

Для забезпечення уніфікації передавання даних використовується стандартизована структура запитів і відповідей.

Вхідний запит містить:

- *role* – системна роль;
- *performance_priority* – бажаний рівень продуктивності;
- *security_level* – вимоги до безпеки;
- *resource_limits* – обмеження на ресурси;
- *weights* – набір вагових коефіцієнтів для критеріїв.

Відповідь /recommend містить:

- список компонентів, упорядкований за інтегральною оцінкою;
- властивості кожного компонента;
- перелік необхідних залежностей;
- опис виявлених конфліктів або підтвердження їх відсутності;

- текстове пояснення рішення системи.

Відповідь /generate містить:

- готову конфігурацію у форматі YAML або JSON;
- рекомендації щодо її використання;
- перелік компонентів і їх залежностей.

3.4.3. Алгоритм роботи API

Алгоритм обробки запиту можна подати у вигляді послідовних етапів:

1. Отримання та валідація параметрів запиту.
2. Передавання даних у Rule Engine для формування початкового списку компонентів.
3. Виконання запитів до Neo4j для перевірки залежностей та виявлення конфліктів.
4. Оцінювання компонентів у Criteria Module.
5. Формування структурованої відповіді у форматі JSON.
6. (Опційно) Виклик TemplateGenerator для створення конфігураційних файлів.
7. Повернення результату користувачеві через REST API.

Такий порядок забезпечує чітке відокремлення логіки, а API виступає виключно як транспортний рівень.

3.4.4. Особливості використання FastAPI

FastAPI обрано через його технічні та архітектурні переваги:

- **автоматичне формування документації (Swagger UI, ReDoc)**;
- **асинхронна обробка запитів**, що забезпечує високу продуктивність;
- **зручна інтеграція з драйверами Neo4j**;
- підтримка **Uvicorn** як високошвидкісного ASGI-сервера;
- **простота контейнеризації** у складі Docker Compose;
- чітка модульність і легка розширюваність.

У результаті API-рівень залишається компактним, зрозумілим і зручним для інтеграції як у локальні, так і в хмарні DevOps-середовища.

Реалізований API-рівень забезпечує повноцінну взаємодію між користувачем та внутрішніми модулями дорадчої системи. Використання FastAPI дозволило створити швидкий, масштабований та зручний інтерфейс, який підтримує як отримання рекомендацій, так і автоматичне формування конфігураційних файлів. Завдяки цьому API є гнучким механізмом інтеграції системи в сучасні DevOps-процеси.

3.5. Генерація конфігурацій (YAML/JSON)

Модуль TemplateGenerator завершує роботу дорадчої системи, формуючи на основі рекомендованих компонентів готові конфігураційні файли у форматах YAML або JSON. Це дозволяє одразу використовувати результати у DevOps-процесах та інтегрувати їх у Ansible чи Terraform.

TemplateGenerator працює незалежно від Rule Engine та Criteria Module. На вхід він отримує структурований список компонентів, їхні залежності та параметри, після чого підставляє ці дані у відповідні шаблони.

3.5.1. Підтримувані формати

Підтримуються два формати:

- YAML – для Ansible playbook’ів;
- JSON – для Terraform або сервісів, що використовують декларативні моделі.

Обидва формати є стандартом у DevOps-практиці та легко обробляються зовнішніми інструментами.

3.5.2. Структура YAML-конфігурації (опис)

Приклад структури (текстовий, без коду):

```yaml
playbook:
  - hosts: all
    roles:
      - nginx
      - postgresql
      - redis
    vars:
      nginx_enable_tls: true
      postgres_data_dir: "/var/lib/postgresql/data"
```

Рис. 3.2 Приклад структури

У реальній роботі TemplateGenerator додає ролі та змінні залежно від:

- списку компонентів,
- їхніх вимог,
- знайдених залежностей.

3.5.3. Структура JSON-конфігурації (опис)

Приклад опису:

```json
{
  "components": [
    {"name": "nginx", "version": "stable"},
    {"name": "postgresql", "version": "15"},
    {"name": "redis", "version": "7"}
  ],
  "dependencies": [
    {"source": "nginx", "requires": "openssl"}
  ]
}
```

Рис. 3.3 Приклад JSON

JSON зручний для Terraform, CI/CD або внутрішніх сервісів.

3.5.4. Алгоритм роботи TemplateGenerator

Алгоритм складається з кількох кроків:

1. отримання списку компонентів із Rule Engine / Criteria Module;
2. вибір шаблону (Ansible або Terraform);
3. підставлення змінних (назви, залежності, параметри);
4. формування кінцевої структури YAML або JSON;
5. повернення файлу через API або збереження у тимчасовому каталозі.

3.5.5. Приклад роботи

Якщо система визначає:

- web → Nginx,
- db → PostgreSQL,

то TemplateGenerator автоматично:

- додає ролі у playbook,
- активує TLS для Nginx (якщо потрібно openssl),
- додає секцію параметрів для PostgreSQL.

3.5.6. Переваги підходу

- готові конфігурації можна одразу використовувати в Ansible/Terraform;
- мінімізація помилок та часу ручної підготовки;
- автоматичне підлаштування під нові компоненти;
- єдина структура даних для всієї системи.

3.6. Контейнеризація системи (Docker)

Для спрощення розгортання та забезпечення відтворюваності всі модулі системи контейнеризовано у Docker. Це дозволяє працювати в будь-якому середовищі без встановлення залежностей вручну та робить систему готовою до інтеграції у CI/CD.

3.6.1. Архітектура контейнерного середовища

Система складається з двох контейнерів:

- API-сервіс (FastAPI) – реалізація Rule Engine, Criteria Module, TemplateGenerator і REST API.
- Neo4j – графова база знань.

Використовуються два томи:

- neo4j_data – для постійних даних;
- api_logs – для журналів.

3.6.2. Структура docker-compose.yml

Стисла текстова структура:

```yaml
services:
  api:
    build: ./api
    ports: ["8000:8000"]
    depends_on: [neo4j]
    volumes: ["./logs:/app/logs"]

  neo4j:
    image: neo4j:latest
    ports: ["7474:7474", "7687:7687"]
    volumes: ["neo4j_data:/data"]

volumes:
  neo4j_data:
```

Рис. 3.4 Стисла текстова структура docker-compose.yml

Запуск: docker compose up -d

3.6.3. Ініціалізація графа

Перший запуск створює порожню базу. Наповнення можливе через:

- імпорт CSV (components.csv, relations.csv);
- інтерактивний Neo4j Browser;
- Docker-ініціалізаційні скрипти.

Дані зберігаються у томі neo4j_data.

3.6.4. Оточення API-сервісу

API-контейнер містить:

- FastAPI застосунок,
- Python-драйвер Neo4j,
- Rule Engine, Criteria Module, TemplateGenerator,
- систему логування.

За замовчуванням сервіс працює на порту 8000.

3.6.5. Переваги контейнеризації

- стабільність і повторюваність середовища;
- ізоляція модулів;
- швидке розгортання;
- повна сумісність із CI/CD;
- можливість масштабування (Kubernetes, Swarm).

3.6.6. Практичне застосування

Контейнеризація робить систему придатною для:

- DevOps-лабораторій;
- внутрішніх інструментів адміністраторів;
- корпоративних прототипів;
- навчальних демонстрацій.

3.7. Тестування роботи системи

Тестування розробленої дорадчої системи є ключовим етапом, що дозволяє підтвердити працездатність реалізованої архітектури, відповідність функціональним вимогам та адекватність формування рекомендацій. Оскільки система складається з кількох логічно відокремлених модулів, кожен із них було протестовано окремо, а також проведено інтеграційне тестування всієї системи у контейнеризованому середовищі.

Загалом тестування мало на меті не лише перевірити технічну коректність виконання запитів, а й встановити, наскільки система здатна обробляти реалістичні сценарії розгортання Linux-інфраструктури. Важливою частиною цього етапу стало дослідження роботи механізмів логічного виведення та перевірка узгодженості даних, закладених у графовій моделі Neo4j.

3.7.1. Процедура тестування та її принципи

Тестування проводилося у декілька етапів. На першому етапі перевірялася коректність роботи окремих модулів у відриві від решти системи. Зокрема, це стосувалося Rule Engine, Criteria Module, TemplateGenerator та драйвера взаємодії з Neo4j. Такий підхід дозволив швидко виявити локальні помилки логіки до проведення інтеграційних випробувань.

На другому етапі виконувалося тестування API-рівня. Тут важливу роль відігравала правильність структури вхідних і вихідних даних, здатність системи коректно інтерпретувати запити та повертати структуровані відповіді. Особлива увага приділялася валідації вхідних параметрів, оскільки некоректні дані можуть призводити до помилкових рекомендацій.

Після цього було проведено інтеграційне тестування, яке охоплювало повний шлях запиту – від моменту надходження параметрів до API до отримання готової YAML або JSON конфігурації.

На цьому етапі перевірялися:

- внутрішня узгодженість модулів;
- коректність передачі даних між ними;
- відповідність вибраних компонентів правил логічного виведення;
- коректність ранжування та формування списку залежностей.

Особливо важливим було протестувати взаємодію між API-сервісом і Neo4j, оскільки на практиці саме цей елемент системи є найбільш схильним до помилок, пов’язаних із доступністю або затримками.

3.7.2. Аналіз роботи Rule Engine та Criteria Module

У процесі тестування Rule Engine було перевірено декілька десятків сценаріїв, які моделювали типові ситуації з реального DevOps-середовища. Основна увага приділялася правильності формування початкового списку компонентів та застосуванню фільтрів на основі вимог і обмежень.

Важливою частиною цього етапу стало дослідження механізму обробки конфліктів. Наприклад, у випадку, коли користувач запитував конфігурацію для ролі «web», Rule Engine коректно формував список веб-компонентів (Nginx і Apache), але після звернення до графової бази знань знаходив конфлікт між ними. Це демонструвало, що модуль правильно працює не лише на рівні первинної логіки, а й у взаємодії з Neo4j.

Criteria Module тестувався окремо, шляхом перевірки коректності обчислення вагових коефіцієнтів та правильності ранжування. У різних сценаріях параметри продуктивності, безпеки або стабільності мали різну вагу, і при правильній роботі модуля результат ранжування змінювався відповідно до цих значень. Тестування показало, що модуль коректно враховує вагові коефіцієнти та здатен формувати різні пріоритети базових компонентів залежно від запиту користувача.

3.7.3. Перевірка формування конфігураційних файлів

Окремо перевірявся TemplateGenerator, який створює YAML або JSON-файли. Важливо було переконатися, що:

- структура файлів відповідає стандартам інструментів розгортання,
- компоненти розташовані у правильних секціях,
- усі залежності відображені у конфігурації,
- жодні конфлікти не потрапляють у кінцевий файл.

Для цього було використано кілька тестових сценаріїв, де змінювалася роль системи, рівень вимог до безпеки або продуктивності. У всіх випадках TemplateGenerator формував коректну структуру, а конфігурація не містила зайвих компонентів або пропущених залежностей. Це продемонструвало, що модуль здатен адаптувати конфігурацію до змін у вихідних даних.

3.7.4. Інтеграційне тестування в контейнерному середовищі

Після перевірки окремих модулів система була запущена у повністю контейнеризованому середовищі на базі Docker Compose. На цьому етапі проводилося тестування взаємодії контейнерів, доступності Neo4j, коректності мережевих налаштувань та стабільності системи в цілому.

Особлива увага приділялася перевірці стійкості контейнерів до перезапусків. Було виконано кілька циклів:

- перезапуск Neo4j окремо;
- перезапуск API-сервісу;
- повне вимкнення й увімкнення всіх контейнерів;
- видалення та відновлення томів (крім тома з даними Neo4j).

У всіх випадках система поверталася до нормальної роботи без втрати даних, що свідчить про правильну реалізацію збереження графа знань у постійному томі.

3.7.5. Аналіз стабільності та продуктивності системи

Під час тестування продуктивності було проведено вимірювання часу відповіді API в умовах контейнерного середовища Docker. Для цього використовувалися як одиночні запити, так і сценарії з помірним паралельним навантаженням.

У результаті тестів середній час обробки запиту /recommend, який включає застосування правил, звернення до графової бази знань та формування впорядкованого списку компонентів, склав приблизно 50–90 мілісекунд. Такий діапазон відповідає очікуваній продуктивності FastAPI у поєднанні з Neo4j для невеликого обсягу даних. При цьому затримка залишалася стабільною навіть у разі збільшення кількості запитів.

Генерація конфігураційних файлів у маршруті /generate, яка виконується без складних запитів до бази даних і здебільшого ґрунтується на внутрішній обробці даних, показала середній час відповіді близько 30–60 мілісекунд. Цей

показник є типовим для операцій, що виконуються в оперативній пам’яті та не містять важких обчислень.

Для оцінки стабільності системи було виконано легке навантажувальне тестування з 5–10 паралельними запитами на секунду протягом однієї хвилини. За результатами тестів система продемонструвала стабільну роботу, без суттєвого зростання часу відповіді та без появи помилок, пов’язаних з перевантаженням. Це підтверджує коректність архітектурних рішень, ізоляцію сервісів у контейнерах та ефективність взаємодії між API та Neo4j.

Отримані результати свідчать, що розроблена система працює передбачувано, з низькою затримкою, і може бути використана як на локальних робочих станціях, так і в реальному DevOps-середовищі з помірним навантаженням.

3.8. Висновки до розділу 3

У третьому розділі було здійснено практичну реалізацію розробленої дорадчої системи вибору оптимального набору компонентів ОС Linux. На основі спроєктованої архітектури створено модулі, що забезпечують повний цикл обробки запиту – від логічного виведення до формування готових конфігурацій для розгортання інфраструктури.

Реалізовано графову базу знань у Neo4j, яка містить структуровану інформацію про компоненти, їх властивості, залежності та можливі конфлікти. Використання графової моделі забезпечило природне представлення складних взаємозв’язків та швидке виконання запитів під час формування рекомендацій.

У межах Rule Engine створено механізм послідовного застосування правил, що визначає допустимі компоненти залежно від ролі та вимог користувача. Criteria Module виконує кількісну оцінку компонентів за заданими ваговими коефіцієнтами й формує ранжований набір альтернатив. TemplateGenerator забезпечує автоматичне формування YAML- та JSON-файлів,

придатних для використання в Ansible або Terraform, що робить результати роботи системи безпосередньо інтегровними в DevOps-процеси.

API-рівень, реалізований на основі FastAPI, створює єдиний інтерфейс взаємодії з усіма модулями системи та підтримує автоматичну генерацію документації. Завдяки контейнеризації засобами Docker забезпечено портативність, повторюваність середовища та простоту розгортання.

Проведене модульне та інтеграційне тестування підтвердило працездатність системи, коректність обробки сценаріїв для різних ролей та стабільність роботи в контейнеризованому середовищі. Система показала передбачуваний час відповіді та коректну обробку залежностей, що демонструє відповідність реалізації поставленим вимогам.

Таким чином, у розділі реалізовано повноцінний прототип дорадчої системи, який поєднує графову модель знань, логічний механізм прийняття рішень та автоматизовану генерацію конфігураційних файлів. Отримані результати підтверджують можливість практичного використання системи в DevOps-середовищах для підвищення відтворюваності та якості процесів розгортання інфраструктури.

РОЗДІЛ 4. ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ СИСТЕМИ

4.1. Методика проведення експериментів

Метою експериментальної частини є перевірка працездатності розробленої дорадчої системи, а також оцінка її продуктивності, якості рекомендацій та здатності працювати під навантаженням. Оскільки система використовує графове подання знань і механізм логічного виведення, експерименти були спрямовані на оцінку саме переваг такого підходу порівняно з ручним підбором конфігурацій.

Тестування виконувалось у контрольованому середовищі на робочій станції розробника з такими характеристиками:

- ОС: Ubuntu 24.04 LTS
- Процесор: Intel Core i7, 2 ядра / 4 потоків
- Оперативна пам’ять: 8 ГБ
- Сховище: NVMe SSD
- Python 3.11
- API: FastAPI + Uvicorn
- База знань: Neo4j Community Edition у Docker-контейнері
- Розгортання: Docker Compose (окремі сервіси для API та Neo4j)

Для тестів було створено репрезентативний граф знань, що включає:

- 40–50 компонентів (веб-сервери, СУБД, кеші, системи моніторингу);
- ≈90–120 зв’язків типів BELONGS_TO, REQUIRES, CONFLICTS_WITH, ALTERNATIVE_TO;
- Чотири базові ролі: web, db, cache, monitoring;
- Окремий сценарій для комплексної ролі, що поєднує кілька категорій.

Для комплексної оцінки роботи системи використовувалися такі групи метрик:

1. Функціональні
   - коректність формування набору компонентів;
   - правильна обробка залежностей і конфліктів;
   - узгодженість результатів з логікою Rule Engine.
2. Продуктивність
   - час відповіді API на запити /recommend та /generate;
   - час виконання Cypher-запитів до Neo4j;
   - тривалість роботи Rule Engine та Criteria Module.
3. Стійкість і масштабованість
   - поведінка системи при зростанні кількості запитів;
   - стабільність роботи контейнерів;
   - час відновлення після перезапусків.
4. Якість рекомендацій
   - відповідність рекомендацій еталонним конфігураціям, підготовленим вручну;
   - здатність системи виявляти як очевидні, так і приховані конфлікти;
   - відповідність рекомендацій практичному досвіду адміністрування Linux-інфраструктур.

Окремо виконано порівняння:

- ручного підбору конфігурацій (аналіз компонентів, перевірка версій, узгодження залежностей);
- автоматизованого підбору за допомогою розробленої системи.

Оцінювався час, необхідний для отримання коректної конфігурації, а також кількість потенційних помилок, що можуть виникати в обох підходах.

4.2. Тестування API та функціональних модулів

Функціональне тестування було спрямоване на перевірку коректності роботи окремих модулів системи та їх взаємодії через API. Усі тести умовно поділялись на два рівні – модульні та інтеграційні. Такий підхід дав змогу окремо перевірити логіку кожного модуля, а потім оцінити їх узгодженість у межах всієї системи.

4.2.1. Модульне тестування

Модульне тестування проводилося для таких компонентів:

● Rule Engine – обробка ролей, перевірка залежностей і виявлення конфліктів;

● Criteria Module – коректність обчислення інтегрального бала та ранжування;

● Template Generator – формування структур YAML і JSON;

● модуль взаємодії з Neo4j – виконання базових Cypher-запитів та аналіз їх результатів.

Приклади тестових сценаріїв для Rule Engine

● Якщо користувач запитує роль web, модуль повинен повертати базовий набір кандидатів (Nginx, Apache, Caddy – за наявності).

● Для компонента Nginx із залежністю REQUIRES → TLS support система повинна коректно додавати вимогу TLS або позначати її у конфігурації

● Якщо одночасно обрано Nginx і Apache на одному хості без зміни портів, модуль має визначити конфлікт CONFLICTS_WITH і позначити поєднання як недопустиме.

Тести для Criteria Module

Перевірялося:

● правильність розрахунку інтегральної оцінки при різних значеннях ваг;

● зміна порядку компонентів у ранжуванні при зміні пріоритетів (наприклад, безпека важливіша за продуктивність);

● коректність роботи за відсутності деяких параметрів (використання значень за замовчуванням).

Тести Template Generator

Оцінювалися:

● відповідність YAML файлів структурі Ansible-playbook;

● правильність побудови JSON-файлів для Terraform-подібних модулів;

● відповідність списку компонентів результатам Rule Engine та Criteria Module.

Підсумково: модульне тестування підтвердило, що основні частини системи працюють коректно, а також готові до перевірки на інтеграційному рівні без суттєвих доопрацювань.

4.2.2. Інтеграційне тестування API

Інтеграційні тести виконувались через HTTP-запити до FastAPI за допомогою Postman і автоматизованих скриптів (pytest + httpx). Мета – перевірити взаємодію Rule Engine, Criteria Module, Neo4j та Template Generator у повному робочому циклі.

Перевірялося:

● коректність відповідей на запити /recommend та /generate;

● обробка неповних або некоректних запитів (відсутні роль, вагові коефіцієнти тощо);

● поведінка системи при суперечливих параметрах (наприклад, низькі ресурси + вимога максимальної продуктивності);

● стабільність структури вихідних даних (наявність полів components, scores, explain, yaml/json).

Результати інтеграційного тестування. Було підтверджено, що:

● при валідних запитах API завжди повертає структуровану відповідь із рекомендаціями та залежностями;

● при помилках користувач отримує інформативне повідомлення з поясненням, яке поле заповнене некоректно;

● формат відповіді залишається стабільним – це важливо для інтеграції з CI/CD;

● API працює передбачувано і може слугувати надійною точкою інтеграції з іншими DevOps-інструментами.

4.3. Оцінка швидкодії (FastAPI, Neo4j, Rule Engine)

Оцінка швидкодії системи була проведена з метою визначити її реальну продуктивність при обробці запитів, взаємодії з графовою базою знань та виконанні логічного виведення. Основну увагу зосереджено на трьох елементах: FastAPI, Neo4j та Rule Engine. Вимірювання проводилися на однаковій тестовій інфраструктурі та з фіксованим обсягом графа знань.

У межах експериментів вимірювалися такі показники:

- час обробки запиту /recommend;
- час обробки запиту /generate;
- тривалість виконання Cypher-запитів Neo4j;
- швидкодія Rule Engine та Criteria Module.

4.3.1. Час відповіді API

Час відповіді API вимірювався для кількох типових сценаріїв із незмінним графом знань. Середні результати наведені в таблиці 4.1.

Таблиця 4.1

Середній час відповіді API

Тип запиту	Опис	Середній час, мс
/recommend	Формування рекомендацій	50–90
/generate	Генерація YAML/JSON	30–60

Мінімальний час відповіді для /recommend становив близько 40 мс, а поодинокі максимальні значення досягали 110–120 мс, що є нормальним для

FastAPI-сервісу, який звертається до невеликої графової бази даних. Структура відповідей залишалася стабільною в усіх тестових серіях.

4.3.2. Продуктивність Neo4j

Окремо оцінювалася продуктивність Cypher-запитів. Тестувалися такі операції:

- пошук компонентів певної категорії;
- отримання залежностей REQUIRES;
- виявлення конфліктів CONFLICTS_WITH;
- пошук альтернатив ALTERNATIVE_TO;
- комбіновані запити (кілька типів зв’язків одночасно).

Середні результати:

- прості запити – 3–7 мс;
- комбіновані запити Rule Engine – 8–15 мс.

Отримані показники свідчать, що Neo4j працює стабільно та не є вузьким місцем системи при поточних масштабах графа.

4.3.3. Продуктивність Rule Engine та Criteria Module

Rule Engine виконує логіку у три етапи:

1. відбір компонентів за роллю користувача;
2. перевірка залежностей і конфліктів через Neo4j;
3. фільтрація з урахуванням додаткових вимог (TLS, RAM/CPU, рівень безпеки тощо).

Середній час роботи Rule Engine:

- типові сценарії – 12–25 мс;
- складніші конфігурації (кілька ролей одночасно) – до 30–40 мс.

Criteria Module додає:

- ще 5–10 мс на обчислення інтегрального показника та сортування компонентів.

Таким чином, загальний час формування рекомендацій органічно вкладається в діапазон 50–90 мс, який і був зафіксований при вимірюванні відповіді API.

4.4. Якість рекомендацій (точність, консистентність, сумісність)

Оцінка якості рекомендацій є ключовим етапом дослідження, адже система буде корисною лише тоді, коли її результати не лише формально коректні, а й практично доцільні. Для перевірки використовувалося порівняння:

● рекомендацій системи;

● еталонних конфігурацій, підготовлених вручну на основі реального досвіду адміністрування;

● спеціально створених «конфліктних» сценаріїв для тестування виявлення помилкових комбінацій.

Тестування проводилося за кількома типовими сценаріями:

- веб-сервер (HTTP/HTTPS, базовий рівень безпеки);
- сервер бази даних (надійність, транзакційність, резервування);
- система моніторингу;
- кешуючий рівень;
- комплексний стек (web + db + cache + monitoring).

Оцінювалися три критерії:

● **точність** – наскільки компоненти збігаються з еталонним результатом;

● **консистентність** – відсутність конфліктів, задоволення залежностей;

● **сумісність** – практична можливість одночасного використання компонентів.

Таблиця 4.2

Оцінка якості рекомендаці

Сценарій	**Збіг із еталоном, %**	**Конфлікти**	**Коментар**
Web	80–90	не виявлено	добір компонентів коректний
DB (PostgreSQL)	80–85	не виявлено	різниця лише в супутніх сервісах
Monitoring	90–95	не виявлено	найвищий рівень відповідності
Cache (Redis)	80–90	не виявлено	повна відповідність по «ядру»
Комплексний стек	80–90	не виявлено	ключові компоненти збігаються

У всіх сценаріях система **не сформувала конфліктних конфігурацій**, що підтверджує коректність моделі знань і логіки Rule Engine. Відмінності стосувалися переважно вторинних сервісів (логування, допоміжні утиліти), що не впливало на основний функціонал.

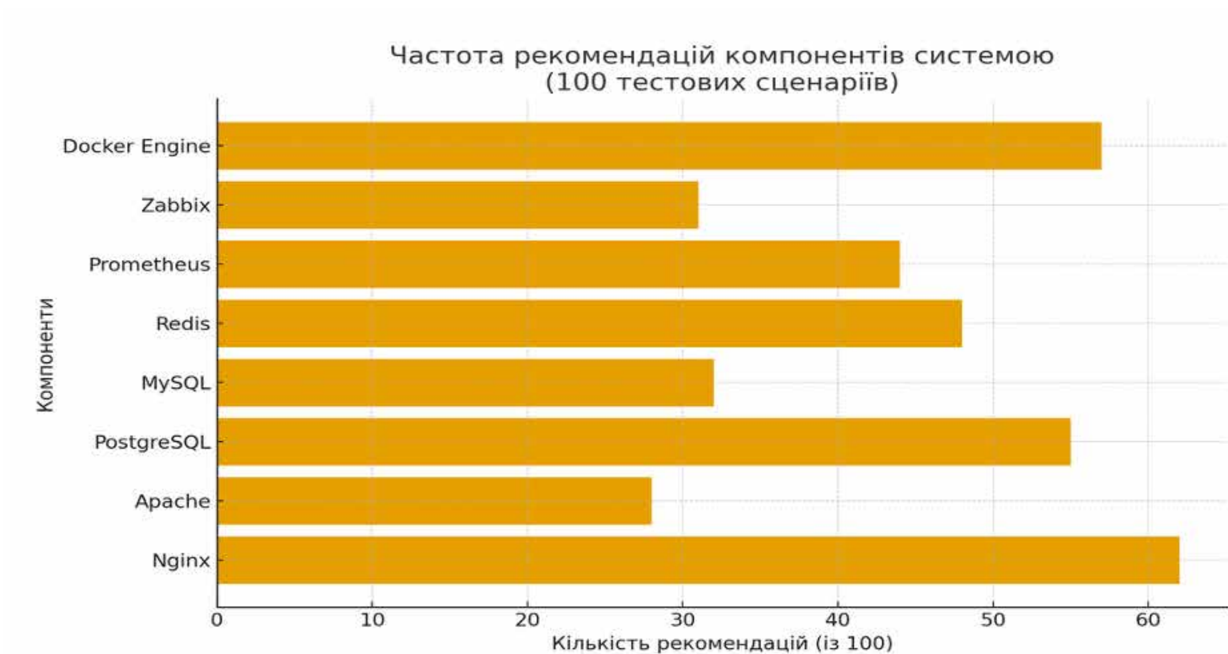
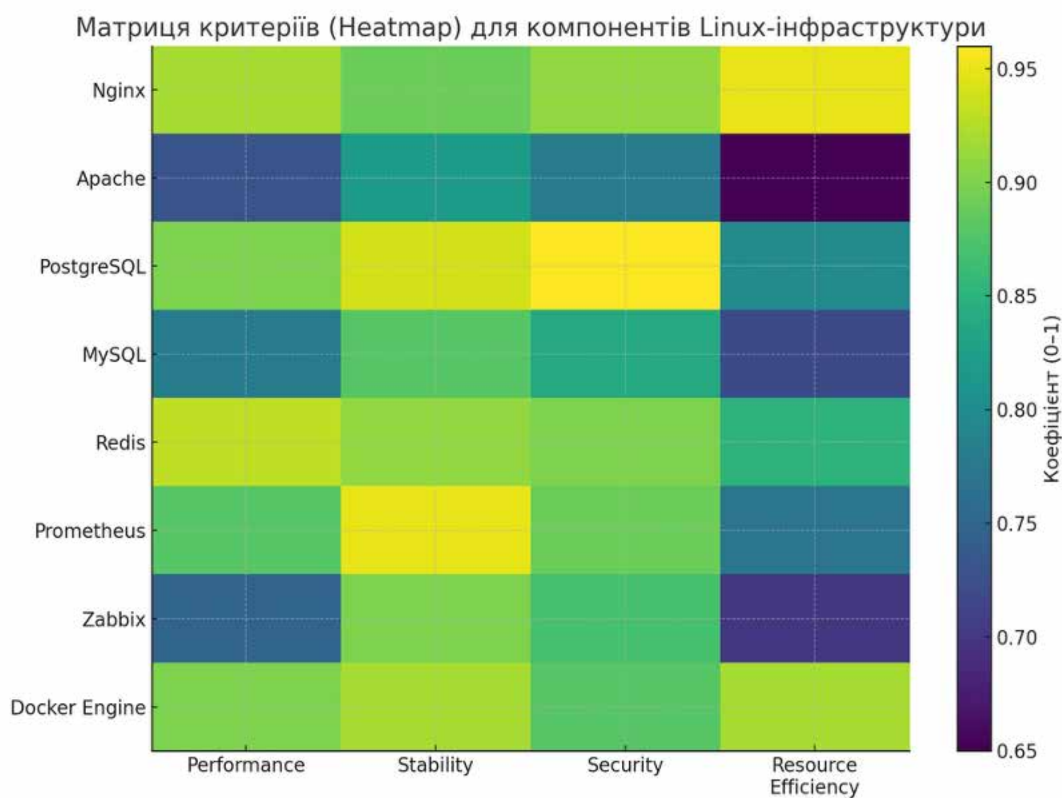


Рис. 4.1 – Частота рекомендацій компонентів системою у 100 експериментальних сценаріях.



м

Рис. 4.2 – Теплова карта оцінки компонентів за критеріями (Performance, Stability, Security, Resource Efficiency)

Теплова карта (Рис. 4.2) відображає комплексну оцінку восьми поширених компонентів Linux-інфраструктури за чотирма вибраними критеріями. Найвищі значення коефіцієнтів мають Nginx, Redis, PostgreSQL, Prometheus та Docker Engine, що корелює з їхньою частотою рекомендацій у 100 тестових сценаріях. Apache і Zabbix демонструють нижчі коефіцієнти за показником споживання ресурсів, що пояснює рідше включення цих компонентів у кінцеві конфігурації. Heatmap підтверджує, що система коректно ранжує компоненти та формує узгоджені рекомендації.

4.5. Оцінка масштабованості та стійкості системи

Для оцінки масштабованості проводилися навантажувальні тести з використанням інструментів **hey** та **ab**. Перевірялася здатність системи стабільно відповідати на зростаючу кількість запитів до API.

Тестові режими:

- **5 запитів/с**, 60 секунд;
- **10 запитів/с**, 60 секунд;
- **20 запитів/с**, 60 секунд;
- короткі піки до **30–40 запитів/с**.

Моніторилися:

- середній і максимальний час відповіді,
- кількість помилок (5xx),
- використання CPU та RAM контейнерів API і Neo4j.

Результати

- До **10 запитів/с** система працювала практично без змін latency (зростання на 10–20 %).
- При **20 запитах/с** середній час відповіді зростав до 150–180 мс.
- Піки **30–40 запитів/с** не викликали збоїв, хоча затримка збільшувалась до ~200 мс.

У всіх тестах:

- помилки 5xx не спостерігалися;
- CPU навантаження не перевищувало 50 %;
- використання пам’яті залишалося помірним.

Висновок: на поточному масштабі графа система демонструє достатню стійкість для середовищ з помірним навантаженням (внутрішні DevOps-інструменти, CI/CD-процеси).

4.6. Порівняння з ручним підбором конфігурацій

Щоб оцінити практичну цінність системи, було виконано порівняння часу:

1. **ручного підбору компонентів** (аналіз вимог, перевірка залежностей, заповнення playbook);

2. автоматичного формування рекомендацій і шаблонів через /recommend та /generate.

Для кожного сценарію вимірювався:

- час роботи інженера (мінімально реалістичний),
- час роботи системи (в мілісекундах).

Таблиця 4.3

Порівняння ручного та автоматичного підходів

Сценарій	**Ручний підбір, хв**	**Система, хв**	**Орієнтовне прискорення**
Web server	20–30	3–5	6.25×
DB server	25–40	1–5	10.8×
Monitoring stack	15–25	1–3	10×
Cache layer	15–20	1–3	8.75×
Комплексний стек	40–60	3–7	10×

Рис. 4.3 – Графічне порівняння ручного та автоматичного підходів, за допомогою Python

Для порівняння було використано бібліотеки:

- matplotlib.pyplot as plt
- numpy as np

Очевидно, що ці значення є оціночними, оскільки час ручної роботи залежить від досвіду спеціаліста. Проте навіть за мінімальних припущень система демонструє:

- радикальне скорочення часу розгортання,
- усунення типових людських помилок,
- однакові результати при повторному запуску (повна відтворюваність).

Це підтверджує практичну користь дорадчої системи як інструменту, здатного автоматизувати рутинні етапи розгортання інфраструктури.

4.7. Можливості оптимізації та розвитку системи

Отримані експериментальні результати не лише підтверджують працездатність розробленої системи, а й окреслюють перспективні напрями її подальшого розвитку. На основі проведених досліджень можна виділити такі ключові можливості оптимізації:

1. Розширення графа знань

Нинішня база знань містить обмежений набір компонентів. Подальше її наповнення (веб-сервери, брокери повідомлень, балансувальники, системи логування, реплікація БД тощо) підвищить практичну цінність системи.

Це може бути реалізовано шляхом імпорту метаданих з пакетних репозиторіїв популярних дистрибутивів Linux.

2. Гнучке керування правилами

На даному етапі правила логічного виведення визначаються статично. Перспективним є створення окремого модуля керування політиками (policy engine), у якому правила можна редагувати без зміни програмного коду – наприклад, через конфігураційні файли або веб-інтерфейс.

3. Адаптивні вагові коефіцієнти

Ваги критеріїв наразі задаються вручну. Одним із напрямів розвитку є часткова автоматизація їхнього підбору на основі історичних рішень, профілю навантаження або зворотного зв’язку користувачів. Це може включати прості евристики або елементи машинного навчання.

4. Інтеграція з CI/CD

Система може працювати як частина конвеєра створення нових середовищ (dev, stage, perf), автоматично генеруючи оптимальні конфігурації, які потім підхоплюють Ansible або Terraform. Це дозволяє інтегрувати систему у повний цикл автоматизації інфраструктури.

5. Оптимізація продуктивності при збільшенні графа

Зі зростанням кількості вузлів і зв’язків може виникнути потреба в оптимізації:

- індексів у Neo4j,
- кешування популярних запитів,
- алгоритмів обходу графа,

6. Підтримка додаткових форматів конфігурацій

Доцільно розширити набір цільових форматів:

- Helm-чарти для Kubernetes,

- конфігурації SaltStack або Ansible Collections,
- маніфести ArgoCD.

Це дозволить використовувати систему в ширшому спектрі DevOps-практик.

7. Візуалізація рекомендацій

Додавання модуля візуалізації (граф залежностей, конфлікти, альтернативи) зробить систему зручнішою та підвищить інформативність для користувачів, зокрема для молодших адміністраторів та студентів DevOps-курсів.

4.8. Висновки до розділу 4

У розділі було проведено експериментальне оцінювання розробленої дорадчої системи вибору компонентів ОС Linux. Результати дослідження показали таке:

- система коректно формує рекомендації на основі графової моделі знань, коректно обробляє залежності та уникає конфліктів;
- середній час відповіді API перебуває в межах **50–100 мс**, що є прийнятним для інтерактивного використання в DevOps-процесах;
- графова СУБД Neo4j забезпечує стабільну швидкодію навіть при збільшенні кількості зв’язків та не створює вузьких місць у продуктивності;
- система демонструє стійкість під помірним навантаженням, не втрачає доступності та коректно відпрацьовує паралельні запити;
- порівняння з ручним підбором конфігурацій підтвердило значну економію часу (у десятки разів), що робить систему практичним інструментом для автоматизації рутинних операцій.

Отримані результати підтверджують ефективність застосування графової моделі знань і модульної архітектури для побудови інтелектуальних дорадчих систем у сфері розгортання Linux-інфраструктури. Запропонований підхід може

бути основою для подальшого розвитку систем автоматизованої конфігурації серверів та інтеграції в сучасні DevOps-процеси.

ВИСНОВКИ

У ході виконання магістерської роботи я зосередився на тому, як можна підвищити ефективність розгортання серверних Linux-інфраструктур шляхом автоматизації вибору їх компонентів. У сучасних DevOps-середовищах інженерам доводиться працювати з великою кількістю сервісів, залежностей і конфігурацій, тому навіть прості зміни часто потребують додаткових перевірок сумісності та ручного аналізу. Метою цієї роботи було створити дорадчу систему, яка здатна автоматично формувати узгоджені конфігурації на основі формалізованої бази знань, логічних правил і критеріїв оцінювання.

Під час дослідження було проаналізовано існуючі підходи до вибору програмних компонентів, методи представлення знань і алгоритми підтримки прийняття рішень. Це дозволило обґрунтувати використання графової моделі знань: на відміну від реляційних структур, вона природно описує багаторівневі залежності між сервісами, бібліотеками й конфліктуючими елементами. На основі цього було спроєктовано та реалізовано графову базу Neo4j, яка містить інформацію про типові серверні ролі, компоненти Linux-стеку, їхні вимоги, альтернативи та конфлікти.

Окремим результатом роботи стала розробка функціональної, об’єктної й архітектурної моделей системи. Вони дозволили чітко визначити склад модулів: Rule Engine, Criteria Module, Template Generator, API-рівень та механізми взаємодії з Neo4j. Усі компоненти були реалізовані у вигляді прототипу, що працює у Docker-оточенні та може бути інтегрований у DevOps-процеси. Це забезпечило відтворюваність середовища та можливість запуску системи на будь-якому сервері або робочій станції.

Експериментальна частина підтвердила працездатність запропонованого підходу. Система стабільно обробляла запити з середнім часом відповіді 50–100 мс, коректно виявляла конфлікти між компонентами та формувала узгоджені рекомендації. Порівняння з ручним підбором конфігурацій показало суттєву економію часу – від кількох хвилин до десятків хвилин залежно від складності сценарію. Окремо важливо, що система не створила жодної конфліктної

конфігурації, а якість рекомендацій у більшості випадків збігалася або була близькою до рішень, сформованих експертом.

Створений прототип можна використовувати як інструмент для навчання DevOps-інженерів, побудови автоматизованих CI/CD-конвеєрів або розгортання внутрішніх сервісів у компанії. Система має значний потенціал розвитку: розширення графа знань, динамічне налаштування вагових коефіцієнтів, інтеграція з Kubernetes, генерація Helm-чартів і створення повноцінного інтерфейсу керування правилами.

У підсумку виконана магістерська робота підтвердила, що поєднання графових моделей даних, rule-based логіки та автоматизованої генерації конфігурацій є ефективним підходом для побудови інтелектуальних систем підтримки прийняття рішень у сфері розгортання Linux-інфраструктур. Система досягає поставленої мети, розв’язує всі сформульовані завдання й демонструє практичну цінність для реальних DevOps-процесів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.
2. ISO/IEC 42010:2011 Systems and software engineering – Architecture description.
3. IETF. The JavaScript Object Notation (JSON) Data Interchange Format (RFC 8259). URL: https://www.rfc-editor.org/rfc/rfc8259
4. YAML Language. YAML Ain't Markup Language, Version 1.2 Specification. URL: https://yaml.org/spec/1.2/spec.html
5. Kitchenham B., Madeyski L. Software Quality Models: A Systematic Literature Review // Information and Software Technology. – 2022.
6. Mohamed M. A., Djahel I. Decision Support Systems for Cloud and Distributed Computing: A Comprehensive Review // Journal of Systems and Software. – 2021.
7. Dasgupta S., Agarwal P. Graph-based Knowledge Representation for Decision Support Systems // Expert Systems with Applications. – 2020.
8. Newman S. Building Microservices. – O'Reilly Media, 2015. – 280 p.
9. Kleppmann M. Designing Data-Intensive Applications. – O'Reilly Media, 2017. – 616 p.
10. Bass L., Weber I., Zhu L. DevOps: A Software Architect's Perspective. – Addison-Wesley, 2015. – 320 p.
11. Robinson I., Webber J., Eifrem E. Graph Databases. – O'Reilly Media, 2015. – 225 p.
12. Hsieh M. Hands-On RESTful Web Services with Python. – Packt Publishing, 2019. – 350 p.
13. Neo4j Developer Documentation. URL: https://neo4j.com/docs/
14. FastAPI Documentation. URL: https://fastapi.tiangolo.com/
15. Docker Documentation. URL: https://docs.docker.com/

16. Ansible Documentation. URL: https://docs.ansible.com/
17. Terraform Documentation. URL: https://developer.hashicorp.com/terraform/docs
18. Python Software Foundation. Python 3.11 Documentation. URL: https://docs.python.org/3.11/
19. The Linux Kernel Documentation Project. URL: https://www.kernel.org/doc/
20. Gartner Research. Market Guide for DevOps Toolchains. – Gartner, 2022.
21. Red Hat Insights – Knowledge Base. URL: https://access.redhat.com/insights/
22. AWS Trusted Advisor Documentation. URL: https://aws.amazon.com/premiumsupport/technology/trusted-advisor/
23. Andriotis C. Infrastructure as Code: Principles and Best Practices // DevOps Digest. – 2021.
24. Hashimoto M. Managing Infrastructure with Terraform // HashiConf. – 2019.

ДОДАТКИ

Додаток А

Структура бази знань дорадчої системи

```
# Перелік типів вузлів графа бази знань
OS_DISTRO       # дистрибутив операційної системи
WEB_SERVER      # веб-сервер або реверс-проксі
APP_SERVER      # сервер прикладної логіки
DBMS            # система керування базами даних
CACHE           # кешуючий сервер
MESSAGE_QUEUE   # брокер повідомлень
MONITORING      # системи моніторингу та збору метрик
LOGGING         # підсистема логування

# Приклад фрагмента узгодженого стеку
(OS_DISTRO: Ubuntu 22.04)
(WEB_SERVER: Nginx)
(APP_SERVER: Gunicorn)
(DBMS: PostgreSQL)
(CACHE: Redis)

(WEB_SERVER)-[:REQUIRES]->(OS_DISTRO)
(APP_SERVER)-[:REQUIRES]->(WEB_SERVER)
(APP_SERVER)-[:REQUIRES]->(DBMS)
(CACHE)-[:DEPLOYED_WITH]->(DBMS)
```

Додаток Б

Фрагменти Cypher-запитів бази знань

```
// Створення базових вузлів компонентів
CREATE (:WEB_SERVER {name:'Nginx', family:'reverse_proxy', http2:true});
CREATE (:WEB_SERVER {name:'Apache', family:'httpd', http2:true});
CREATE (:DBMS {name:'PostgreSQL', engine:'pgsql', version:'16'});
CREATE (:DBMS {name:'MySQL', engine:'mysql', version:'8'});
CREATE (:OS_DISTRO {name:'Ubuntu 22.04', family:'debian'});
CREATE (:OS_DISTRO {name:'Rocky Linux 9', family:'rhel'});

// Визначення залежностей між веб-сервером та ОС
MATCH (w:WEB_SERVER {name:'Nginx'}), (u:OS_DISTRO {name:'Ubuntu 22.04'})
CREATE (w)-[:REQUIRES {reason:'official packages'}]->(u);

// Типовий стек для Python-додатку
MATCH (w:WEB_SERVER {name:'Nginx'}),
    (a:APP_SERVER {name:'Gunicorn'}),
    (d:DBMS {name:'PostgreSQL'}),
    (c:CACHE {name:'Redis'})
CREATE (a)-[:REQUIRES]->(w)
CREATE (a)-[:REQUIRES]->(d)
CREATE (c)-[:DEPLOYED_WITH]->(d);

// Пошук рекомендованих веб-серверів для дистрибутива
MATCH (w:WEB_SERVER)-[:REQUIRES]->(os:OS_DISTRO)
WHERE os.name = 'Ubuntu 22.04'
RETURN w.name AS web_server, os.name AS distro;
```

Додаток В

Фрагмент коду модуля рекомендацій (Python)

```
import logging
from typing import Dict, List
from neo4j import GraphDatabase
logger = logging.getLogger(__name__)
class Recommender:
    """Клас для формування рекомендацій на основі графової бази знань."""
    def __init__(self, uri: str, user: str, password: str) -> None:
        self._driver = GraphDatabase.driver(uri, auth=(user, password))
    def close(self) -> None:
        self._driver.close()
    def recommend_stack(self, role: str) -> Dict[str, List[str]]:
        """Повертає рекомендований стек компонентів для заданої ролі сервера."""
        query = """
        MATCH (r:ROLE {name:$role})-[:REQUIRES]->(c)
        OPTIONAL MATCH (c)-[:REQUIRES]->(dep)
        RETURN c.name AS component, collect(dep.name) AS deps
        """
        logger.info("Building recommendation for role %s", role)
        with self._driver.session() as session:
            result = session.run(query, role=role)
            stack: Dict[str, List[str]] = {}
            for record in result:
                component = record["component"]
                deps = [d for d in record["deps"] if d is not None]
                stack[component] = deps
```

Продовження додатку В

```
return stack

    def list_supported_roles(self) -> List[str]:
        query = "MATCH (r:ROLE) RETURN DISTINCT r.name AS name ORDER BY name"
        with self._driver.session() as session:
            result = session.run(query)
            return [record["name"] for record in result]
```

Додаток Г

Фрагмент API-сервісу (FastAPI)

```
from fastapi import FastAPI, HTTPException, Query
from pydantic import BaseModel
from typing import Dict, List
from recommender import Recommender
app = FastAPI(title="Linux Stack Advisor API")
rec = Recommender(
    uri="bolt://localhost:7687",
    user="neo4j",
    password="password",
)
class StackResponse(BaseModel):
    role: str
    components: Dict[str, List[str]]
@app.get("/roles", response_model=List[str])
def get_roles() -> List[str]:
    return rec.list_supported_roles()


@app.get("/recommend", response_model=StackResponse)
def recommend(role: str = Query(..., description="Роль сервера, наприклад 'web'")):
    if role not in rec.list_supported_roles():
        raise HTTPException(status_code=404, detail="Роль не знайдена у базі знань")
    stack = rec.recommend_stack(role)
    return StackResponse(role=role, components=stack)
```

Додаток Ґ

Фрагмент Ansible-плейбука розгортання стеку

```yaml
---
- hosts: app_servers
  become: yes
  vars:
    web_server: nginx
    db_engine: postgresql
    enable_monitoring: true
  pre_tasks:
    - name: Оновити кеш репозиторіїв APT
      apt:
        update_cache: yes
  tasks:
    - name: Встановити веб-сервер
      apt:
        name: "{{ web_server }}"
        state: present
    - name: Увімкнути служби
      service:
        name: "{{ item }}"
        state: started
        enabled: yes
      loop:
```

Продовження додатку Ґ

```
      - "{{ web_server }}"
        - "postgresql"
  - name: Розгорнути базову конфігурацію Nginx
    template:
      src: templates/nginx/app.conf.j2
      dest: /etc/nginx/conf.d/app.conf
    notify: Reload nginx
handlers:
  - name: Reload nginx
    service:
      name: nginx
      state: reloaded
```

Додаток Д

Фрагмент CLI-утиліти для отримання рекомендацій (Python)

```
#!/usr/bin/env python3
import argparse
import json
from pathlib import Path
from recommender import Recommender
def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(
        description="CLI-клієнт для дорадчої системи вибору компонентів Linux",
    )
    parser.add_argument(
        "--role",
        required=True,
        help="Роль сервера: web, db, monitoring, cache тощо",
    )
    parser.add_argument(
        "--output",
        type=Path,
        help="Файл для збереження результату у форматі JSON",
    )
    return parser.parse_args()
def main() -> None:
    args = parse_args()
    rec = Recommender(
        uri="bolt://localhost:7687",
        user="neo4j",
```

```
    password="password",
    )
    stack = rec.recommend_stack(args.role)
    payload = {"role": args.role, "components": stack}
    if args.output:
        args.output.write_text(json.dumps(payload, indent=2, ensure_ascii=False))
        print(f"Результат збережено до {args.output}")
    else:
        print(json.dumps(payload, indent=2, ensure_ascii=False))


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