Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів»

Виконав: студент II курсу групи 3ПІ-22м
спеціальності 121 – Інженерія програмного забезпечення
Миргородський А.В.
(прізвище та ініціали)

Керівник: к.т.н., доц. каф. ПЗ
Романюк О.В.
(прізвище та ініціали)
«08» грудня 2023 р.

Опонент: к.т.н., доцент каф. ЗІ
Куперштейн Л.М.
(прізвище та ініціали)
«08» грудня 2023 р.

Допущено до захисту
завідувач кафедри ПЗ
д.т.н. проф. Романюк О.Н.
(прізвище та ініціали)
«08» грудня 2023 р.

ВНТУ – 2023

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти ІІ-й (магістерський)
Галузь знань 12 – Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма – Інженерія програмного забезпечення

УЗГОДЖЕНО
Керівник відділу розробки
ПЗ ТОВ «ДСофтвер»
Малик Р.О.
«25» верес 2023 року

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
д.т.н., проф. Романюк О.Н.

«19» вересня 2023 року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Миргородському Андрію Вікторовичу

1. Тема роботи – розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів.

Керівник роботи: Романюк Оксана Володимирівна, к.т.н., доцент кафедри ПЗ, затверджені наказом вищого навчального закладу від «18» вересня 2023 року № 247.

2. Строк подання студентом роботи: 5 грудня 2023 р.

3. Вихідні дані до роботи: модель розробки – ітеративна; метод передачі повідомлень між серверами – виклик віддалених процедур (RPC); вхідні дані – текстові файли в форматі YAML з даними для виконання методів або описом бажаного стану системи; вихідні дані – логи виконання методів та приведення системи в бажаний стан; середовище розробки – JetBrains GoLand; мова програмування – Go.

4. Зміст текстової частини: вступ; аналіз стану питання та постановка задач дослідження, розробка методів та алгоритмів програмного продукту, розробка програмних компонент застосунку, тестування програмного забезпечення, економічна частина, висновки, список використаних джерел, додатки.

5. Перелік ілюстративного матеріалу: титульний слайд; актуальність теми; мета, об’єкт та предмет дослідження; задачі дослідження; наукова новизна; практична цінність одержаних результатів; порівняльний аналіз аналогів; метод і блок-схема алгоритму забезпечення високої доступності; метод і блок-схема алгоритму автоматичної інсталяції програм-агентів; метод і блок-схема

алгоритму автоматичного створення серверів з використанням хмарни[х] провайдерів; структура графічного інтерфейсу користувача; тестуванн[я] програмного забезпечення; дослідження ефективності розробленого метод[у] забезпечення високої доступності; економічна частина; впровадження, апробаці[я] та публікації результатів роботи; фінальний слайд.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		завдання видав	виконанн[я] прийняв
1-4	Романюк О.В. к.т.н, доцент кафедри ПЗ	19.09.2023	05.12.202[3]
5	Причепа І.В., к.е.н., доцент кафедри ЕПВМ	13.11.2023	01.12.202[3]

7. Дата видачі завдання: 19 вересня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської кваліфікаційної роботи	Строк виконання етапів роботи	Примітк[а]
1	Аналіз стану питання та постановка задач дослідження	20.09.2023 – 01.10.2023	Вик.
2	Розробка методів та алгоритмів програмного продукту	02.10.2023 – 09.10.2023	Вик.
3	Розробка програмних компонент застосунку	10.10.2023 – 19.10.2023	Вик.
4	Дослідження ефективності запропонованих методів та тестування програмного забезпечення	20.10.2023 – 12.11.2023	Вик.
5	Економічна частина	13.11.2023 – 01.12.2023	Вик.

Студент ________ **Миргородський А.[...]**
(підпис) (прізвище та ініціа[ли])

Керівник магістерської кваліфікаційної роботи ________ **Романюк О.[...]**
(підпис) (прізвище та ініціа[ли])

АНОТАЦІЯ

УДК 004.42

Миргородський А.В. Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів. Магістерська кваліфікаційна робота зі спеціальності 121 – інженерія програмного забезпечення, освітня програма – інженерія програмного забезпечення. Вінниця: ВНТУ, 2023. 120 с.

На укр. мові. Бібліогр.: 47 назв; рис.: 40; табл.: 15.

У магістерській кваліфікаційній роботі розроблено методи та алгоритми для підвищення ефективності процесів розгортання та масштабування електронних ресурсів, робота яких демонстрована за допомогою розробленого програмного забезпечення для управління конфігураціями.

Подальшого розвитку отримав метод забезпечення високої доступності ПЗ для управління конфігураціями. Удосконалено метод автоматичної інсталяції програм-агентів на керовані сервери. Подальшого розвитку отримав метод автоматичного створення серверів з використанням хмарних провайдерів. Розроблено програмний застосунок для управління конфігураціями, проведено його тестування для перевірки працездатності розроблених методів. Досліджено ефективність розробленого методу забезпечення високої доступності.

Для розробки використано мову програмування Go, середовище розробки JetBrains GoLand, фреймворк gRPC для мережевої комунікації та бібліотеку React для побудови GUI.

Отримані в магістерській кваліфікаційній роботі результати можна використати для побудови високоефективної системи автоматизованого розгортання і масштабування електронних ресурсів.

Ключові слова: управління конфігураціями, висока доступність, кластеризація, відмовостійкість, горизонтальне масштабування, хмарні ресурси.

ABSTRACT

UDC 004.42

Myrhorodskyi A.V. Development of methods and software tools for configuration management to improve the efficiency of deployment and scaling of digital resources. Master's thesis in specialty 121 – software engineering, educational program – software engineering. Vinnytsia: VNTU, 2023. 120 p.

In Ukrainian language. Bibliogr .: 47 titles; fig .: 40; tab .: 15.

In the master's qualification work, methods and algorithms were developed to improve the efficiency of the processes of deploying and scaling digital resources, the work of which was demonstrated using the developed configuration management software.

Further development has been given to the method of ensuring high availability of configuration management software. The method of automatic installation of agent software on managed servers was improved. Further development was given to the method of automatic creation of servers using cloud providers. A software application for configuration management has been developed and tested to verify the efficiency of the developed methods. The effectiveness of the developed method of ensuring high availability has been examined.

During the development, the Go programming language, the JetBrains GoLand development environment, the gRPC framework for network communication, and the React library for building GUI were used.

The results obtained in the master's qualification work can be used to build a highly efficient system for automated deployment and scaling of electronic resources.

Keywords: configuration management, high availability, clustering, fault tolerance, horizontal scaling, cloud resources.

ЗМІСТ

ВСТУП.. 4
1 АНАЛІЗ СТАНУ ПИТАННЯ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ.. 8
1.1 Аналіз стану питання 8
1.2 Порівняльний аналіз аналогів 11
1.3 Аналіз методів для забезпечення високої доступності системи 18
1.4 Аналіз методів для автоматичної інсталяції програм-агентів.................... 21
1.5 Аналіз методів інтеграції з хмарними провайдерами 23
1.6 Постановка задач розробки 27
1.7 Висновки 27
2 РОЗРОБКА МЕТОДІВ ТА АЛГОРИТМІВ ПРОГРАМНОГО ПРОДУКТУ.... 28
2.1 Розробка методу забезпечення високої доступності ПЗ для управління конфігураціями 28
2.2 Розробка методу автоматичної інсталяції програм-агентів на керовані сервери.. 32
2.3 Розробка методу автоматичного створення серверів з використанням хмарних провайдерів... 34
2.4 Розробка блок-схем методів та алгоритмів 37
2.5 Розробка структури графічного інтерфейсу користувача........................... 44
2.6 Висновки 48
3 РОЗРОБКА ПРОГРАМНИХ КОМПОНЕНТ ЗАСТОСУНКУ 50
3.1 Варіантний аналіз і обґрунтування вибору засобів для реалізації програмного засобу 50
3.2 Вибір середовища розробки 56
3.3 Програмна реалізація застосунку 61
3.4 Висновки 73
4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 74
4.1 Аналіз методів тестування програмного забезпечення............................... 74
4.2 Тестування розробленого програмного продукту 77

4.3 Дослідження ефективності розробленого методу забезпечення високої доступності 82
4.4 Розробка інструкції користувача 89
4.5 Системні вимоги 94
4.6 Висновки 94
5 ЕКОНОМІЧНА ЧАСТИНА 95
5.1 Проведення комерційного та технологічного аудиту науково-технічної розробки 95
5.2 Розрахунок витрат на проведення науково-дослідної роботи 98
5.3 Розрахунок економічної ефективності науково-технічної розробки при її можливій комерціалізації потенційним інвестором 106
5.4 Висновки 110
ВИСНОВКИ 112
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 114
ДОДАТКИ 121
ДОДАТОК А Технічне завдання .... **Ошибка! Закладка не определена.**
ДОДАТОК Б Протокол перевірки МКР на плагіат**Ошибка! Закладка не определена.**
ДОДАТОК В Акт впровадження на підприємстві**Ошибка! Закладка не определена.**
ДОДАТОК Г Лістинг коду 128
ДОДАТОК Д Ілюстративна частина 149

ВСТУП

Обґрунтування вибору теми дослідження. Сучасні цифрові системи, що використовуються кінцевими користувачами у повсякденному житті для виконання роботи, дозвілля та відпочинку, потребують постійної підтримки та контролю. Лічені хвилини простою через аварійну ситуацію з інфраструктурою можуть принести великим ІТ-компаніям значні фінансові та репутаційні втрати [1]. Тому, надзвичайно важливими є питання підтримки в робочу стані як апаратної, так і програмної складових електронних ресурсів.

Прості скрипти та програми для автоматизації типових завдань добре підходять лише для простих задач, але мають ряд недоліків, як-от складність підтримки та масштабування. Через це виник окремий клас програмного забезпечення, що призначений для управління конфігураціями. Основна мета застосунків для управління конфігураціями – це привести надану систему (набір з одного або більше серверів, що можуть виконувати різні ролі та вимагати встановлення і конфігурацію різних компонентів) в бажаний стан [2]. Цей бажаний стан може включати в себе як просту інсталяцію програмний компонент, так і більш складні задачі, як-от динамічна генерацію конфігураційних файлів, зміна параметрів ОС або реєстру тощо.

Програмне забезпечення для управління конфігураціями дозволяє швидко перетворити набір нових чистих серверів в повністю функціонуючу систему, що готова до експлуатації, з мінімальними витратами часу та без втручання користувача ПЗ, а стандартизація усіх задач по автоматизації у форматі одного програмного продукту дозволяє зменшити часові та фінансові витрати на підтримку [3].

Проте, сучасні електронні ресурси та обчислювальні системи розвиваються в сторону подальшого зменшення можливого впливу аварійних ситуацій та непрацездатності окремих серверів. Через це поступово з’явилися такі терміни, як висока доступність, горизонтальне масштабування, відмовостійкість тощо. Дедалі більше систем відмовляються від монолітної

архітектури та намагаються частково або повністю децентралізувати основні потоки виконання, що дозволяє збільшити продуктивність ПЗ шляхом розподілення задач між декількома окремими машинами, а не за допомогою зміни сервера на більш потужний [4].

Класичне програмне забезпечення для управління конфігураціями погано пристосоване для таких динамічних сценаріїв, а також рідко надає варіанти забезпечення високої доступності – самі сервери, що контролюють процес приведення системи в бажаний стан, залишаються вразливими до будь-яких збоїв та аварійних ситуацій [5].

Саме тому розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів є досить актуальною задачею.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалась відповідно до плану науково-дослідних робіт кафедри програмного забезпечення.

Мета та завдання дослідження. Метою роботи є підвищення ефективності процесів розгортання та масштабування електронних ресурсів.

Для цього необхідно виконати такі **завдання**:

– провести аналіз існуючих методів підвищення ефективності процесів розгортання та масштабування електронних ресурсів;

– розробити метод і алгоритм для забезпечення високої доступності ПЗ для управління конфігураціями;

– розробити метод і алгоритм для автоматичної інсталяції програм-агентів на керовані сервери;

– розробити метод і алгоритм для автоматичного створення серверів з використанням хмарних провайдерів;

– розробити графічний інтерфейс користувача для створюваного ПЗ;

– розробити програмний застосунок для управління конфігураціями;

– провести тестування програмного застосунку;

– дослідити ефективність методу забезпечення високої доступності;

– розробити інструкцію користувача.

В результаті виконання перелічених завдань буде розроблено програмне забезпечення для управління конфігураціями, що в повній мірі зможе демонструвати реалізацію запропонованих методів та за їх допомогою підвищити ефективність процесів розгортання та масштабування ресурсів.

Об'єкт дослідження – процеси управління конфігураціями при розгортанні та масштабуванні електронних ресурсів.

Предмет дослідження – методи та засоби підвищення ефективності процесів розгортання та масштабування електронних ресурсів.

Методи дослідження. У процесі дослідження використовувались: комплексний аналіз з метою визначення недоліків існуючих технічних рішень задачі та подальший синтез отриманих даних для формування нових функціональних характеристик і вимог; теорія алгоритмів для розробки та вдосконалення алгоритмів ПЗ; комп'ютерне моделювання для аналізу та перевірки отриманих теоретичних положень; методи обробки експертної інформації для дослідження ефективності розроблених методів.

Наукова новизна одержаних результатів.

1. Подальшого розвитку отримав метод для забезпечення високої доступності, який, на відміну від існуючих, було адаптовано та розширено шляхом застосування технік кластеризації, балансування навантаження та алгоритму консенсусу для застосування в програмному забезпеченні для управління конфігураціями, що дозволило керувати станом системи без застосування центрального сервера та продовжувати роботу при непрацездатності частини вузлів.

2. Удосконалено метод автоматичної інсталяції програм-агентів на керовані сервери, що на відміну від інших рішень застосовує гібридну комунікацію з вузлами системи та дозволяє автоматизувати типові дії кінцевого користувача програмного забезпечення по підготовці кластера.

3. Подальшого розвитку отримав метод автоматичного створення серверів з використанням хмарних провайдерів, що на відміну від аналогів дозволяє не

тільки створити нові хмарні ресурси, а й відразу застосовувати їх у складі інфраструктури програмної системи для управління конфігураціями, що дозволило автоматизувати розгортання кластера.

Практична цінність отриманих результатів. Практична цінність одержаних результатів полягає в тому, що на основі отриманих в магістерській кваліфікаційній роботі теоретичних положень запропоновано алгоритми та програмні засоби для підвищення ефективності процесів розгортання та масштабування електронних ресурсів.

Впровадження. Впровадження результатів досліджень підтверджується відповідним актом та використовується в організації ТОВ «ДСофтвер» для підвищення ефективності розгортання цифрової інфраструктури.

Особистий внесок здобувача. Усі наукові результати, викладені у магістерській кваліфікаційній роботі, отримані автором особисто. У роботах, опублікованих у співавторстві, здобувачу належить аналіз методів для управління конфігураціями при розгортанні електронних ресурсів [5]; розробка розподілених систем з використанням алгоритму консенсусу Raft [6]; розробка методу забезпечення високої доступності для програмного забезпечення управління конфігураціями [7].

Апробація матеріалів магістерської кваліфікаційної роботи. Результати роботи доповідалися на:

– Міжнародній науково-практичній Інтернет-конференції «Електронні інформаційні ресурси: створення, використання, доступ» (Суми/Вінниця, 2022);

– LII Науково-технічній конференції факультету інформаційних технологій та комп'ютерної інженерії (Вінниця, 2023).

Публікації. Основні результати досліджень опубліковано в 3 наукових працях, у тому числі 1 стаття у фаховому виданні України, 2 – у матеріалах конференцій.

1 АНАЛІЗ СТАНУ ПИТАННЯ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ

1.1 Аналіз стану питання

Сучасні електронні ресурси, обчислювальні системи та різноманітні онлайн-сервіси є дуже чутливими до різноманітних перебоїв, затримок та інших проблем з працездатністю [8]. Для компанії або організації, наприклад, помилка в програмному забезпеченні може принести значні репутаційні ризики, а непрацююча веб-платформа викличе фінансові збитки, що пропорційні кожній хвилині недоступності сервісу для клієнтів.

Високі вимоги до надійності електронних ресурсів та стабільності їх роботи призвели до появи цілого сегменту методик та підходів для вирішення цих задач. До них можна віднести високу доступність, горизонтальне масштабування, надмірність системних компонентів та багато інших. Це, в свою чергу, створило нову проблему: більшість цих понять передбачають використання дублюючого апаратного й програмного забезпечення, щоб у випадку непрацездатності одного елементу його міг швидко замінити інший, а отже розробникам і DevOps-інженерам треба підтримувати в роботі набагато більшу кількість серверів.

В залежності від типу та характеру виконуваних задач обслуговування робочих машин може сильно відрізнятися. В одному випадку, можна використовувати хмарних провайдерів та поширений для них підхід до роботи з ресурсами: інфраструктура повинна бути гнучкою і не покладатися на жорсткі обмеження щодо серверів та інших компонентів системи [9]. Саме тому популярні провайдери AWS, GCP, Azure та інші дозволяють створити нову віртуальну машину на основі знімку ОС з усім необхідним ПЗ лише за лічені хвилини. В такому випадку можна швидко замінити будь-який складовий компонент електронного ресурсу та використовувати інші корисні можливості, як-от автоматичне масштабування. Таким чином, такі короткострокові навантаження рідко вимагаються додаткової автоматизації чи обслуговування.

В певних ситуаціях необхідно вести роботу з апаратними, а не віртуальними серверами. Зазвичай це стається тоді, коли необхідно виконувати значні довготривалі навантаження або підтримувати більш низькорівневу інфраструктуру – систему віртуалізації для інших серверів, контейнерні додатки та інше. Тоді має сенс максимально довго підтримувати ці сервери в робочому стані, але це буде вимагати додаткового обслуговування. В такому випадку логічно скористатися певним інструментом автоматизації або керування конфігураціями.

Іншим прикладом використання програмного забезпечення для управління конфігураціями може бути інсталяція комплексної системи або програмного продукту. Наприклад, якщо це багатокомпонентна обчислювальна платформа, то процес її встановлення на один або декілька серверів може включати в себе розпаковку декількох десятків окремих програм-складових, генерацію комплексних конфігураційних файлів, інсталяцію та налаштування бази даних та багато інших завдань. Виконання усіх цих маніпуляцій вручну може вимагати до декількох годин часу, а розробка внутрішнього інструменту автоматизації задач лише для одного продукту є окремим часо- та ресурсозатратним проєктом.

Подібні інструменти для автоматизації можна віднести до програмного забезпечення для управління конфігураціями. Типовими задачами такого ПЗ є створення нових ресурсів (особливо якщо це інструмент для роботи з хмарними сервісами), установка різних додатків, генерація конфігураційних та прикладних файлів, виконання різноманітних додаткових інструкцій та інше. Часто всі ці дії об'єднують під єдиною назвою процесу – приведення системи в бажаний стан.

Використання програмного забезпечення для управління конфігураціями має відразу декілька переваг. По-перше, більшість таких інструментів підтримують підхід «Інфраструктура як код» (англ. Infrastructure as Code, IaC), що передбачає збереження бажаного стану системи у вигляді коду або певного іншого конфігураційного файлу, який кінцевий інструмент зможе використати в процесі виконання [10]. Це може бути як просто набір інструкцій для автоматичного застосування на серверах, так і більш комплексний формат, що

буде дозволяти гнучко підлаштовувати виконання завдань під конкретний сервер.

Очевидні переваги використання підходу IaC – це можливість повторного використання вже створених конфігурацій бажаного стану. В добре спроектованій системі це дозволить за лічені хвилини підготувати аналогічний до вже існуючого сервер – налаштувати на ньому всі необхідні параметри ОС, встановити програмне забезпечення, внести конфігураційні правки в ті чи інші застосунки.

Використання IaC також дозволяє простіше відслідковувати зміни в бажаному стані системи та контролювати їх при роботі в великих командах – для цього можна використовувати будь-яку систему контролю версій (СКВ, англ. source code management, SCM), текстовий файл у довільному форматі буде просто відслідковувати в ній. Якщо ж додати до цього ще й автоматизований запуск ПЗ для управління конфігураціями при виявленні змін в SCM, то така система вже буде слідувати базовим принципам GitOps – цілого фреймворку для керування CI/CD інфраструктурою, що використовує SCM в якості єдиного джерела правди при конфліктах конфігурацій.

Варто розуміти, що в випадку використання статичного контенту на серверах, тобто коли немає необхідності динамічно генерувати якісь дані або вносити зміни для налаштування машини – тоді ПЗ для управління конфігураціями не дасть суттєвого виграшу в швидкості. Кращим варіантом буде просто зробити знімок файлової системи та на його основі підготовлювати нові сервери (особливо, якщо робота ведеться з використанням засобів віртуалізації).

Якщо необхідно виконати завдання динамічного характеру після першочергового налаштування серверів – тут інструменти для автоматизації будуть підходити найкраще. Прикладом може бути налаштування програмного забезпечення у режимі високої доступності (наприклад, кластеру Neo4j – графової бази даних). Велика кількість таких застосунків задля кращих гарантій безпеки не дозволяють динамічно під’єднати нові вузли в кластер – їх необхідно спочатку описати в конфігураційних файлах і лише потім ініціювати програмне

забезпечення. Сучасні інструменти для управління конфігураціями дозволяють в повністю автоматичному режимі вирішити цю проблему: підключитися до кожної віддаленої машини, зупинити необхідні сервіси та/або додатки, з використанням шаблонізатора згенерувати нові конфігураційні файли – при цьому динамічно отримавши інформацію про весь список серверів, запустити попередньо зупинене ПЗ та зробити додаткові перевірки, щоб запевнитись в працездатності нової конфігурації системи.

Таким чином, програмне забезпечення для управління конфігураціями – це багатофункціональний інструмент для автоматизації задач в різних сценаріях розгортання та підтримки електронних ресурсів. Використання принципів «Infrastructure as Code», керування бажаними станами системи та GitOps підходу дозволяє вирішувати проблеми динамічного налаштування серверів.

1.2 Порівняльний аналіз аналогів

Сфера програмного забезпечення для управління конфігураціями має як своїх уже закріплених лідерів, так і багато нових продуктів, що намагаються привернути увагу користувачів за допомогою додаткового функціоналу, який націлений на специфічні проблеми окремих груп споживачів. Цільові клієнти групи ПЗ для керування конфігураціями – це системні адміністратори, SRE-інженери, DevOps-інженери та інші R&D-співробітники, які залучені в процес побудови й підтримки інфраструктури певного продукту або ж мають обширні задачі по автоматизації процесів.

Для порівняльного аналізу було обрано наступні програмні продукти:

– Ansible;

– Puppet;

– Chef;

– SaltStack.

Це популярні програмні рішення та впевнені лідери ринку в цій сфері. Їх аналіз дозволить визначити критично-важливий функціонал та найбільш значні недоліки для ПЗ для управління конфігураціями, які можна буде використати

при розробці власних методів та алгоритмів. Розглянемо більш детально індивідуальні особливості кожного програмного рішення.

Ansbile – це популярне відкрите програмне забезпечення, що з'явилося в 2012 році в компанії AnsibleWorks та пізніше було викуплене Red Hat, після чого отримало платну версію Ansible Tower з додатковим функціоналом, GUI та іншими інструментами, що в основному орієнтовані на корпоративних клієнтів.

На рисунку 1.1 наведено інтерфейс Ansible Tower.

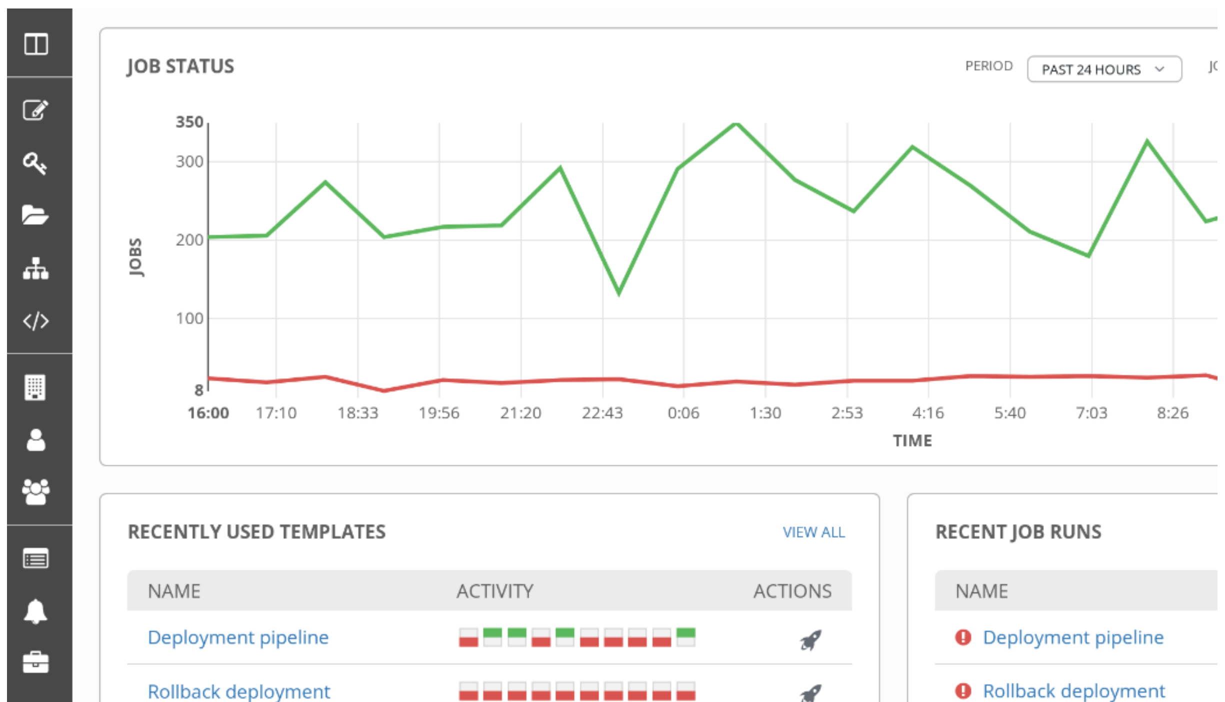


Рисунок 1.1 – Інтерфейс програмного забезпечення Ansible Tower

Ansible є значно молодшим за інші популярні продукти, але він швидко отримав популярність за допомогою своєї внутрішньої архітектури [11]: використання Push-моделі (ПЗ необхідно встановити лише на керуючий сервер, машини-клієнти потребують мінімального додаткового налаштування), модулів на мові Python та широко-підтримуваного протоколу SSH дозволило отримати репутацію швидкого та легкого в використанні інструменту. З іншого боку, погана підтримка не Unix-подібних операційних систем та недостача певного функціоналу в open-source версії продукту може стати на заваді для деяких клієнтів.

Puppet – більш старий інструмент, що з’явився у 2005 році та розроблявся однойменною компанією Puppet аж до 2022 року, коли вона була викуплена разом з усіма програмними продуктами іншою компанією Perforce Software.

Puppet розроблено з використанням Ruby, а для опису бажаного стану системи використовується DSL (англ. domain-specific language, предметно-орієнтована мова) на основі цієї мови програмування. Для міжсерверної взаємодії він використовує Pull-модель – керовані сервери вимагають інсталяції окремих програм-агентів, які будуть регулярного перевіряти керуючий сервер на наявність нової конфігурації для них [12].

Першочергово це також було програмне забезпечення з відкритим кодом, але аналогічно до Ansible пізніше з’явилася комерційна версія Puppet Enterprise з додатковим функціоналом. Puppet є більш поширеним і добре підтримуваним інструментом, він має більш розвинутий користувацький інтерфейс та систему збору даних, але є складнішим у використанні та вивченні, тому рідко використовується для невеликих задач і проєктів.

Інтерфейс Puppet Enterprise наведено на рисунку 1.2.

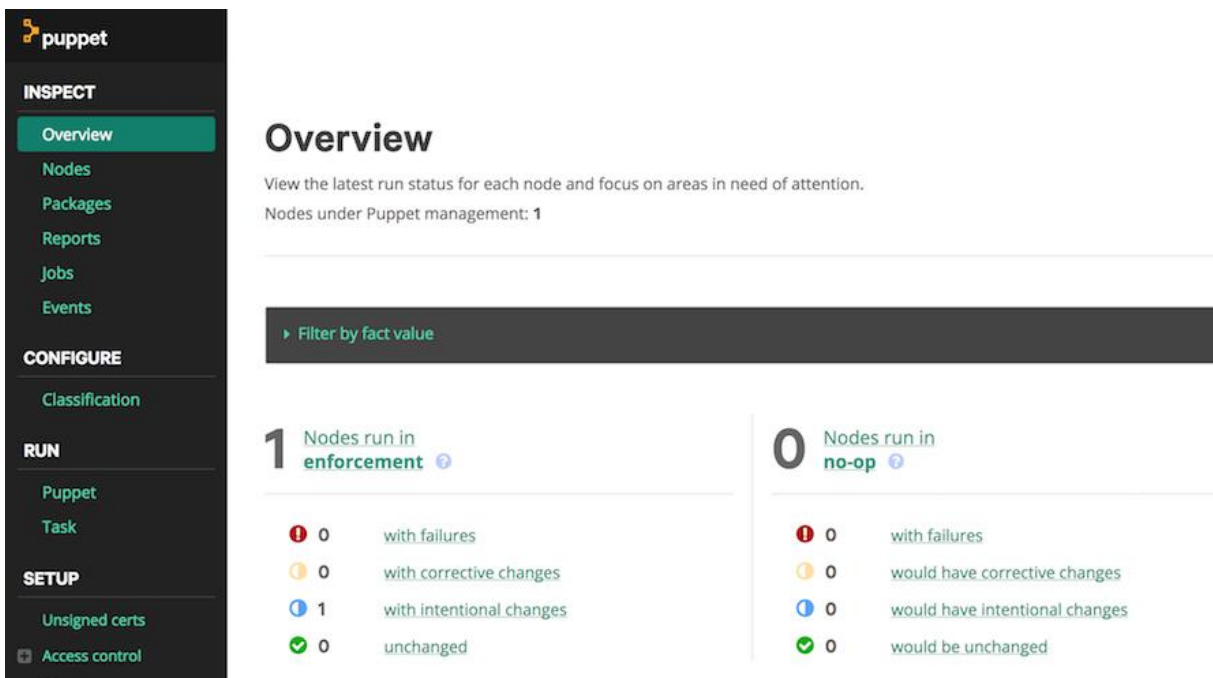


Рисунок 1.2 – Інтерфейс програмного забезпечення Puppet Enterprise

Chef – програмне забезпечення для управління конфігураціями, що з’явилося в 2009 році і розроблялося компанією Chef Software до 2020 року, коли її купила Progress Software Corporation. З грудня 2021 року цей продукт також називають Progress Chef. Аналогічно до конкурентів наявна платна версія Chef Automate з додатковим функціоналом.

Користувацький інтерфейс Chef Automate наведено на рисунку 1.3.

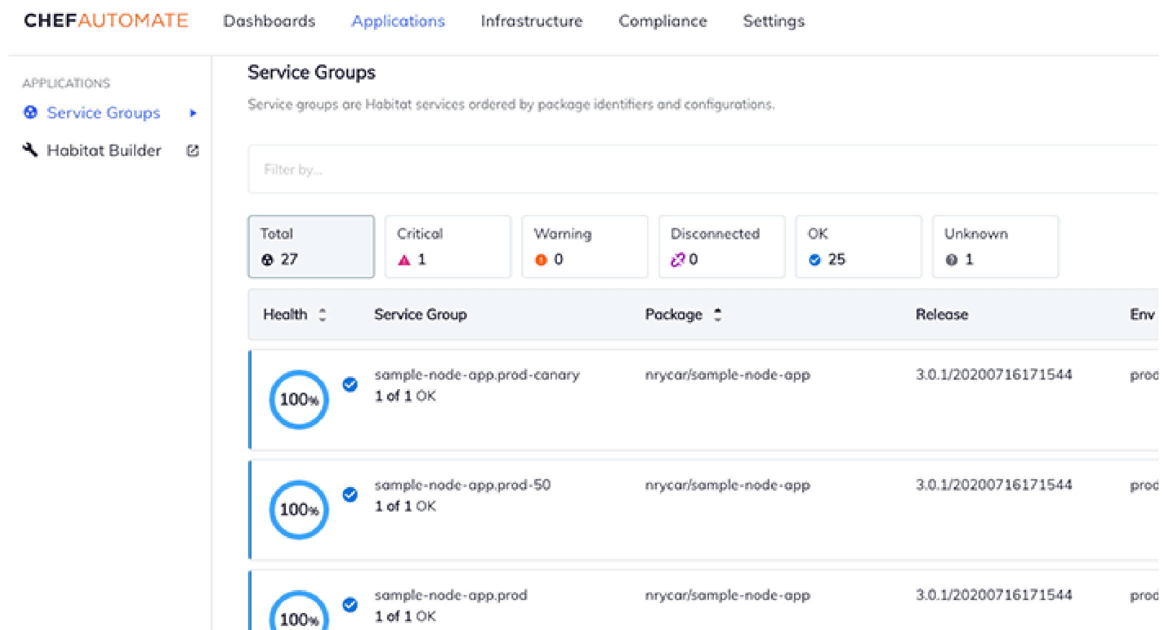


Рисунок 1.3 – Інтерфейс програмного забезпечення Chef Automate

Chef також розроблено на мові програмування Ruby, але пізніше деякі серверні компоненти було переписано на Erlang. Аналогічно до Puppet використовується власна DSL на основі Ruby для опису бажаного стану системи та Pull-модель для міжсерверної взаємодії [13], але загальна архітектура дещо відрізняється. Окрім керуючого сервера (в термінології Chef – це Chef Server) та набору керованих серверів (Chef Nodes) також використовується одна чи декілька клієнтських машин (Chef Workstation). Workstation – це машина кінцевого інженера, який за допомогою CLI та набору інших інструментів розробляє конфігурації для Chef, після чого готові дані надсилаються і зберігаються на Chef Server. Керовані ноди в свою чергу з певною періодичністю опитують керуючий сервер на наявність нової конфігурації для них. Основна

відмінність – це відсутність прямого доступу розробників до керуючого сервера, всі необхідні дії можна виконати за допомогою готових інструментів з інших машин.

Більш комплексна архітектура Chef є його основною перевагою, вона може краще підходити для систем з частими мережевими проблемами або строгими безпековими вимогами. З іншого боку, така ускладненість та використання Ruby з власним DSL робить Chef складним для вивчення початківцями.

SaltStack – програмний продукт, що з'явився трохи раніше за Ansible, в 2011 році. На момент створення був більш відомий під короткою назвою Salt та лише через певний час отримав додаткові модулі, як-от Salt Cloud, й продукт почали називати SaltStack (іноді SaltStack Platform). Також існує комерційна версія SaltStack Enterprise, але на відміну від конкурентів вона орієнтована не на розширення базового функціоналу, а на інтеграцію з розробником ПЗ для віртуалізації VMware.

На рисунку 1.4 зображено графічний інтерфейс SaltStack Enterprise.

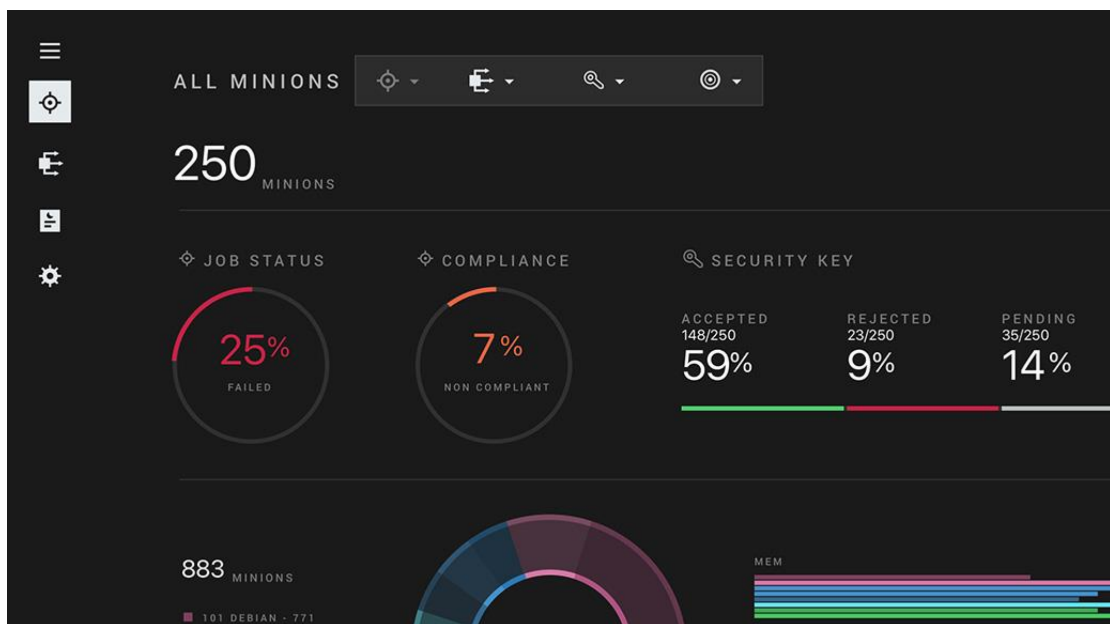


Рисунок 1.4 – Інтерфейс програмного забезпечення SaltStack Enterprise

Архітектура SaltStack відрізняється від аналогів – використовується Push-модель для міжсерверної взаємодії, як це реалізовано в Ansible, але аналогічно

до Puppet та Chef необхідно встановлювати додатки-агенти на керовані ноди [14]. Причина полягає в використанні ZeroMQ – асинхронної бібліотеки обміну повідомлень, що дозволяє SaltStack керувати значно більшою кількістю серверів одночасно, ніж при застосуванні вбудованих в ОС протоколів зв'язку. Крім того, підтримується робота в режимі високої доступності, коли застосовується декілька керуючих серверів. Продукт розроблено з використанням мови програмування Python, а для опису конфігурацій використовується YAML.

Основною перевагою SaltStack є його швидка, надійна і високо-масштабована архітектура, але на противагу стає більш складний процес інсталяції та конфігурації, а також більш комплексний процес вивчення і використання цього інструменту.

Після дослідження усіх аналогів було визначено список критеріїв для їх оцінювання в порівнянні з власним програмним продуктом «Detrint». Результати порівняння наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльні характеристики програмних продуктів

Критерій	Ansible	Puppet	Chef	SaltStack	Detrint
Підтримка режиму високої доступності	-	-	-	+	+
Можливість працювати без встановлення агентів на сервери	+	-	-	-	-
Автоматична інсталяція агентів	+/-	-	-	-	+
Підтримка non-UNIX-подібних ОС в якості керуючого сервера	-	-	-	-	+
Підтримка роботи з хмарними провайдерами	+/-	+/-	+/-	+	+
Автономне підтримування бажаного стану системи	-	+	+	-	-
Підсумок	2	1,5	1,5	2	4

Розглянемо більш детально обрані для оцінювання критерії. Підтримку режиму високої доступності з існуючих аналогів має лише SaltStack, який може працювати з декількома керуючими серверами й переключатися між ними у випадку непрацездатності поточної ноди-лідера.

Лише Ansible може працювати без встановлення агентів на сервери, адже підтримує протоколи SSH та WinRM, а після з'єднання з керованою машиною може використовувати вбудовані в ОС інструменти для приведення системи в бажаний стан. Жоден з розглянутих продуктів-аналогів не здатен встановлювати додатки-агенти на машини самостійно, а Ansible отримує за цим критерієм половину балу – він працює без використання агентів, але в деяких випадках (особливо при використанні UNIX-подібних ОС) може вимагати додаткового встановлення Python на керовані ноди для правильної роботи окремих модулів.

Всі популярні програмні продукти для управління конфігураціями підтримують роботу з MS Windows, MacOS, Linux та деякими іншими UNIX-подібними ОС, але не можуть використовувати Windows або Windows Server в якості керуючих машин.

Всі розглянуті програмні продукти мають підтримку хмарних провайдерів, але лише в SaltStack вона представлена офіційно – решта ПЗ має сторонні модулі для цього, що розробляються і підтримуються спільнотою. Puppet та Chef підтримують Pull-метод для взаємодії між керуючим і керованими серверами, тому такі системи можуть працювати й повністю автономно – машини самостійно, без команди користувача, можуть отримати нову конфігурацію в будь-який час від керуючого сервера.

Таким чином, в підрозділі було детально розглянуто популярні програмні продукти для управління конфігураціями, такі як Ansible, Puppet, Chef та SaltStack. Вивчено їх функціонал, переваги та недоліки. Було відібрано ряд критеріїв для порівняння характеристик цих програмних продуктів та потенційної власної розробки під назвою «Detrint». Згідно цього аналізу Detrint за обраними критеріями отримав 4 бали й він на 50% краще за Ansible та SaltStack, які отримали по 2 бали, а також на 62,5% краще за Puppet та Chef, що

отримали по 1,5 бали. Можна зробити висновок, що розробка власного програмного продукту є доцільною та дозволить конкурувати з існуючими аналогами.

1.3 Аналіз методів для забезпечення високої доступності системи

Висока доступність – це здібність системи протистояти апаратним і програмним збоям та забезпечувати працездатність і можливість обробки запитів користувачів навіть при відмові частини компонентів [15]. Для забезпечення високої доступності використовують цілий ряд різноманітних підходів:

– кластеризація та надлишковість;

– резервне копіювання та автоматичне відновлення;

– масштабованість;

– балансування навантаження;

– моніторинг та автоматизація типових задач.

Розглянемо більш детально окремі підходи.

Кластеризація (англ. clustering) – це підхід, де використовують декілька фізичних або віртуальних серверів для забезпечення високої доступності системи або окремих її сервісів [16]. Сервери в кластері можуть працювати по-різному в залежності від реалізації режиму високої доступності: це може бути просте дублювання ресурсів або ж повноцінне розподілення навантаження між всіма вузлами кластеру.

При дублюванні зазвичай один сервер виступає в якості головного – він приймає запити і на читання даних, і на їх модифікацію. Решта вузлів або зовсім не беруть участі в обробці запитів, або ж опрацьовують лише операції вичитування даних – таким чином лише головний сервер має можливість змінювати стан системи і містить гарантовано коректну версію всіх даних. Другорядні вузли можуть створюватися шляхом резервного копіювання і відновлення або ж постійною синхронізацією даних між лідером кластера та іншими серверами – такий підхід зазвичай використовується в системах управління базами даних з підтримкою високої доступності. В деяких випадках

використовується комбінація обох способів – це залежить лише від обраної стратегії відмовостійкості.

Повноцінне розподілення навантаження передбачає одночасну роботу всіх або частини вузлів кластеру, але це не означає, що відразу декілька машин можуть бути лідерами системи – зазвичай окремі сервери або виконують окремі відведені для них ролі, або ж якась внутрішня або зовнішня система відповідає за розподіл навантаження [17]. До другого способу також можна відвести безсерверні обчислення (англ. serverless computing). Через технічні особливості їх механізму роботи неможливо локально зберігати довгострокові дані, тому для цього використовують окремі сервіси для роботи з кешем або ж зовнішні БД. Таким чином, з використанням безсерверних обчислень загальна архітектури системи частково ускладняється, але її масштабування стає набагато простішим.

Крім того, кластеризацію з розподіленням навантаження можна віднести до горизонтального масштабування – підходу до підвищення продуктивності та надійності системи шляхом додавання нових ресурсів. Вертикальне масштабування, з іншого боку, передбачає просте збільшення характеристик серверів, що має фізичні ліміти в вигляді доступних апаратних ресурсів. Горизонтальне масштабування дозволяє значно гнучкіше масштабувати обчислювальну систему, але вимагає відповідну програмну підтримку.

Резервне копіювання та автоматичне відновлення – це важливі процеси для високої доступності, але окремо від інших підходів вони не можуть забезпечити відмовостійкість без повної або часткової недоступності системи. Резервне копіювання будь-яких важливих даних з БД або ж елементів інфраструктури (знімки ОС, файли для IaC та інше) є хорошою практикою, особливо важливою для production-систем і додатків [18]. В комбінації з механізмами автоматичного відновлення можна отримати хороші значення RTO (англ. Recovery Time Objective, цільовий час відновлення – допустимий час простою сервісу в разі збою) та RPO (англ. Recovery Point Objective, цільова точка відновлення – допустимий обсяг втрати даних у разі збою), що дозволять покращити загальну відмовостійкість і високу доступність продукту.

Масштабованість (англ. scalability) – це здатність системи збільшувати власні параметри для обробки зростаючої кількості даних і запитів [19]. Важливою вимогою масштабованості є уникнення будь-яких втрат якості обробки даних – зовнішній користувач при збільшенні навантаження на систему не повинен спостерігати значних змін в її поведінці. Для забезпечення необхідного рівня продуктивності в будь-який момент часу зазвичай використовують горизонтальне або вертикальне масштабування. Більшість хмарних провайдерів мають власні сервіси та інструменти для підтримки горизонтального масштабування, такі як AWS Auto Scaling Groups, Google Cloud Managed Instance Groups, Azure Autoscale та інші.

Балансування навантаження (англ. load balancing) – це ще один підхід, який зазвичай використовується в поєднанні з іншими методиками реалізації високої доступності. Це процес розподілу запитів або іншого типу навантаження між декількома фізичними або віртуальними серверами, точками входу, компонентами системи тощо [20].

Для балансування навантаження може використовуватися як спеціалізоване програмне забезпечення, так і апаратні рішення, які можуть забезпечити ще більшу пропускну здатність системи та працювати на нижчих рівнях мережевої моделі OSI. Типовий приклад використання – це застосувати балансувальник навантаження в якості вхідної точки для кластеру, в якого декілька вузлів одночасно можуть обробляти запити. Сучасні програмні або апаратні рішення вміють не тільки розподіляти запити, а й забезпечувати додаткові перевірки працездатності вузлів (health checks) та підтримку довготривалих сесій (session stickiness, sticky session, session affinity) [21].

Моніторинг та автоматизація типових задач є важливою частиною підтримки в працездатному стані будь-якої системи з високою доступністю. Правильно налаштована система моніторингу дозволить вчасно дізнатися про проблеми з кластером або ж виявити чинники, що впливають на продуктивність його роботи [22]. Наприклад, збір і аналіз даних про завантаженість вузлів та параметри горизонтального масштабування можуть допомогти оптимізувати

програмне забезпечення (знайти більш завантажені компоненти, ліквідувати вузькі місця додатку тощо) та зменшити витрати на сервери в тому випадку, коли попередньо не виявлені проблеми призводять до використання зайвих ресурсів. Інструменти для автоматизації задач можуть керувати додатковими системами сповіщення або ж вирішувати типові задачі на основі зібраних даних від систем моніторингу, звільняючи цим додатковий час для DevOps-інженерів та SRE-спеціалістів.

Таким чином, в даному підрозділі розглянуто ряд підходів та методик, що застосовуються для забезпечення високої доступності системи. Реалізація цих методів або їх комбінації та способи застосування можуть сильно відрізнятися в залежності від типу й задач обчислювальної системи або програмного продукту, але на основі проаналізованих даних можна розробити вдосконалений метод забезпечення високої доступності для програмного забезпечення для управління конфігураціями.

1.4 Аналіз методів для автоматичної інсталяції програм-агентів

Програмні продукти для управління конфігураціями, що використовують додаткові програми-агенти для взаємодії між керуючим та керованими серверами, зазвичай мають кращі показники швидкодії та продуктивності, адже можуть більш гнучко оптимізувати мережеву активність. Основним мінусом такого підходу є необхідність додаткової конфігурації агентів на серверах перед тим, як можна буде почати використовувати ПЗ для управління конфігураціями. Розглянемо більш детально існуючі методи та підходи, що дозволяють спростити й автоматизувати цей процес.

Для віддаленого доступу часто використовують системи дистанційного керування – це може бути RDP (Remote Desktop Protocol) для серверів на базі Windows Server, VNC (Virtual Network Computing) для UNIX-подібних систем або ж інше стороннє програмне забезпечення [23]. Але такі рішення в основному призначені для використання людьми-інженерами, а не для програмного доступу чи автоматизації, тому в основному використовуються для вирішення нетипових

задач або більш зручної взаємодії з ПЗ, що має лише графічний інтерфейс і не підтримує роботу в режимі CLI. Крім того, такий тип віддаленого доступу отримав поширення лише на ОС Windows Server, коли ж на серверах з UNIX-подібними операційними системами рідко застосовують GUI і основний спосіб доступу – це застосування SSH.

Застосування сторонніх інструментів для автоматизації в даному контексті не є актуальним, адже стоїть задача простої одноразової інсталяції та налаштування агентів, після чого всі подальші роботи по керуванню конфігураціями системи повинні виконуватися через них. Тому, розглянемо більш детально способи відділеного доступу, що використовуються існуючими продуктами для управління конфігураціями без програм-агентів, а саме протоколи SSH та WinRM.

SSH (Secure Shell) – це мережевий протокол для віддаленого доступу та тунелювання TCP-трафіку, що був першочергово розроблений в 1995 році та далі продовжує розвиватися під версією SSH-2. Він спеціально проектувався в якості криптографічного протоколу, що може забезпечити безпечне з’єднання між клієнтом і сервером навіть через незахищену мережу.

SSH підтримує асиметричне шифрування даних та декілька способів автентифікації, що на відміну від звичайного застосування комбінації логіна та пароля може підвищити безпеку системи [24]. Крім дистанційного доступу цей протокол також часто використовують для перенаправлення портів між двома вузлами, що дозволяє захистити незашифрований TCP трафік без застосування TLS або ж тимчасово надати доступ до локальних портів сервера без додаткових мережевих налаштувань.

Основні недоліки SSH полягають в складності його налаштування, особливо при застосуванні більш комплексних способів авторизації, а також орієнтованості на досвідчених користувачів, що менш актуально при застосуванні цього протоколу для автоматизації або інших задач в DevOps-сфері. Крім того, конфігурація SSH на різних ОС може сильно відрізнятися – MS Windows лише з останніми версіями отримала підтримку SSH.

WinRM (Windows Remote Management) – це інший протокол для віддаленого доступу, що був спеціально розроблений для ОС Windows. На відміну від SSH він базується на протоколі HTTPS і в деяких випадках може показувати гіршу швидкість роботи, але він добре інтегрований в Windows і довгий час був єдиним офіційним способом для програмного віддаленого доступу для цієї ОС [25].

Старі версії WinRM були менш стабільними, особливо в середовищах з ненадійним мережевим з'єднанням, але ситуація змінилася після появи PowerShell Remoting, який працює з більш новими редакціями WinRM. Цей протокол аналогічно до SSH може бути складним в конфігуруванні, але він не має таких функцій, як перенаправлення портів і деяких більш комплексних способів авторизації. В цілому, цей протокол менш популярний через орієнтацію на Windows, але він широко використовуються для серверів на базі цієї ОС.

Таким чином, в даному підрозділі було розглянуто популярні протоколи та інші способи для віддаленого доступу до серверів, які можна застосувати для автоматичної інсталяції і налаштування програм-агентів. Протоколи SSH та WinRM мають певні недоліки, але на їх основі можна розробити вдосконалений метод для встановлення агентів для ПЗ для управління конфігураціями.

1.5 Аналіз методів інтеграції з хмарними провайдерами

Більшість хмарних провайдерів надають відразу декілька способів роботи зі своїми сервісами та ресурсами. Окрім дружелюбного до користувача графічного інтерфейсу зазвичай ще можна скористатися CLI або ж відповідними API та SDK. Розглянемо всі можливі варіанти інтеграції з хмарними провайдерами для створення нових серверів.

Майже кожен хмарний провайдер має власний зручний графічний інтерфейс для роботи з їх сервісами та створення і конфігурації відповідних ресурсів. Наприклад, Amazon Web Services має AWS Management Console, GCP надає доступ через Google Cloud Console, а Microsoft Azure – за допомогою Azure Portal. З їх використанням можна на будь-якому пристрої отримати доступ

до WebUI з основним функціоналом, що надає хмарний провайдер. Графічний інтерфейс є незамінним як для досвідчених користувачів, так і для новачків, адже він дозволяє не тільки в більш простому форматі вивчати можливості хмари, а й більш вільно експериментувати при проектуванні нової архітектури й інфраструктури для обчислювальної системи [26]. З іншого боку, в деяких випадках присутні додаткові обмеження зі сторони графічного інтерфейсу – доступ до нового функціоналу або експериментальних сервісів зазвичай надають в першу чергу через CLI або API/SDK і вже після цього розробляють інтеграцію з GUI, тому може бути присутня певна затримка в часі.

Розробка повноцінної інтеграції між програмним забезпеченням та GUI для хмарного провайдера є технічно можливою, але нераціональною задачею через перспективу застосування відповідних API або SDK, що спеціально для цього призначені. Тому, GUI підходить лише для створення серверів в ручному режимі й не може бути застосований для автоматизації в складі ПЗ для управління конфігураціями.

CLI (Command Line Interface) може застосовуватися як в ручному режимі для виконання певних операцій, так і для автоматизації в деяких окремих випадках. Більшість хмарних провайдерів аналогічно до підтримки GUI також розповсюджують і підтримують роботу через свої власні CLI: AWS CLI, gcloud CLI, Azure CLI та інші. Їх основне призначення – це надати більш досвідченим користувач можливість швидко виконати типові операції без потреби взаємодії з WebUI або іншими інструментами [27]. Наприклад, створення віртуальної машини в AWS за допомогою CLI займає лічені секунди та декілька введених команд, але по цій же причині такий варіант не підходить початківцям – він вимагає досвіду як роботи з хмарним провайдером, так і з самим CLI.

В деяких випадках CLI викликають не в ручному режимі, а програмно з іншого додатку. В такому підході є свої переваги: іноді треба виконати лише декілька операцій з хмарними ресурсами й повноцінна інтеграція з SDK або API хмарного провайдера є нераціональною задачею з точки зору часо- та трудовитрат. Тому, більш простим варіантом є виклик зовнішнього інструменту

для роботи з хмарою та розробка простої обгортки для роботи з ним. Особливо часто це застосовують при створенні допоміжних програм і скриптів, що виконуються на віртуальних машинах хмарних провайдерів – в більшості випадків вони вже мають інсталяцію відповідних CLI та базові безпекові налаштування доступу, тому немає потреби самостійно вирішувати ці питання.

При інтеграції CLI з програмним забезпеченням для управління конфігураціями ці питання залишаються і стають ще більш важливими – умовний керуючий сервер може працювати відразу з декількома різними середовищами, що можуть належати не тільки до різних хмарних акаунтів, але й до різних провайдерів. Тому, з'являються додаткові задачі по підтримці в працездатному стані таких інтеграцій: необхідно перевірити чи встановлені всі потрібні CLI, чи немає дня них оновлень, що може бути критично важливим в ряді безпекових питань, чи для всіх хмарних провайдерів було надано потрібний доступ через API-ключі, токени або інші засоби, чи потрібно оновити ці дані та багато інших задач.

Застосування API або SDK частково спрощує питання підтримки для кінцевого користувача програмного забезпечення, адже це дозволить інтегрувати функціонал по роботі з хмарними провайдерами напряму в застосунок і не буде необхідності вручну або автоматизовано встановлювати та налаштовувати CLI. Крім того, такий варіант значно розширює список варіантів використання – програмне створення віртуальних машин з різними варіантами конфігурації або інших хмарних ресурсів є більш гнучким процесом саме з застосуванням API/SDK та можливостей певної мови програмування, аніж при використанні CLI або GUI – такі варіанти значно гірше масштабуються.

Ще один варіант інтеграції – це застосування сторонніх IaC-інструментів, що спеціалізуються на хмарних провайдерах. До цієї категорії можна віднести як повноцінні програмні рішення по типу HashiCorp Terraform, так і відповідні хмарні сервіси. Terraform – це потужний інструмент, який автоматизує процес створення, зміни та управління інфраструктурою, що підтримує велику кількість хмарних провайдерів та інших інфраструктурних сервісів й інструментів. З його

застосуванням можливо не лише створювати окремі ресурси в AWS, Google Cloud або ж Azure, а навіть управляти ресурсами в Kubernetes-кластері, репозиторіями в GitHub або іншому Git-репозиторії тощо. Основна перевага Terraform – це можливість за допомогою одного інструменту та єдиної мови опису конфігурацій (HCL, HashiCorp Configuration Language – спеціально розроблена для Terraform та інших інструментів HashiCorp) керувати майже всією цифровою інфраструктурою бізнесу [28]. З іншого боку, інтеграція повноцінного комплексного IaC-додатку в програмне забезпечення для управління конфігураціями є складною задачею з рядом відкритих питань, як-от ліцензування такого продукту або ж потреба в додаткових навичках використання Terraform у кінцевого користувача ПЗ.

Альтернативою є застосування IaC-сервісів від самих хмарних провайдерів. Великі компанії давно зрозуміли переваги підходу Infrastructure as Code та представили свої власні сервіси для зручного створення і керування ресурсами, як-от AWS CloudFormation, Google Cloud Deployment Manager або ж Azure Resource Manager. На відміну від універсального Terraform ці інструменти спеціально будувалися під конкретного провайдера – вони можуть мати менше функціоналу, але є більш зручними в використанні й застосовують простіший синтаксис на основі JSON або YAML для опису шаблонів. Інтеграцію цих сервісів з ПЗ для управління конфігураціями можна виконати за допомогою відповідних API або SDK.

Таким чином, в даному підрозділі було розглянуто переваги та недоліки різних способів інтеграції з хмарними провайдерами, як-от застосування CLI, розробка інтеграції через GUI, використання IaC-інструментів та сервісів, різноманітних API і SDK. Застосування хмарних IaC-сервісів та відповідних їм API/SDK має найкраще співвідношення переваг до недоліків і є найбільш зручним для кінцевого користувача, тому на їх основі можна розробити вдосконалений метод для автоматичного створення серверів з використанням хмарних провайдерів для програмного забезпечення для управління конфігураціями.

1.6 Постановка задач розробки

На основі аналізу поточного стану питання, порівняння існуючих програмних продуктів за рядом визначених критеріїв і з урахуванням існуючих методів і підходів було визначено наступні завдання, які необхідно виконати, для розробки програмного забезпечення для управління конфігураціями:

1. Розробити метод і алгоритм для забезпечення високої доступності ПЗ для управління конфігураціями.
2. Розробити метод і алгоритм для автоматичної інсталяції програм-агентів на керовані сервери.
3. Розробити метод і алгоритм для автоматичного створення серверів з використанням хмарних провайдерів.
4. Розробити графічний інтерфейс користувача для створюваного ПЗ.
5. Розробити програмний застосунок для управління конфігураціями.
6. Провести тестування програмного застосунку.
7. Дослідити ефективність методу забезпечення високої доступності.
8. Розробити інструкцію користувача.

Технічне завдання наведено в додатку А.

1.7 Висновки

У першому розділі було розглянуто сучасний стан питання IaC-інструментів та програмного забезпечення для управління конфігураціями, наведено причини та приклади їх застосування. Було досліджено предметну область, проведено аналіз існуючих аналогів на основі ряду критеріїв, наведено їх переваги й недоліки. Аналіз поточного стану питання та існуючих аналогів довів доцільність розробки власного програмного продукту для управління конфігураціями, що зможе надати кінцевому користувачу новий функціонал та виправити виявлені недоліки аналогів. Було проаналізовано існуючі методи для реалізації режиму високої доступності, автоматичної інсталяції програм-агентів та інтеграції з хмарними провайдерами. На основі отриманих даних проведено постановку задач розробки.

2 РОЗРОБКА МЕТОДІВ ТА АЛГОРИТМІВ ПРОГРАМНОГО ПРОДУКТУ

2.1 Розробка методу забезпечення високої доступності ПЗ для управління конфігураціями

В ході дослідження було проаналізовано різноманітні методи для забезпечення високої доступності програмних систем та прийнято рішення розробити власний метод, що буде підходити для програмного забезпечення для управління конфігураціями.

Основна проблема при керуванні конфігураціями, яку може вирішити режим високої доступності – це втрата зв’язку між керуючим та керованими серверами в процесі приведення системи в бажаний стан. Це може бути викликано мережевими проблемами або непрацездатністю апаратного чи програмного забезпечення керуючого сервера. Кількість керованих серверів варіюється від 2-3 до декількох десятків або сотень в залежності від конкретного варіанту використання, тому таку групу серверів можна сприймати в якості кластеру, де кожен вузол виконує свою власну функцію, але відсутні надлишкові вузли й дані для управління кластером (бажаний стан системи) знаходяться на окремій машині – керуючому сервері.

Для реалізації повноцінного кластерного підходу в першу чергу необхідно змінити модель взаємодії між серверами. Типове ПЗ для управління конфігураціями застосовує централізований режим – всі керовані вузли отримують інструкції від одного керуючого сервера, виконують ці інструкції та надсилають результати назад для завершення роботи або подальшої обробки.

При розробці ПЗ з підтримкою кластеризації найважливішим питанням є зберігання та обмін даними. Колишні керовані вузли повинні знати які інструкції необхідно виконати для досягнення бажаного стану системи в процесі управління конфігураціями. Кластерний підхід накладає мало обмежень на систему, тому можна застосувати гібридний режим роботи – модифікувати архітектуру залишаючи центральний керуючий сервер.

Простим рішенням може бути застосування окремого зовнішнього сховища даних, з яким будуть спілкуватися усі вузли (наприклад, СКБД на окремому сервері, який не є частиною кластера та може бути додатково захищений на мережевому рівні). На відміну від повністю централізованого підходу буде відсутній єдиний контролер, що роздає інструкції – його роль частково бере на себе кожен робочий вузол. Для такого сховища даних можна незалежно налаштувати власний режим високої доступності тими методами, що найкраще підходять тій чи іншій СКБД. Це дозволить вирішити проблему єдиної точки відмови.

Іншим варіантом є застосування принципу надлишковості для керуючого сервера. При такій архітектурі центральний контролер залишається частиною кластера, але він дублюється одним або декількома додатковими серверами, на які пасивно реплікуються усі необхідні дані, щоб в будь-який момент часу з мінімальними затримками можна було переключитись з одного керуючого вузла на інший [29].

Обидва рішення добре підходять для гібридних систем, які поступово відмовляються у своїй архітектурі від централізації та додають підтримку режиму високої доступності, але вони не є оптимальними для ПЗ для управління конфігураціями через потребу застосування окремого виділеного сервера для зберігання даних і контролю над процесом роботи. Необхідно застосувати децентралізований (або розподілений) режим взаємодії – керуючий сервер більше не застосовується, натомість всі вузли в системі знають одне про одного та можуть в більш вільному режимі обмінюватись повідомленнями в залежності від присвоєної ролі.

В розподіленій програмній системі з’являється нова проблема – узгодження даних. Сервери можуть не тільки повністю виходити з ладу через апаратну або програмну помилку, а й просто частково ставати недоступними в мережі з ненадійним з’єднанням. В такому випадку окремий вузол може продовжити ізольовану роботу й накопичити дані про стан системи, що відрізняється від інших вузлів. При відновленні мережевого з’єднання з іншими

серверами виникає конфлікт – існує декілька різних варіантів поточного стану кластеру, тому виконання подальших інструкцій може призвести до неочікуваних результатів.

Довгий час для вирішення цієї проблеми широко застосовували Paxos – сімейство протоколів та алгоритмів консенсусу, що розробили в 1990-х. Основна мета Paxos та подібних до нього алгоритмів – це досягнення консенсусу, тобто вирішення цього конфлікту суперечливих даних і вибір «коректної» версії, з якою кластер може продовжити далі працювати. Не дивлячись на популярність, сімейство протоколів Paxos є складним у розумінні та реалізації, тому для розробки розподілених або децентралізованих систем почали з'являтися нові алгоритми консенсусу.

Для нових програмних продуктів дедалі частіше застосовують Raft – алгоритм консенсусу, що вперше був опублікований в 2013 році. Raft робить фокус на простоті розуміння алгоритму, але без втрати ефективності роботи та відмовостійкості, що притаманні Paxos. Специфікації Raft описують лише загальний алгоритм та не надають конкретної реалізації, тому його часто беруть за основу для розробки власних розподілених систем. Різноманітні імплементації алгоритму застосовуються в таких програмних продуктах як MongoDB, Etcd, Neo4j тощо.

Розглянемо більш детально принципи роботи Raft та переваги його застосування в процесі розробки власного методу забезпечення високої доступності. В основі алгоритму консенсусу лежить скінченний автомат з трьома станами-ролями для вузлів кластера: лідер, послідовник та кандидат [6]. Для визначення актуальності даних (стану системи) застосовується монотонно зростаючий показник – термін або артикул (англ. term). Під час обміну даними між вузлами значення терміну оновлюється з певною періодичністю – якщо в цей час поточний лідер кластеру не може обробити запит через непрацездатність, то починається процес голосування та зміни ролей.

Таким чином, Raft описує простий в сприйнятті та реалізації алгоритм, що дозволяє створити розподілену програмну систему, але для розробки ПЗ для

управління конфігураціями його необхідно розширити та модифікувати у власний метод з метою вирішення ряду проблем та адаптації до вимог власної програмної системи. По-перше, для збереження консенсусу та коректного оновлення стану системи клієнтам дозволяється працювали лише з поточним лідером кластеру. Це створює додаткове навантаження на один з вузлів, а також залишає відкритим питання розподілення задач між серверами – при управлінні конфігураціями кожен вузол має власні інструкції для приведення системи в бажаний стан.

Проблему зв’язку між поточним лідером кластеру та клієнтом неможливо вирішити за допомогою звичайного балансування навантаження, адже балансувальник повинен не просто перенаправити запит до працездатного вузла, але ще й визначити чи цей сервер є поточним лідером. Більш комплексні програмні балансувальники можна налаштувати для такої інтеграції з ПЗ для управління конфігураціями, але це може обмежити деякі варіанти використання через специфіку налаштування та більших вимог до самої інфраструктури.

Інший варіант, який обрано для власного методу забезпечення високої доступності, полягає в застосуванні додаткового механізму обробки запитів. Вузли системи можуть вільно обмінюватися інформацію між собою, а також завжди знають який з серверів є поточним лідером. При отриманні нового запиту вузол-послідовник може перевірити та отримати адресу лідера та виступити в ролі тимчасового проксі-сервера для цього запиту – тобто перенаправити повідомлення до лідера для подальшої обробки. В залежності від застосованого протоколу комунікації механізм перенаправлення може бути реалізований навіть вбудованими засобами (наприклад, API на основі HTTP/REST має такі можливості).

Питання маршрутизації запитів клієнтів до поточного лідера кластера вирішене, але залишаються проблеми підвищеного навантаження та розподілу задач між вузлами. В оригінальному Raft основною задачею лідера є не просто обробка вхідних повідомлень, а централізоване оновлення даних на всіх вузлах системи – саме це й дозволяє досягти консенсусу. Базуючись на цьому,

програмне забезпечення для управління конфігураціями може мати декілька окремих сервісів для комунікації з користувачем – окремий набір API може бути доступний на всіх вузлах без залежності до їх ролі та виконувати загальні операції для зчитування даних, а інший сервіс буде працювати лише на сервері-лідері та приймати важливі запити по оновленню даних системи. Відповідно, вузли-послідовники будуть перенаправляти до лідера саме ці повідомлення для оновлення.

Для розподілу задач можна застосувати схожий принцип – лідер кластеру реплікує на всі сервери новий стан, але самостійно не розподіляє задачі. Замість цього на кожному вузлі-послідовнику буде працювати окремий сервіс для аналізу стану при його оновленні. Основна задача такого сервісу – це зчитати нові відомості для поточного сервера з загального масиву даних, порівняти поточний стан вузла з новим та в разі потреби запустити процес приведення системи в бажаний стан на окремо взятому сервері, що й є основною задачею ПЗ для управління конфігураціями.

Таким чином, в підрозділі розроблено вдосконалений метод забезпечення високої доступності ПЗ для управління конфігураціями за допомогою розширення, адаптації та поєднання існуючих підходів. Метод бере за основу алгоритм консенсусу Raft та підхід кластеризації і розширює їх додатковими рішеннями для забезпечення потрібного функціоналу.

2.2 Розробка методу автоматичної інсталяції програм-агентів на керовані сервери

При попередньому аналізі було обрано протоколи SSH та WinRM в якості таких, що можуть бути використані при розробці власного методу автоматичної інсталяції програм-агентів на керовані сервери. Розглянемо більш детально особливості роботи такого методу.

По-перше, для застосування подібного методу автоматичної інсталяції є ряд передумов, які необхідно виконати. Безпекові налаштування більшості операційних систем за замовчуванням не дозволяють застосування протоколів

віддаленого доступу або ж мають занадто сувору й обмежену конфігурацію, що перешкоджає запуску будь-яких автоматичних скриптів. Для початку роботи з подібними серверами кінцевому користувачу необхідно самостійно змінити відповідні налаштування у ручному режимі – автоматично без дій людини це зробити неможливо, тому подібну конфігурацію можна віднести до категорії системних вимог для застосування створюваного ПЗ для управління конфігураціями.

Також може бути необхідна додаткова мережева конфігурація – налаштування за замовчуванням можуть блокувати необхідні порти та протоколи на рівні мережевого екрану ОС. У випадку локальних (англ. on-premises) серверів така конфігурація також повинна виконуватись вручну. При використанні ресурсів хмарних провайдерів частину дій можливо автоматизувати. Наприклад, в AWS застосовуються охоронні групи (англ. security groups) в якості зовнішнього мережевого екрану на рівні віртуальної машини (у випадку AWS – це EC2 Instance) [30]. За допомогою відповідного API можна налаштувати їх без втручання в саму операційну систему сервера. Більш детально особливості такої конфігурації розглянуто в методі автоматичного створення серверів з використанням хмарних провайдерів.

Після виконання усіх необхідних передумов основні дії розроблюваного методу повинні виконуватися автоматично та під контролем ПЗ для управління конфігураціями. Для початку роботи програмний продукт повинен отримати вхідні дані в вигляді списку нових серверів, які необхідно додати в кластер.

З кожним новим сервером встановлюється з’єднання за допомогою відповідного протоколу віддаленого доступу: SSH для Linux та WinRM для Windows. Також, за допомогою OpenSSH можна застосувати зв’язок по SSH і для Windows-серверів, що може потенційно покращити стабільність зв’язку. За допомогою протоколу віддаленого доступу необхідно виконати дві дії: скопіювати на сервер необхідні програмні компоненти та провести їх первинне налаштування, яке необхідне для запуску ПЗ для управління конфігураціями. Відповідно, всі вже працюючі вузли системи повинні реплікувати між собою не

тільки стан системи, а й ці додаткові файли для запуску нових агентів. Їх зберігання лише на одному з серверів або в іншому централізованому сховищі створює ще одну єдину точку відмови.

З урахуванням перспективи роботи в мережі з нестабільним з'єднанням власний метод повинен виконувати додаткові перевірки цілісності файлів на кроці копіювання. Первинне налаштування програмних компонент на новому сервері включає генерацію ідентифікуючих даних для вузла: наприклад, ID, GUID або унікальний текстовий рядок, що може бути більш зручним для кінцевого користувача. Новий сервер також повинен отримати інформацію про поточного лідера кластеру та інших його учасників.

Якщо певні вузли системи в цей момент будуть недоступними, то це не перешкоджає процедурі розгортання нового агента – він може отримати відомості про них від інших серверів кластера або ж в момент їх повернення до працездатного стану. Крім того, при первинному налаштуванні може відбуватись обмін додатковими даними для встановлення захищеного з'єднання між серверами, як-от TLS-сертифікатами або іншими ключами доступу.

Після процедури налаштування усі програмні компоненти ПЗ на новому сервері готові до використання. Вузол можна додавати до системи відповідно до процедури, що визначає метод для забезпечення високої доступності. Після цього можна закрити з'єднання по SSH або WinRM та переключитись на комунікацію всередині кластеру.

Таким чином, в підрозділі розроблено вдосконалений метод для автоматичної інсталяції програм-агентів на керовані сервери. Метод бере за основу протоколи віддаленого доступу та передбачає виконання ряду процедур для запуску нового вузла системи.

2.3 Розробка методу автоматичного створення серверів з використанням хмарних провайдерів

На основі попереднього аналізу було прийнято рішення розробити власний метод для автоматичного розгортання серверів за допомогою IaC-інструментів

або відповідних API або SDK хмарних провайдерів. Загальний процес роботи методу не буде відрізняти для різних платформ хмарних обчислень – інструменти для роботи з віртуальними або фізичними серверами та відповідні мережеві сервіси зазвичай мають подібні одне до одного функціонал та можливості.

Аналогічно до попереднього методу є підготовчі кроки, які необхідно виконати один раз в ручному режимі. До цього відносяться налаштування безпеки та надання ПЗ для управління конфігураціями доступу до хмарних ресурсів. При розробці методу враховано можливості трьох найбільших хмарних провайдерів: AWS, GCP та Azure.

Для керування ідентифікацією та доступом в AWS та GCP застосовуються сервіси під назвою IAM, а в Azure – Active Directory. В першу чергу, ПЗ для управління конфігураціями повинно отримати через ці сервіси доступ до API. Сервіси мають схожий функціонал та виділяють декілька різних методів для автентифікації та авторизації.

Перший з них – це облікові записи користувачів. Вони призначені для використання реальними людьми, для автентифікації використовують логін та пароль або ж інший набір конфіденційних даних, а також за допомогою облікового запису можна або напряму отримати доступ до API, або згенерувати відповідні постійні чи тимчасові ключі доступу [31].

Наступний метод доступу – це застосування сервісних або службових записів, назва може відрізнятися в залежності від хмарного провайдера. Наприклад, в AWS це IAM-роль (англ. IAM Role), в GCP – сервісний акаунт (англ. Service Account), а в Azure – керована ідентифікація або особа (англ. Managed Identity). Такий сервісний акаунт не має логіну та паролю і використовує спеціальні методи автентифікації для отримання доступу, тому може працювати лише всередині віртуальної мережі хмарного провайдера. Відповідно, його основне призначення – це програмний доступ до ресурсів, а типовий приклад використання – надання можливості одному ресурсу проводити певні дії над іншим всередині хмари.

Програмне забезпечення для управління конфігураціями може застосовувати обидва способи автентифікації та авторизації, основним обмеженням є лише середовище запуску початкових вузлів кластеру. Наприклад, якщо кластер створено на окремих локальних (on-premises) серверах, а потім було прийнято рішення розширити його за допомогою нових машин з хмари і отримати гібридне середовище, то необхідно застосовувати ключі доступу від облікового запису реального користувача, адже відбувається доступ до хмарних ресурсів «ззовні».

Відповідно до цього, якщо початковий кластер повністю розгорнуто на хмарних ресурсах, то для його подальшого розширення буде достатньо доступу від сервісних акаунтів. Для такого варіанту застосування це переважний варіант з точки зору безпеки, адже авторизацію та генерацію ключів доступу виконують внутрішні автоматичні механізми хмарного провайдера, тому відсутні ризики витоку даних та людської помилки при цих операціях. Запропонований метод автоматичного створення серверів повинен враховувати можливість обробки обох сценаріїв.

Для створення нових серверів необхідно мати ряд вхідних даних, включаючи як параметри самої віртуальної чи фізичної машини, так і налаштування мережевого оточення. Якщо взяти за приклад AWS, то найважливішим елементом там є віртуальна приватна хмара (англ. VPC, Virtual Private Cloud), яка надає користувачу віртуальну ізольовану мережу. Наступним рівнем виступають підмережі VPC (англ. VPC Subnets), які розбивають велику мережу на окремі логічні групи зі своїми власними налаштуваннями [32]. Вже всередині підмережі можна працювати з серверами. Крім того, необхідно створити й застосувати до серверів безпекові групи (англ. SG, Security Groups) для ізоляції серверів всередині VPC одне від одного та унеможливлення небажаного публічного доступу до них.

Запропонований метод повинен працювати з двома основними варіантами використання: користувач може або надати дані вже створених VPC, Subnets та SG, або ж ввести потрібні налаштування та дати можливість ПЗ для управління

конфігураціями підготувати окремі ресурси для роботи кластеру. Другий варіант може бути кращим при деяких обставинах, адже ізолює програмну систему на рівні VPC від інших додатків та систем, що можуть одночасно працювати на одному акаунті хмарного провайдера.

Після підготовки мережевої інфраструктури створення самих серверів можна виконати за допомогою відповідного API хмарного провайдера в повністю автоматичному режимі. Підтримка автоматичного масштабування (наприклад, за допомогою AWS Auto Scaling) в рамках цього методу не розглядається, адже критерії для зміни розміру кластеру сильно залежать від сценарію використання вузлів системи.

Додатковим кроком після створення серверів виступає їх валідація – необхідно перевірити можливість зв’язку між уже існуючими вузлами кластеру та новими машинами в хмарі. Якщо зв’язок частково або повністю відсутній – користувач помилився в налаштуваннях або ж існує інша проблема з мережевою інфраструктурою. На прикладі AWS це може бути неправильна конфігурація SG або NACL. В такому випадку за бажанням користувача метод повинен автоматично видалити нові ресурси, щоб запобігти небажаних витрат до моменту виправлення помилок. При успішній валідації виконання методу можна вважати успішним, ПЗ може приступати до автоматичної інсталяції агентів на нових серверах.

Таким чином, в підрозділі було розроблено вдосконалений метод для автоматичного створення серверів. Метод бере за основу існуючі API та SDK хмарних провайдерів та застосовує власну логіку підготовки та перевірки мережевої інфраструктури й інших ресурсів задля розширення існуючого кластеру ПЗ для управління конфігураціями.

2.4 Розробка блок-схем методів та алгоритмів

На основі методів, що були розроблені в підрозділах 2.1-2.3, було створено відповідні алгоритми та блок-схеми. Алгоритм забезпечення високої доступності описує загальний потік виконання вузла кластера, а саме його запуск та обробку

вхідних команд від клієнта з залежністю від поточної ролі сервера. Блок-схема алгоритму наведена на рисунку 2.1.

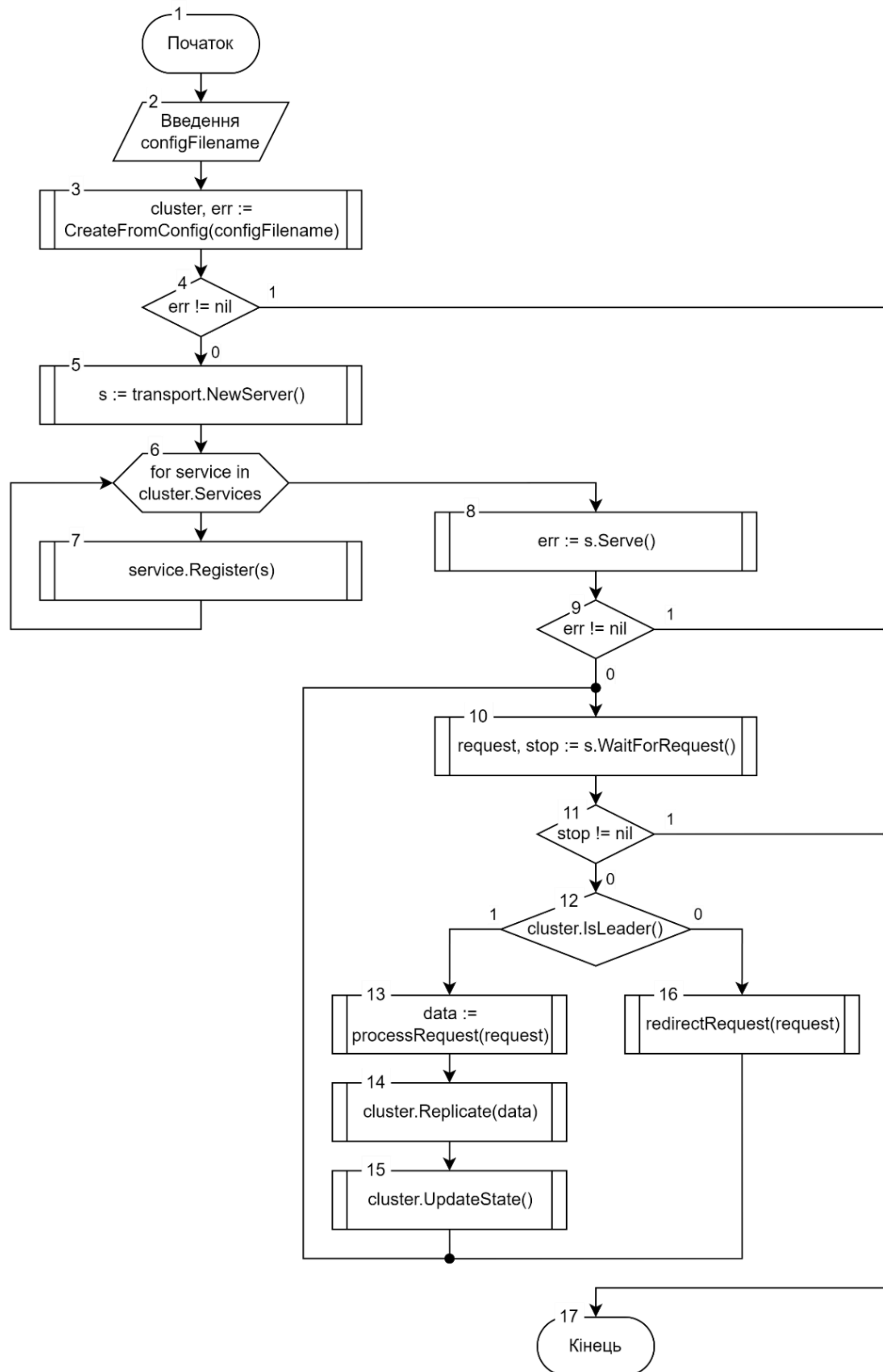


Рисунок 2.1 – Блок-схема алгоритму забезпечення високої доступності

Розглянемо більш детально кроки алгоритму.

Крок 1. Початок.

Крок 2. Користувач вводить шлях до файлу конфігурації вузла, в деяких випадках може використовуватися шлях за замовчуванням (наприклад, локальний файл в директорії додатку); значення зберігається в змінній configFilename.

Крок 3. Відбувається ініціація вузла кластера: функція CreateFromConfig за переданим шляхом зчитує конфігурація, отримані значення застосовуються для підготовки локального сховища даних та інших процедур.

Крок 4. Якщо на попередньому кроці при ініціації вузла виникла помилка – відбувається перехід до кроку 17. Інакше – перехід до кроку 5.

Крок 5. Відбувається підготовка програмного сервера, який необхідний як для комунікації між вузлами системи, так і для роботи окремих сервісів вузла.

Крок 6. Розпочинається цикл, який перебирає всі сервіси, що необхідні для роботи кластера. Виконання тіла циклу розпочинається на кроці 7, завершення циклу викликає перехід до кроку 8.

Крок 7. Сервіс кластеру реєструється на локальному сервері та ініціює внутрішній стан.

Крок 8. Відбувається запуск сервера та зареєстрованих на ньому сервісів вузла кластера.

Крок 9. Якщо на попередньому кроці при запуску сервера виникла помилка – відбувається перехід до кроку 17. Інакше – перехід до кроку 10.

Крок 10. Сервер очікує нових вхідних повідомлень від клієнта або іншого вузла. Виконання продовжується коли отримано новий запит request або ж керуючий сигнал для зупинки stop.

Крок 11. Якщо отримано керуючий сигнал stop, то відбувається перехід до кроку 17. Інакше – перехід до кроку 12.

Крок 12. Якщо поточний вузол являється лідером кластера, то відбувається перехід до кроку 13. Інакше – перехід до кроку 16.

Крок 13. Відбувається обробка запиту та отримання даних для реплікації.

Крок 14. Поточний лідер кластера реплікує нові дані стану на інші сервери.

Крок 15. Відбувається процедура порівняння попереднього стану та оновленого; за потребою виконуються додаткові дії для досягнення бажаного стану системи, тобто запускається основний функціонал ПЗ для управління конфігураціями.

Крок 16. Вхідний запит перенаправляється з вузла-послідовника до вузла-лідера.

Крок 17. Кінець виконання.

Алгоритм автоматичної інсталяції програм-агентів на керовані сервери описує загальний потік виконання при підготовці до експлуатації нового вузла. Блок-схема алгоритму наведена на рисунку 2.2.

Розглянемо кроки даного алгоритму.

Крок 1. Початок.

Крок 2. Користувач вводить структуру serverData, що містить необхідні дані для підключення до сервера (наприклад, IP-адресу, мережеві порти та протокол підключення).

Крок 3. Вузол кластера намагається встановити віддалене з’єднання з сервером на основі наданих даних.

Крок 4. Якщо при підключенні виникли помилки, то відбувається перехід до кроку 13. Інакше – перехід до кроку 5.

Крок 5. Вузол зберігає поточну конфігурацію кластера в змінну clusterInfo.

Крок 6. На основі поточних даних кластера clusterInfo та даних нового вузла serverData генерується налаштування для нового сервера.

Крок 7. Через віддалене з’єднання на новий вузол копіюються програмні компоненти ПЗ.

Крок 8. Через віддалене з’єднання копіюється конфігурація нового вузла системи.

Крок 9. Відбувається розгортання нового вузла, а саме запуск програмних компонент зі згенерованим конфігураційним файлом.

Крок 10. Відбувається перевірка працездатності вузла в кластері. Результати перевірки записуються в змінну results.

Крок 11. Якщо на попередньому кроці виявлено проблеми в роботі вузла (новий сервер не може функціонувати в складі кластера або виникли інші проблеми), а також користувач у вхідних даних вказав кількість повторних спроб для автоматичної інсталяції програм-агентів, то відбувається перехід до кроку 5. Інакше – перехід до кроку 12.

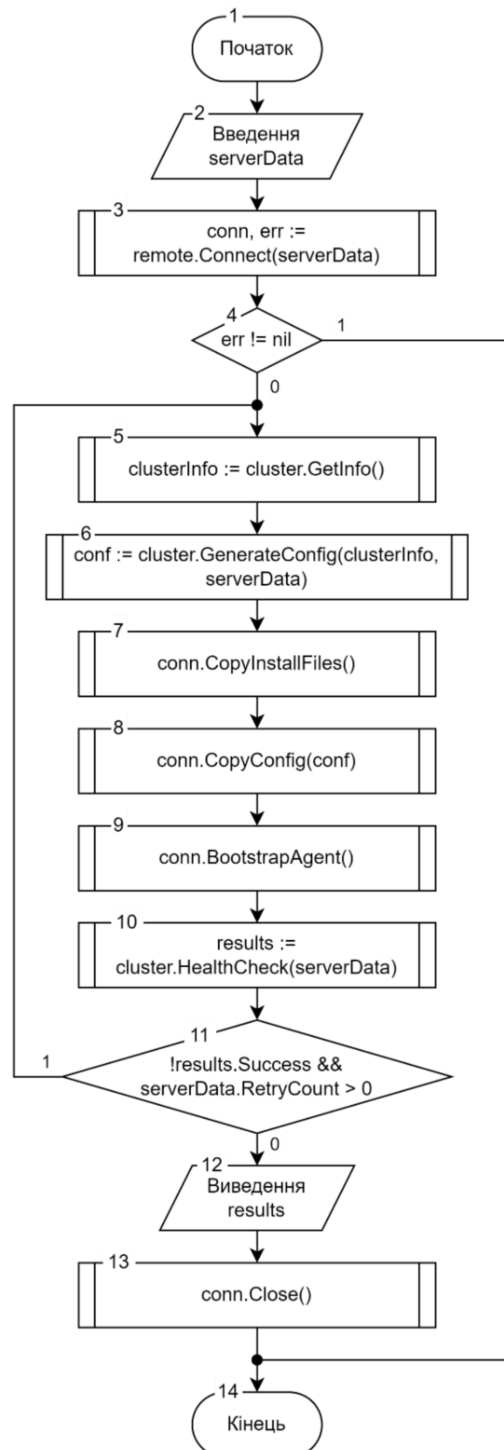


Рисунок 2.2 – Блок-схема алгоритму автоматичної інсталяції програм-агентів

Крок 12. Виведення даних про результати перевірки працездатності нового вузла системи.

Крок 13. Віддалене з’єднання з сервером закривається – подальша робота з вузлом відбувається повністю через комунікацію всередині кластера.

Крок 14. Кінець.

Алгоритм для автоматичного створення серверів з використанням хмарних провайдерів описує процес підготовки інфраструктури для ПЗ для управління конфігураціями. Блок-схема алгоритму наведена на рисунку 2.3.

Розглянемо більш детально кроки виконання алгоритму.

Крок 1. Початок.

Крок 2. Користувач вводить необхідні вхідні дані: конфігурацію серверів, конфігурацію мережі та параметри доступу до хмарного провайдера. Дані зберігаються відповідно у структури serversConfig, networkConfig та accessConfig.

Крок 3. На основі параметрів доступу створюється клієнт для роботи з хмарним провайдером.

Крок 4. Якщо на попередньому кроці сталася помилка (наприклад, авторизаційні дані некоректні або їх термін дії закінчився), то відбувається перехід до кроку 16. Інакше – перехід до кроку 5.

Крок 5. Відбувається перевірка інформації про мережеві ресурси: якщо в отриманій від користувача структурі зазначено, що мережева інфраструктура на стороні хмарного провайдера вже існує, то відбувається перехід до кроку 7. Інакше – перехід до кроку 6.

Крок 6. На основі вхідних даних створюються хмарні ресурси для мережевої інфраструктури.

Крок 7. Відбувається перевірка мережевих ресурсів, що зазначені в структурі вхідних даних.

Крок 8. Якщо при перевірці ресурсів не виявлено помилок та мережа на стороні хмарного провайдера готова до використання, то відбувається перехід до кроку 8. В іншому випадку (наприклад, користувач допустив помилку в вхідних

даних або ж не всі з зазначених ресурсів були попередньо створені) відбувається перехід до кроку 16.

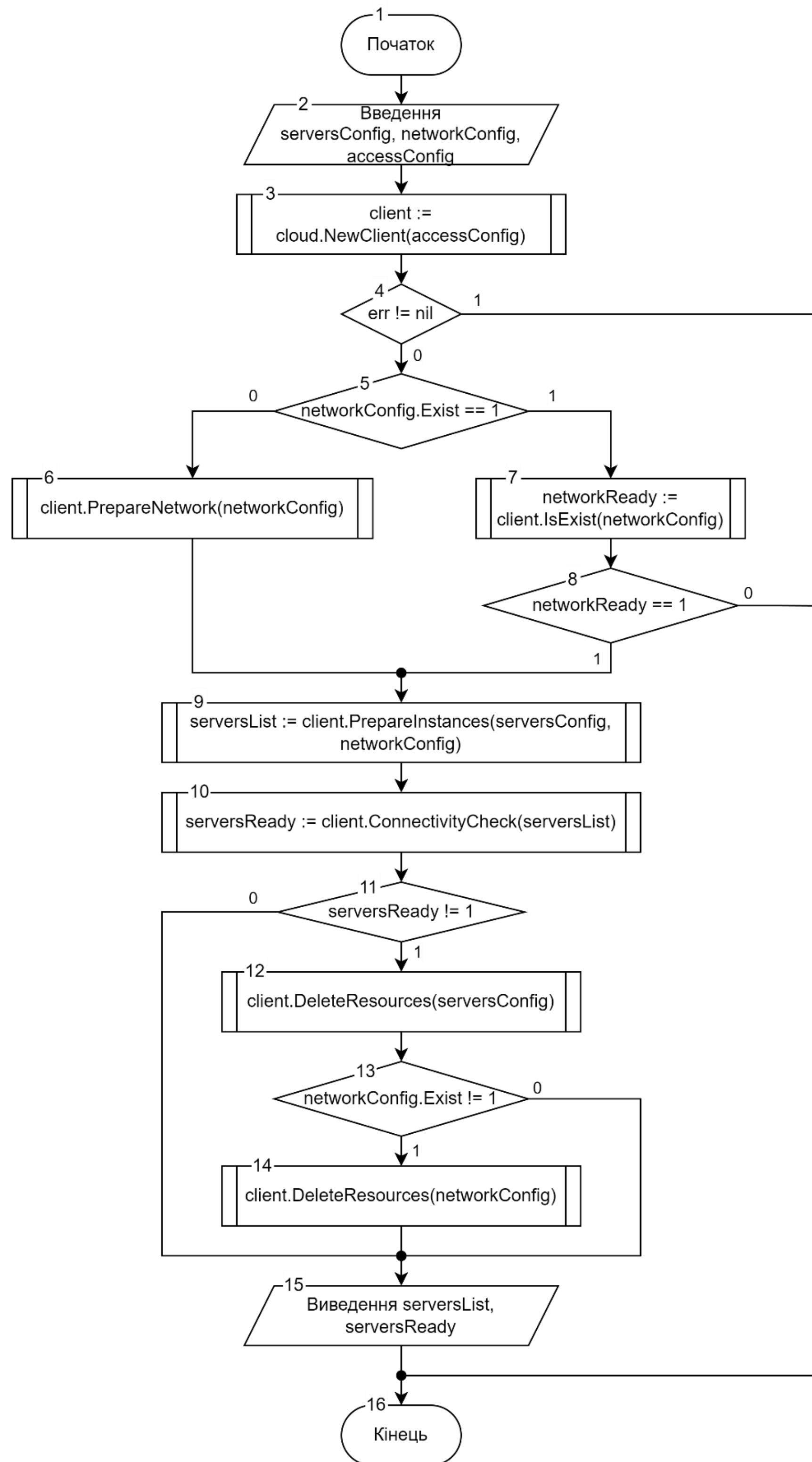


Рисунок 2.3 – Блок-схема алгоритму автоматичного створення серверів з використанням хмарних провайдерів

Крок 9. На основі конфігурації мережі та серверів за допомогою клієнта хмарного провайдера розгортаються нові вузли.

Крок 10. Відбувається базова перевірка на мережеву доступність нових серверів. Цей крок необхідний для виявлення потенційних проблем з конфігурацією самих вузлів або ж мережевих ресурсів, які можуть зашкодити подальшому використанню серверів в складі кластера.

Крок 11. Якщо на попередньому кроці виявлено проблеми з серверами, то відбувається перехід до кроку 12. Інакше – перехід до кроку 15.

Крок 12. Відбувається видалення створених серверних ресурсів.

Крок 13. Якщо під час виконання алгоритм самостійно створював мережеву інфраструктуру, то відбувається перехід до кроку 14. Інакше – перехід до кроку 15.

Крок 14. Відбувається видалення створених ресурсів мережевої інфраструктури.

Крок 15. Виведення результатів роботи алгоритму, а саме списку серверів та результат їх перевірки. Якщо перевірки були успішні, то сервери готові до подальшої установки програм-агентів та функціонування в складі кластера. Якщо ж ні – наведені ресурси вже були видалені до кінця виконання з метою економії коштів на хмарні ресурси, а для користувача ці дані надаються лише для інформування.

Крок 16. Кінець.

Таким чином, в даному підрозділі було розглянуто принципи роботи основних алгоритмів створюваного програмного забезпечення для управління конфігураціями, розроблено та детально описано відповідні блок-схеми.

2.5 Розробка структури графічного інтерфейсу користувача

Графічний інтерфейс користувача (англ. Graphical User Interface, GUI) – це тип інтерфейсу, що призначений для взаємодії користувача з програмним забезпеченням на електронному пристрої за допомогою візуальних елементів, як-от кнопки, вікна, меню тощо. GUI призначений для спрощення сприйняття та

взаємодії з програмним продуктом шляхом відмови від запам’ятовування і введення комплексних команд та інструкцій в командний рядок. Замість цього застосовується більш інтуїтивне та логічно організоване графічне представлення.

Інтерфейс користувача веб-застосунок (англ. WebUI, Web User Interface) – це один із різновидів GUI, що використовується для забезпечення взаємодії користувача з веб-застосунками та веб-сайтами. Основні принципи роботи WebUI аналогічні до звичайного GUI, але для його відображення застосовується веб-браузер [33]. Це дозволяє обійти типові проблеми при перенесенні ПЗ на інші платформи та застосовувати один стек технологій для розробки інтерфейсу. Результуючий застосунок буде доступний з будь-якого пристрою, що має встановлений браузер та доступ до внутрішньої або публічної мережі.

Такий підхід добре підходить для ПЗ для управління конфігураціями, адже користувачу не треба встановлювати жоден клієнтський застосунок, йому необхідно лише знати адресу хоча б одного з вузлів системи. На основі цього було розроблено схему головного вікна програми, що наведена на рисунку 2.4.

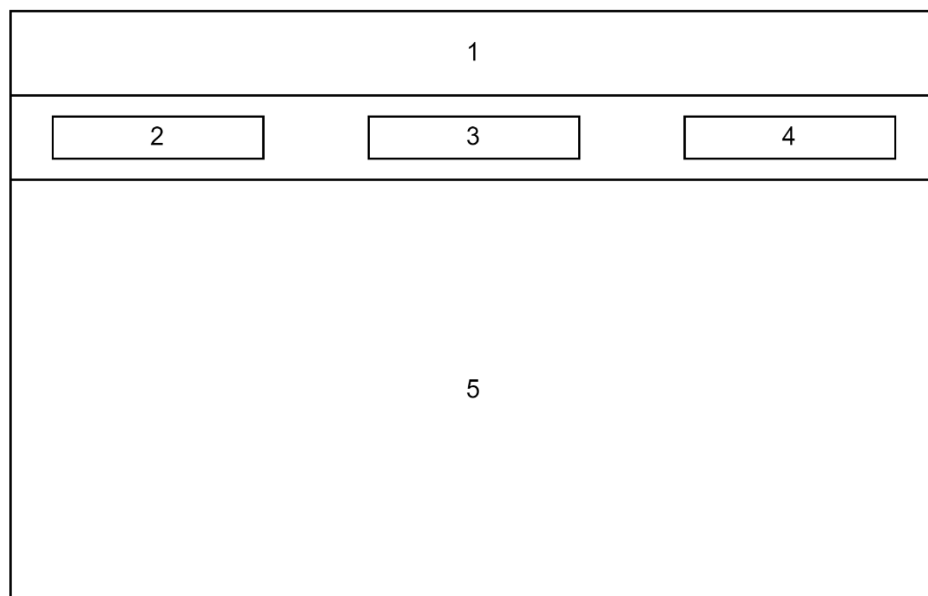


Рисунок 2.4 – Структурна схема головного вікна застосунку

Основні елементи інтерфейсу головного вікна:

1) керуючі елементи вікна браузера (адресний рядок, системні кнопки керування вікном, панель інструментів тощо; набір залежить від браузера користувача);

2) кнопка переключення на вкладку «Редактор кластера»;

3) кнопка переключення на вкладку «Редактор стану»;

4) кнопка переключення на вкладку «Переглядач логів»;

5) вміст поточної вкладки.

Кнопки головного вікна (2-4) відповідають лише за переключення між різними вкладками та змінюють вміст центральної області (5), де і розміщуються основні елементи інтерфейсу для використання функціоналу ПЗ.

Розглянемо більш детально схему інтерфейсу вкладки «Редактор кластера», що зображена на рисунку 2.5.

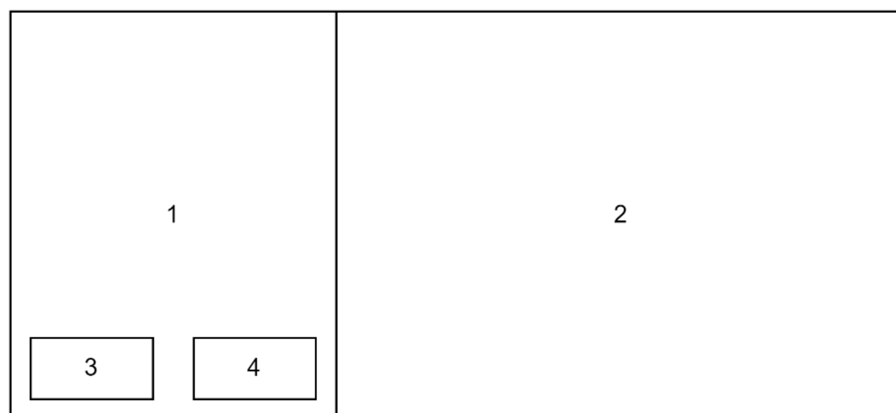


Рисунок 2.5 – Структурна схема вкладки «Редактор кластера»

Вкладка «Редактор кластера» містить наступні елементи інтерфейсу:

1) меню зі списком вузлів-членів кластера;

2) область з додатковою інформацією по обраному серверу;

3) кнопка редагування даних обраного сервера;

4) кнопка додавання нового вузла в кластер.

Меню (1) дозволяє переглянути список поточних вузлів кластера та обрати будь-який з них для перегляду додаткової інформації або ж перейти в режим редагування за допомогою кнопки (3). Крім того, користувач може розширити

кластер натисканням відповідної кнопки (4). Основна робоча область (2) може змінювати своє наповнення в залежності від поточного режиму роботи. При перегляді інформації це може бути просто набір полів без можливості зміни даних, а в режимі редагування або додавання сервера – може виконувати функцію повноцінної форми-редактора.

Структурна схема вкладки «Редактор стану» наведена на рисунку 2.6.

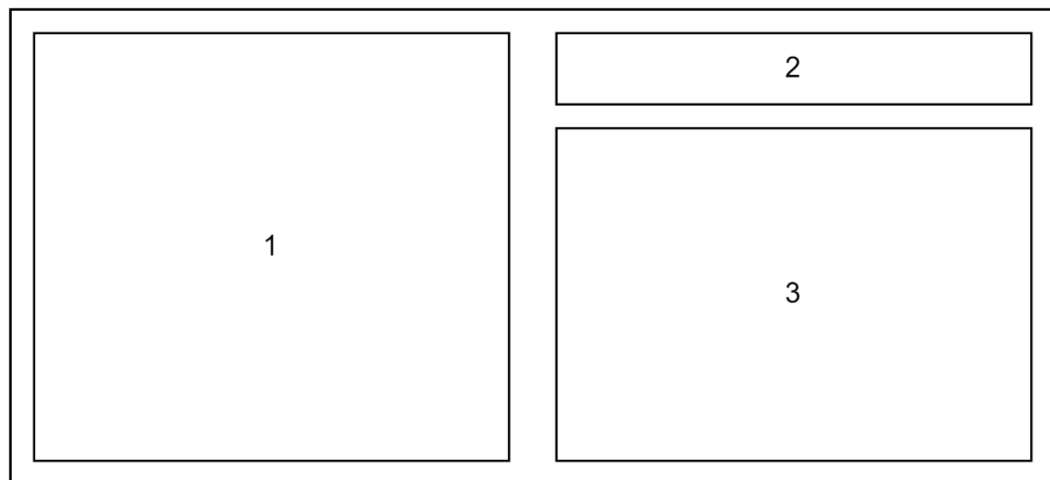


Рисунок 2.6 – Структурна схема вкладки «Редактор стану»

Вкладка «Редактор стану» включає в себе наступні елементи інтерфейсу:

1) область з інформацією про загальний стан кластеру;

2) кнопка редагування стану системи;

3) меню зі списком окремих серверів та їх індивідуальним станом.

Загальний бажаний стан системи копіюється і розповсюджується серед усіх вузлів системи, але кожен окремий сервер може мати свої індивідуальні характеристики та знаходитись на певному етапі приведення власного стану до загального бажаного. Саме по цій причині вкладка включає окрему область для перегляду загального стану (1) та меню з додатковими даними для кожного сервера (3). Кнопка (2) дозволяє ввійти в режим редагування стану для введення нових команд по управлінню конфігураціями серверів або ж змінити вхідні дані для цих команд.

Схема вкладки «Переглядач логів» наведена на рисунку 2.7.

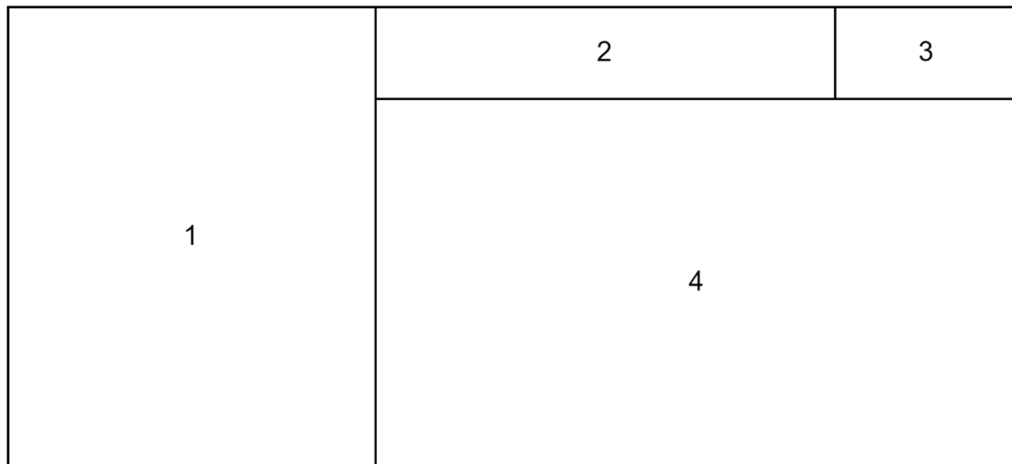


Рисунок 2.7 – Структурна схема вкладки «Переглядач логів»

Елементи інтерфейсу вкладки:

1) меню зі списком вузлів системи;

2) область з інформацією про обраний сервер;

3) текст-індикатор стану обраного вузла;

4) логи виконання ПЗ на обраному сервері.

Вкладка «Переглядач логів» призначена для більш детальної перевірки поточного стану ПЗ для управління конфігураціями на кожному з серверів. За допомогою меню (1) користувач може вибрати будь-який з вузлів системи, а відповідна область (2) та індикатор (3) коротко покажуть його статус. Область з логами (4) дозволяє перевірити як різноманітні сервісні події ПЗ, так і події, що пов’язані безпосередньо з управлінням конфігураціями.

Таким чином, в підрозділі було розглянуто структуру графічного інтерфейсу користувача для створюваного програмного продукту. Наведено базові відомості про GUI та WebUI, розроблено структурні схеми головного вікна додатку та його окремих вкладок, детально описано окремі елементи інтерфейсу ПЗ та їх призначення.

2.6 Висновки

У другому розділі було розроблено основні методи, що необхідні для створюваного ПЗ для управління конфігураціями, а саме:

– метод забезпечення високої доступності, що базується на алгоритмі консенсусу Raft та дозволяє програмній системі працювати в режимі кластеру з частково непрацездатними або недоступними вузлами;

– метод автоматичної інсталяції програм-агентів, що за допомогою протоколів віддаленого доступу дозволяє підготувати нові вузли системи до роботи;

– метод автоматичного створення серверів з використанням хмарних провайдерів, що дозволяє підготувати необхідну інфраструктуру для розширення кластера за допомогою хмарних ресурсів.

Для кожного з методів було розроблено відповідний алгоритм та блок-схему, детально розглянуто кожен крок алгоритму.

Було розроблено структуру графічного інтерфейсу користувача, розглянуто базові відомості про GUI та WebUI. Детально описано елементи головного вікна застосунку та кожної з вкладок інтерфейсу.

3 РОЗРОБКА ПРОГРАМНИХ КОМПОНЕНТ ЗАСТОСУНКУ

3.1 Варіантний аналіз і обґрунтування вибору засобів для реалізації програмного засобу

Більшість сучасних високорівневих мов програмування підходять для вирішення широкого спектру задач. Вибір конкретної мови визначає не тільки деякі характеристики кінцевого програмного продукту, а й значну кількість робочих процесів та склад екосистеми розробника, як-от доступ до окремих бібліотек, фреймворків, інструментів тощо.

При створенні програмного забезпечення для управління конфігураціями було проаналізовано можливості та екосистему чотирьох мов програмування, а саме C++, C#, Python та Go. Розглянемо більш детально кожну з них.

C++ є розвитком мови програмування C, яку розширили об’єктно-орієнтованою парадигмою та більш багатою стандартною бібліотекою. З порівнюваних мов програмування C++ є найстаршою – вона була стандартизована в 1998 році, отримує нові версії кожні декілька років, а також має декілька незалежних імплементацій в вигляді компіляторів від Microsoft, GNU, Intel тощо.

Основні переваги мови C++ [34]:

– швидкодія – компілятори для C++ за роки розвитку навчились робити велику кількість низькорівневих оптимізацій, що разом з іншим можливостями цієї мови дає можливість створювати одні з найбільш швидких додатків;

– керованість – ручний контроль над використанням та розподілом пам’яті дозволяє ще більше оптимізувати ПЗ, особливо при розробці для малопотужних пристроїв;

– сумісність з C – при розробці додатків на C++ можна застосовувати й бібліотеки від C, що значно розширює можливості розробника, особливо при роботі з низькорівневими процесами, драйверами, мультимедійними додатками тощо.

Основні недоліки C++:

– безпека роботи з пам’яттю – ручний контроль дає багато можливостей, але так само легко дозволяє створити додаткові проблеми в додатку через необережність або незнання окремих аспектів мови програмування, а «розумні» вказівники з більш нових стандартів лише частково вирішують це питання;

– складність – деякі низькорівневі можливості мови програмування, що C++ отримав від C, є складними для вивчення та застосування, а стандартна бібліотека хоч і є обширною, але менш комфортна для використання в порівнянні з іншими мовами;

– неоднорідність екосистеми – C++ в першу чергу є стандартом, який реалізовують різні компанії та спільноти у вигляді своїх власних компіляторів, тому можуть існувати деякі відмінності в роботі в різних середовищах; також через складність стандартизації відсутні деякі популярні інструменти розробників, як-от менеджери пакетів та залежностей – існують лише сторонні інструменти, які не мають повноцінної позиції на ринку та достатнього поширення в порівнянні з іншими офіційними застосунками.

Мова програмування C# була створена Microsoft у 2000 році в якості конкурента для Java та C++. Маючи подібний синтаксис та ряд зручних можливостей для розробників, як-от автоматичний менеджмент пам’яті та платформу .NET Framework, вона швидко отримала популярність на ОС Windows, а значно пізніше і на інших платформах.

Основні переваги мови C# [35]:

– екосистема – сучасна .NET екосистема надає можливість розробляти практично будь-які додатки без використання сторонніх бібліотек, майже все необхідне є в складі офіційних фреймворків;

– баланс продуктивності – додатки на C# рідко можуть досягти ефективності програм на C++, але дружелюбність мови до розробників створює хороше співвідношення між продуктивністю створюваного ПЗ та довжиною циклу розробки;

– підтримка – C# активно розвивається Microsoft та іншими компаніями-користувачами, тому регулярні оновлення виходять як для самої мови

програмування, так і для її офіційних інструментів, як-от Visual Studio; на відміну від C++ тут відсутній процес стандартизації мови і довгий цикл затвердження змін.

Недоліки C#:

– орієнтація на Windows – довгий час C# та .NET розроблялися під платформи від Microsoft, розробка на інші ОС та пристрої була хоч і можливою, але дуже обмеженою; нові версії .NET орієнтовані на кросплатформенну розробку, але не всі елементи екосистеми зараз повноцінно оновлені для цього, а також залишається велика кількість вже існуючого ПЗ, фреймворків та бібліотек на старих версіях .NET Framework;

– обмеженість сфер застосування – C# дуже добре підходить для створення десктопних та серверних додатків, але погано орієнтований на розробку для вбудованих систем або низькорівневого ПЗ;

– середовище виконання – в C# та ряді інших мов програмування від Microsoft застосовується віртуальна машина Common Language Runtime (CLR), тому кінцевий код додатку компілюється в проміжний байт-код для виконання на CLR, а не машинні інструкції; така архітектура спрощує запуск додатків на інших платформах та ОС, але може поставити додаткові обмеження по продуктивності й ефективності роботи в окремих сценаріях.

Python отримав свою популярність не через швидкодію або ефективність роботи з пам’яттю – в цих сферах ця мова програмування поступається іншим через інтерпретованість та динамічну типізацію. Код на Python більш простий для сприйняття в порівнянні з C-подібними мовами, а велика кількість синтаксичного цукру та обширна стандартна бібліотека роблять Python зручним у використанні. Гнучкість та доступність цієї мови призвели до її поширення в багатьох сферах – зараз Python використовують навіть для складних математичних моделей або штучного інтелекту.

Переваги Python [36]:

– простота – чистий та зрозумілий синтаксис дозволяють використовувати цю мову програмування як розробникам, так і спеціалістам з інших сфер, а

можливість інтеграції з іншими мовами дозволяє використовувати практично будь-який інструмент в такому форматі, що призвело до поширення Python за межі звичайних задач по розробці додатків;

– універсальність – обширна стандартна бібліотека Python та велика кількість сторонніх програмних компонентів дозволяють використовувати цю мову для веб-розробки, аналізу даних, машинного навчання або штучного інтелекту, наукових досліджень, автоматизації задач тощо; Python погано підходить лише для задач, що вимагають низькорівневого доступу або високої швидкодії і ефективного використання пам’яті;

– швидкий розвиток – Python розробляється та підтримується спільнотою, тому ця мова програмування регулярно отримує нові версії та виправлення помилок, а також відкриті дані дозволяють іншим командам створювати свої власні інтерпретатори Python з додатковими можливостями.

Основні недоліки Python:

– швидкодія – Python є інтерпретованою мовою програмування і значно поступається в швидкодії C++ та C#, а динамічна типізація та автоматичне керування пам’яттю помітно збільшують використання RAM;

– багатопотоковість – Python застосовує глобальне блокування інтерпретатора (англ. Global Interpreter Lock, GIL) для обмеження доступу окремих виконуваних потоків до інтерпретатора мови, що помітно ускладнює розробку багатопотокових програм та зменшує їх швидкодію;

– портативність – для виконання Python-скриптів необхідно попередньо встановити інтерпретатор та в деяких випадках ще завантажити сторонні бібліотеки, якщо такі застосовувались; цей крок необхідний як розробникам, так і кінцевим користувачам ПЗ, тому розповсюдження результуючого програмного продукту є більш складним процесом.

Go – мова програмування, що активно розвивається спільнотою та підтримується компанією Google, часто згадується також під назвою Golang. Ця мова програмування намагається поєднати швидкодію компільованих та зручність розробки інтерпретованих мов. Простий синтаксис, обширна

стандартна бібліотека та легка робота з багатопотоковими задачами роблять Golang зручним для розробки широкого спектру інструментального та системного ПЗ.

Основні переваги Go [37]:

– швидкодія – компілятор Golang збирає готові виконувані файли без застосування віртуальних машин або інших проміжних інструментів для середовища виконання, тому за швидкістю наближається до C та C++;

– багатопотоковість – Go дозволяє легко запустити будь-яку функцію в багатопотоковому режимі, а стандартна бібліотека має додаткові інструменти для управління одночасним доступом до даних, тому розробка високоефективних додатків є більш простою в порівнянні з іншими мовами;

– статичне компонування – за замовчуванням Go використовує лише статичне компонування бібліотек (динамічне може застосовуватись при явному використанні сторонніх компонентів, як-от DLL-файлів), тому вихідний виконуваний файл є повністю автономним та не потребує встановлення додаткових залежностей для роботи.

Недоліки мови Go:

– збірник сміття – для автоматичного управління пам’яттю в Golang застосовується збірник сміття, що полегшує розробку додатків, але ефективність використання RAM може бути меншою в порівнянні з іншими мовами в деяких сценаріях;

– шаблони програмування – Go має нестандартну реалізацію ООП, тому не всі класичні рішення ідеально підходять при розробці додатків на цій мові програмування, а досвідченим розробникам необхідний додатковий час на адаптацію;

– сфери застосування – компіляція в машинний код та швидкість розробки зробили Go популярним вибором для розробки різноманітного інструментального ПЗ та засобів автоматизації, особливо в сфері DevOps, але при створенні користувацьких GUI-додатків вибір інструментів та бібліотек значно менший.

На основі перерахованих переваг та недоліків було зроблено порівняння цих чотирьох мов програмування за певними критеріями, що описані в таблиці 3.1. Розглянемо більш детально ці критерії, особливості оцінювання та виставлені бали.

Таблиця 3.1 – Порівняння мов програмування

Критерій	C++	C#	Python	Go
Швидкодія розробленого ПЗ	1	1	0	1
Швидкість макетування та розробки	0	0,5	1	1
Управління залежностями	0	1	1	1
Підтримка багатоплатформності	1	0,5	0,5	1
Розвиток екосистеми мови	1	0,5	1	0,5
Підсумок	3	3,5	3,5	4,5

Швидкодія розробленого ПЗ – усі мови програмування, окрім Python, є компільованими. Додатки, розроблені на C++, C# та Golang можуть мати різну швидкодію через особливості роботи з пам’яттю, але вони все-одно будуть більш ефективними за інтерпретовані скрипти на Python.

Швидкість макетування та розробки – для сучасних комерційних додатків цей параметр може бути особливо важливим, адже дозволяє зробити цикл розробки помітно коротше й раніше віддати клієнту MPV (англ. Minimal Viable Product). Python та Go мають більш простий синтаксис, коли як C++ складніший у використанні й вимагає додаткових знань для коректного управління пам’яттю. C# займає проміжне місце між цими мовами.

Управління залежностями – C++ лише в останніх стандартах почав реалізовувати підтримку модулів, більшість компіляторів або не підтримують їх, або підтримка є лише експериментальною, сторонні інструменти для керування залежностями є малопопулярними. Решта мов програмування мають офіційні пакетні менеджери.

Підтримка багатоплатформності – цей критерій включає в себе простоту розробки та збірки проектів на різних платформах. З цього боку гірше себе

показує C# через потребу встановлення CLR та особливості роботи .NET платформи на сторонніх ОС, а також Python через проблеми з залежностями, що можуть використовувати попередньо скомпільовані компоненти.

Розвиток екосистеми мови – C++ та Python мають широкий вибір IDE та інших інструментів розробки, а C# та Golang частково обмежені в деяких категоріях. C# довгий час був орієнтований лише на розробку для платформ від Microsoft і має якісні офіційні інструменти, як-от Visual Studio, але екосистема для інших ОС почала швидко розвиватися лише після поширення кросплатформних версій .NET. Go має обширну екосистему бібліотек та фреймворків, але вибір IDE досі залишається невеликим.

За результатами порівняння найкраще підходить для розробки програмного забезпечення для управління конфігураціями мова Go. Її основні сфери застосування (DevOps, автоматизація тощо [38]) співпадають зі створюваним продуктом, а ефективність виконання, велика екосистема та швидкість розробки дозволять покращити результуючий застосунок.

Таким чином, в даному підрозділі було розглянуто переваги й недоліки мов програмування C++, C#, Python та Go. Наведено короткі відомості про кожну мову, розглянуто основні переваги та недоліки, сформовано таблицю з порівнянням на основі ряду критеріїв, детально розглянуто кожен проаналізований критерій та виставлені оцінки. За результатами порівняння мову програмування Go було обрано для розробки програмного продукту.

3.2 Вибір середовища розробки

Інтегроване середовище розробки (англ. Integrated Development Environment, IDE) – це програмне забезпечення, що надає розробникам інструменти для більш ефективної розробки, тестування та налагодження коду створюваного додатку. Зазвичай це централізоване середовище, яке дозволяє у зручному режимі за допомогою графічного інтерфейсу використовувати редактор коду, компілятор або інтерпретатор, налагоджувач, різноманітні інтеграції з системами керування версіями тощо.

Для мови програмування Go існує лише декілька повноцінних IDE, але за допомогою підтримки LSP є можливість використовувати цю мову програмування в більшості популярних редакторах. LSP (Language Server Protocol) – це відкритий протокол комунікації редакторів або IDE з мовними серверами, що був розроблений компанією Microsoft.

Мовний сервер відповідає за перевірку помилок, підказки синтаксису, автоматичне завершення коду – він здатен замінити вбудований в IDE аналізатор коду. Тобто, з’являється можливість застосувати практично будь-яку мову програмування для якої існує LSP-сумісний мовний сервер в будь-якому редакторі, що підтримує цей протокол.

Для розробки на Go популярними є Visual Studio Code, LiteIDE та GoLand. Розглянемо кожен з програмних продуктів більш детально.

Visual Studio Code – це не повноцінна IDE, а редактор коду від Microsoft, що має відкритий вихідних код та активно підтримується спільнотою. Приклад графічного інтерфейсу наведено на рисунку 3.1.

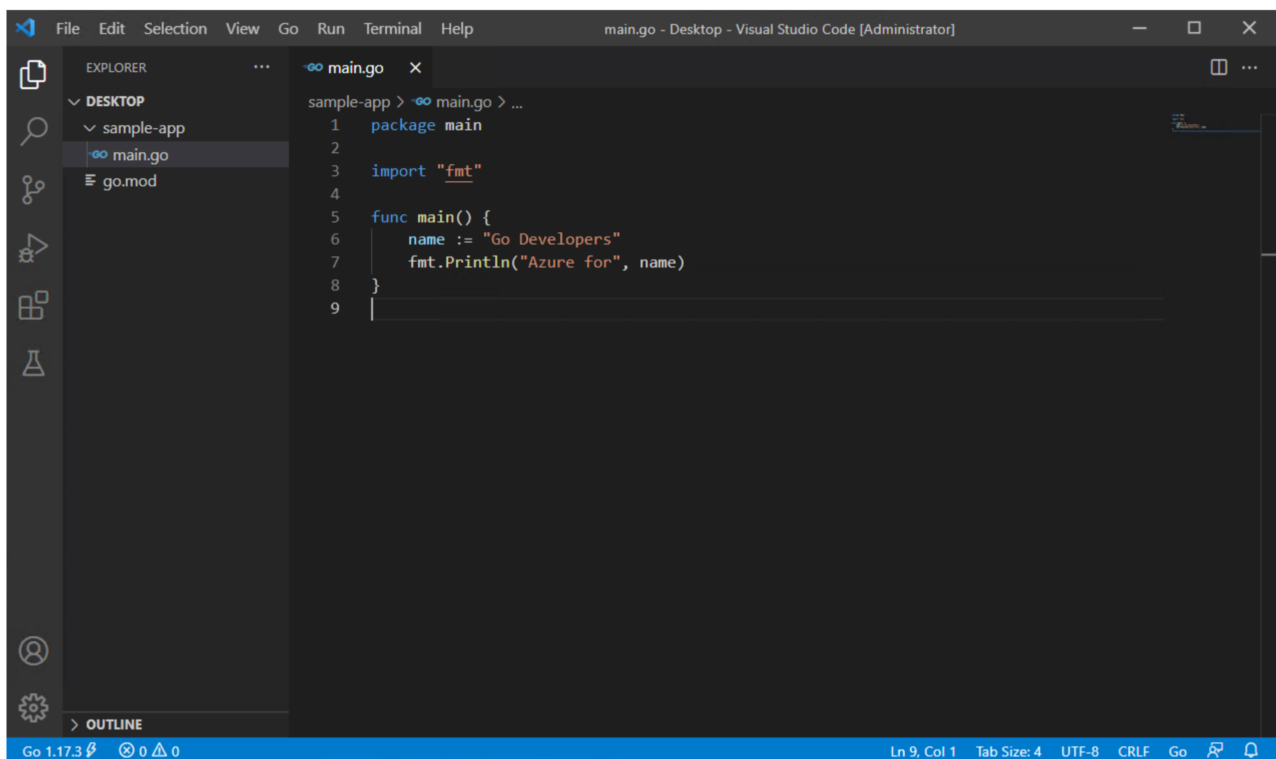


Рисунок 3.1 – Приклад інтерфейсу Visual Studio Code

Базовий функціонал VS Code поступається іншим продуктам, але обширний каталог плагінів дозволяє перетворити цей редактор коду на зручний інструмент для розробки. Швидкодія, розширюваність та безкоштовність – це основні переваги цього продукту над іншими. Для роботи з Golang існує добре підтримуваний плагін з підтримкою Gopls – офіційного мовного сервера для мови Go. Основні недоліки VS Code – це менш розвинуті системи рефакторингу та збірки проектів [39].

LiteIDE – це повноцінне інтегроване середовище розробки, що спеціалізується на роботі з Go. Має відкритий вихідний код та розробляється силами спільноти з 2012 року. Основними перевагами LiteIDE є швидкодія, безкоштовність та широка підтримка різних версій Golang – середовище зможе працювати як з новими проектами, так і з застарілим кодом, що не має підтримки модулів та інших сучасних функцій Go.

Приклад графічного інтерфейсу LiteIDE зображено на рисунку 3.2.

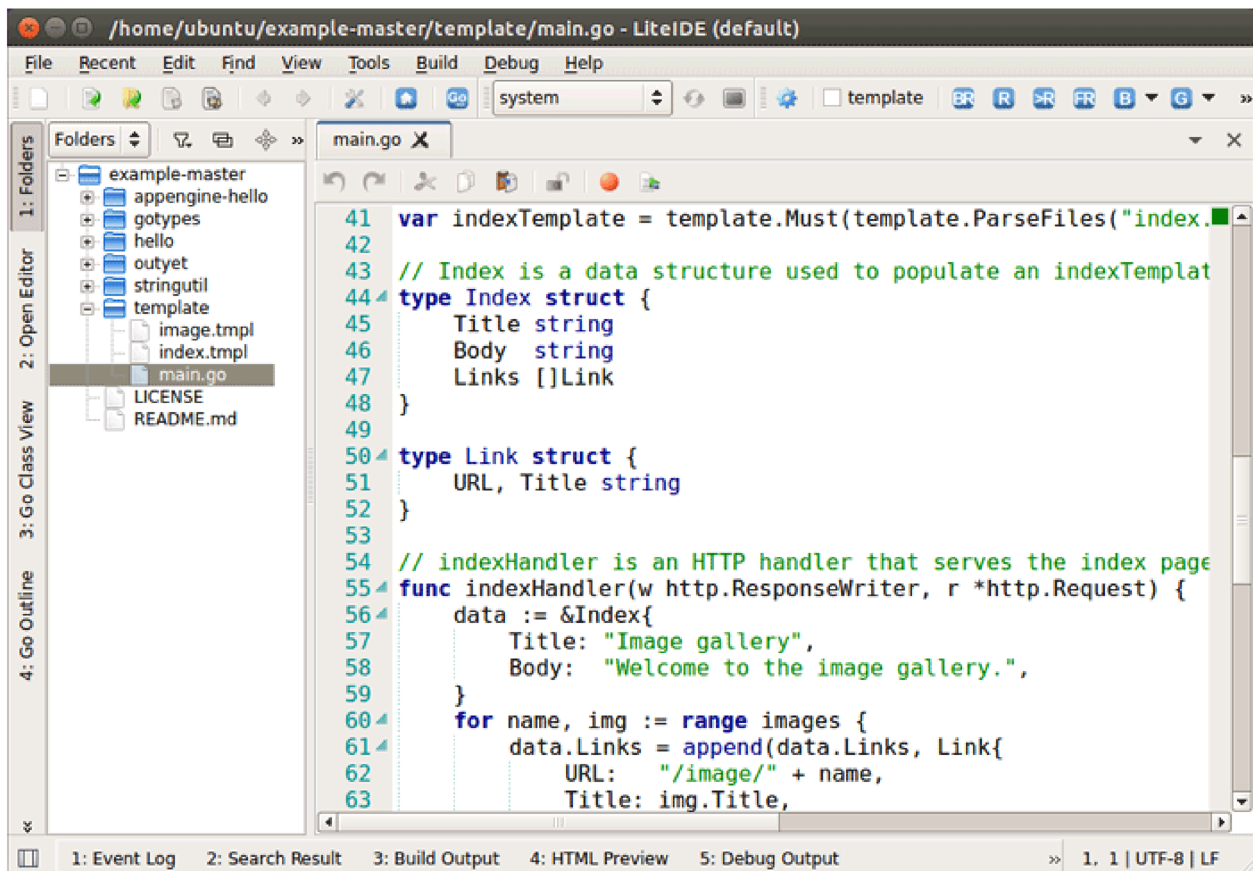


Рисунок 3.2 – Приклад графічного інтерфейсу LiteIDE

Основними недоліками цього IDE є погана розширюваність (наявна підтримка плагінів, але вони малочисельні й для їх розробки необхідно використовувати C++, що менш зручно та більш повільно для сторонніх розробників) та відсутність інтеграцій з іншими зовнішніми інструментами, тому LiteIDE гірше взаємодіє з існуючою екосистемою Golang [40].

GoLand – це повноцінній комерційний продукт від компанії JetBrains, яка добре відома своїми інтегрованими середовища розробки для різних мов програмування. GoLand, як і більшість інших IDE від JetBrains, базується на основі Intellij IDEA, тому має якісний аналізатор коду, зручні інструменти для рефакторингу та обширну базу плагінів для розширення функціоналу [41]. Основним недоліком цього середовища є підвищене споживання ресурсів – для комфортної роботи в GoLand необхідно мати більш потужний пристрій, аніж для інших IDE або середовищ розробки.

Приклад графічного інтерфейсу GoLand наведено на рисунку 3.3.

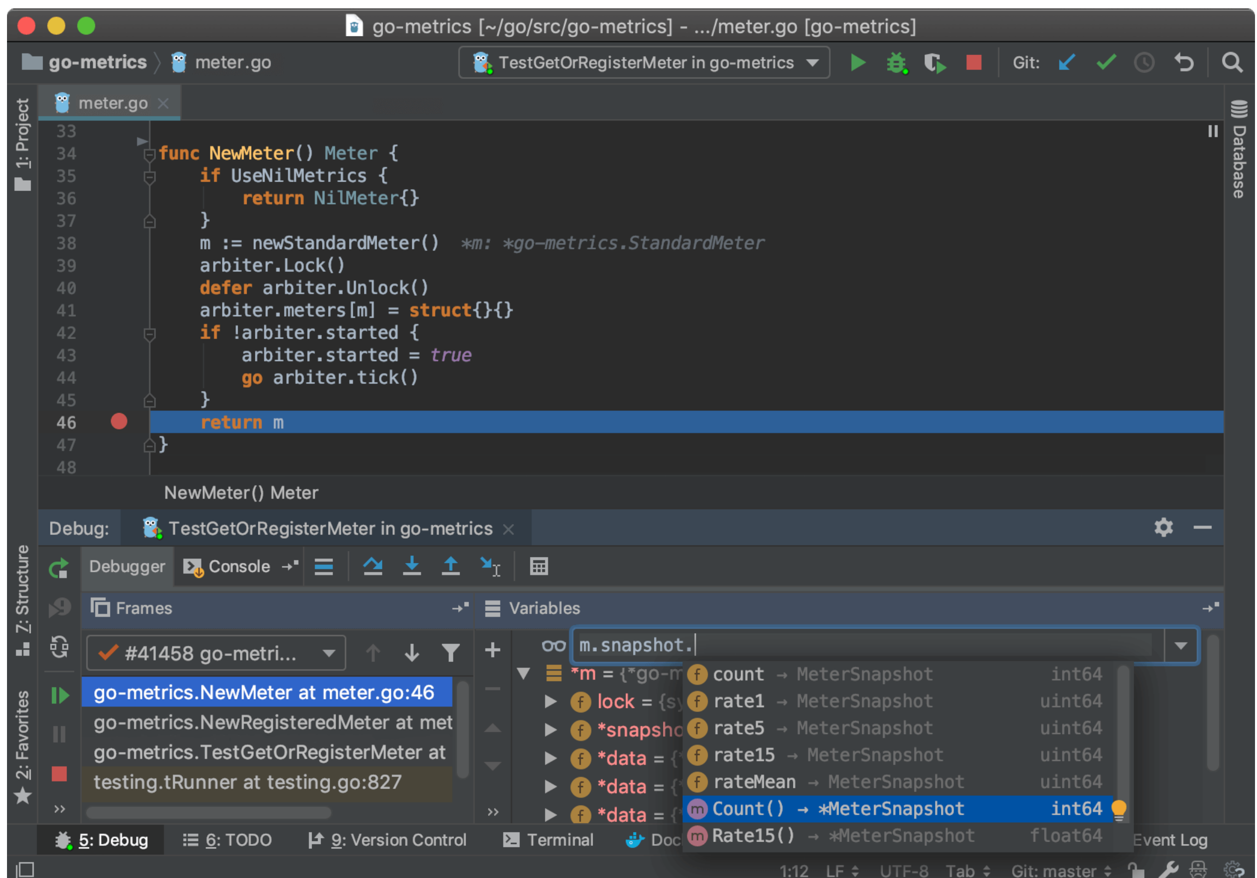


Рисунок 3.3 – Приклад графічного інтерфейсу GoLand

Для об'єктивного порівняння розглянутих середовищ розробки було відібрано ряд критеріїв. Результати порівняння наведено в таблиці 3.2. Розглянемо більш детально критерії оцінювання.

Таблиця 3.2 – Порівняння засобів розробки

Критерій	VS Code	LiteIDE	GoLand
Швидкодія та використання ресурсів	0,5	1	0,5
Розширюваність	1	0,5	1
Підтримка ОС та платформ	1	1	1
Засоби рефакторингу коду	0,5	0,5	1
Засоби аналізу коду	1	0,5	1
Підсумок	4	3,5	4,5

Швидкодія та використання ресурсів – розроблене на C++ середовище розробки LiteIDE споживає найменше ресурсів та має непогану швидкодію відносно до свого функціоналу. VS Code та GoLand є більш вимогливими до робочої машини розробника, хоча також показують хорошу швидкодію при виконанні більшості типових задач.

Всі розглянуті програмні продукти мають можливість розширення функціоналу за допомогою системи плагінів. VS Code є редактором, а не повноцінною IDE, тому орієнтований на застосування плагінів і має обширну базу для практично будь-яких задач. GoLand базується на платформі Intellij IDEA, тому сумісний з великою кількістю плагінів від інших IDE від JetBrains. Найбільше відстає в цьому порівняння LiteIDE – його база плагінів значно менша, а їх розробка більш складна через застосування C++.

За критерієм підтримки різних операційних систем та платформ усі розглянуті продукти отримують по балу. LiteIDE може бути зібраний у вигляді нативного додатку для потрібної платформи, VS Code працює на будь-якій ОС, що підтримується фреймворком Electron, а GoLand побудований за допомогою Java та поставляється зі своєю власною версією JRE, що дозволяє йому працювати на широкому спектрі обладнання.

GoLand має одні з найкращих засобів для рефакторингу та аналізу коду – це саме ті інструменти, які зробили це середовище та інші продукти на базі Intellij IDEA від JetBrains такими популярними серед розробників. LiteIDE поступається по кількості функціоналу та якості його виконання, тому отримує по половині бала за кожним критерієм. VS Code має якісний вбудований аналізатор коду, що був розроблений Microsoft, а для підтримки Go застосовується протокол LSP та відповідний мовний сервер Gopls. Засоби рефакторингу достатні для звичайного редактора, але вони не такі функціональні, як в GoLand. Таким чином, VS Code отримує 1 та 0,5 бали за аналіз коду та рефакторинг відповідно.

Таким чином, в даному підрозділі було розглянуто переваги та недоліки популярних редакторів та середовищ розробки для мови програмування Go, як-от Visual Studio Code, LiteIDE та GoLand. Проведено аналіз цих трьох продуктів за рядом критеріїв, детально розглянуто виставлені оцінки. За результатами порівняння найбільш підходящим середовищем розробки для створення ПЗ для управління конфігураціями було обрано GoLand від JetBrains.

3.3 Програмна реалізація застосунку

Під час програмної реалізації додатку для управління конфігураціями було розроблено алгоритми, що описані в підрозділах 2.1-2.3, а також велику кількість додаткових програмних компонент, що необхідні для коректного функціонування результуючого продукту. Розглянемо більш детально код програмного забезпечення.

В мові програмування Go, аналогічно до багатьох інших мов, використовується концепція пакетів для логічного та концептуального розділення окремих структур, методів та функцій за їх призначенням. Застосунок включає в себе наступні пакети:

– config – містить основні структури даних, що необхідні для ініціації ПЗ, а також додаткові компоненти для їх серіалізації та десеріалізації;

– state – один з центральних пакетів, що містить структури та функції для роботи з локальним станом вузла та загальним реплікованим станом кластера,

пакет включає реалізацію основних методів для досягнення високої доступності програмного забезпечення;

– cluster – допоміжний пакет, що спрощує доступ до локального стану та стану кластера для інших програмних компонент;

– web – відповідає за обслуговування статичних ресурсів, що необхідні для роботи WebUI (frontend-частина ПЗ), а також реалізує REST API (backend-частина ПЗ), через який запускається переважна частина іншого функціоналу додатку;

– cloud – містить методи та структури даних для роботи з хмарними провайдерами, що необхідні для роботи алгоритму автоматичного створення серверів;

– remote – містить методи та структури даних для підключення до віддалених серверів та виконання типових дій по підготовці вузлів кластера, що необхідні для роботи алгоритму автоматичної інсталяції програм-агентів.

Крім того, будь-яка програма на мові Go, що збирається в виконуваний файл, містить пакет main з кодом для точки входу. Пакет main може мати довільну структуру, але компілятор буде шукати в ньому саме функцію main без аргументів та значень, що повертаються. Ця функція і є вхідною точкою виконуваного файлу.

Вхідні параметри, що могли бути передані через інтерфейс командного рядка або змінні середовища, можна отримати за допомогою окремих пакетів стандартної бібліотеки – вони не доступні в якості аргументів функції main, як це реалізовано в деяких інших мовах програмування. Розглянемо більш детально точну входу для розроблюваного ПЗ.

Першим етапом ініціації вузла програмної системи є зчитування файлу конфігурації. Ім’я файлу може бути передане користувачем при запуску з командного рядка або може використовуватися значення за замовчуванням. Застосунок намагається знайти конфігураційний файл відносно директорії виконуваного файлу, потім вичитує його вміст в бінарному форматі та намагається його десеріалізувати. Для файлу конфігурації використовується

формат YAML, який більш зручний для сприйняття кінцевим користувачем, аніж JSON або XML. Для логування помилок та додаткових інформаційних або налагоджувальних повідомлень застосовується бібліотека zerolog, що на відміну від пакетів стандартної бібліотеки дозволяє застосовувати принцип структурованого логування.

Код, що відповідає за вичитування та логування конфігурації вузла, наведено на рисунку 3.4.

```
23 func main() {  Andrii Myrhorodskyi
24     // Config
25     configFilename := flag.String( name: "config",  value: "config.yaml",  us
26
27     bytes, err := os.ReadFile(*configFilename)
28     if err ≠ nil {
29         log.Fatal().Err(err).Msg( msg: "Failed to read config file")
30     }
31
32     var cfg config.Config
33     err = yaml.Unmarshal(bytes, &cfg)
34     if err ≠ nil {
35         log.Fatal().Err(err).Msg( msg: "Failed to unmarshal config file")
36     }
```

Рисунок 3.4 – Лістинг коду для роботи з файлом конфігурації

На рисунку 3.5 наведено структури даних, що використовуються для роботи з конфігурацією вузла.

```
3  type Config struct {  9 usages  Andrii Myrhorodskyi
4      Node Node `yaml:"node"`
5  }
6
7  type Node struct {  2 usages  Andrii Myrhorodskyi
8      GRpcAddress string `yaml:"grpc_address"`
9      WebAddress  string `yaml:"web_address"`
10     LocalID     string `yaml:"local_id"`
11     DataDir     string `yaml:"data_dir"`
12     LogsFile    string `yaml:"logs_file"`
13     StableFile  string `yaml:"stable_file"`
14     LocalFile   string `yaml:"local_file"`
15     Bootstrap   bool   `yaml:"bootstrap"`
16 }
```

Рисунок 3.5 – Структура даних з тегами для роботи з конфігурацією

Go використовує спеціальні теги, що об’являються після типу змінної, які визначають додаткові параметри для процесів серіалізації та десеріалізації даних. В даному випадку, тег «yaml» для кожного параметра структури визначає назву відповідного ключа в YAML-файлі.

Без застосування тегів при серіалізації використовуються назви самих змінних з відповідним форматуванням (CamelCase у випадку Go), що може бути небажано для підтримки єдиної конвенції щодо іменування в YAML-документах. Крім того, за допомогою тегів можна передати додаткові налаштування: виключити окреме поле при серіалізації, задати параметри обробки комплексних структур, ігнорувати окремі поля при нульовому значенні тощо.

Приклад YAML-файлу, який може бути зчитаний структурою Config з рисунку 3.5, зображено на рисунку 3.6.

```
1  node:
2    grpc_address: 127.0.0.1:3000
3    web_address: 127.0.0.1:3001
4    local_id: raft_node_1
5    data_dir: ./data/
6    logs_file: logs.dat
7    stable_file: stable.dat
8    local_file: local.json
9    bootstrap: false
```

Рисунок 3.6 – Приклад конфігураційного файлу, що відповідає використаним в структурі тегам

Наступним кроком є підготовка до роботи серверів. Одночасно кожен вузол системи запускає звичайний HTTP-сервер для обслуговування статичних файлів для WebUI та точок доступу до REST API, а також окремий сервер для роботи з gRPC. WebUI запускається на всіх вузлах з метою уникнення потенційних мережевих проблем – окремий вузол може продовжувати роботу в складі системи, але не дозволяти доступним по протоколам HTTP/HTTPS. Якщо

запускати frontend та backend частини лише на одному з серверів, то така мережева конфігурація може відрізати доступ кінцевого користувача до системи. Єдиний недолік такого підходу – це потреба в більш обережному проектуванні API, адже одні й ті самі операції можуть бути викликані одночасно на різних вузлах через різні точки доступу. Проте, в ПЗ для управління конфігураціями застосовується режим високої доступності на основі алгоритму консенсусу, що спрощує синхронізацію даних та одночасний доступ до них.

gRPC (Google Remote Procedure Call) – це система віддаленого виклику процедур, що базується на основі протоколу HTTP/2 та формату Protobuf для опису інтерфейсів [42]. Основні функції gRPC застосовують нові можливості HTTP/2, як-от мультиплексування запитів та відповідей, використання бінарного кодування замість текстового в HTTP/1.1, стиснення даних заголовків повідомлень, server-side push повідомлення тощо. Це дозволяє створювати більш ефективні системи, що готові до високого навантаження у реальному часі. Крім того, функція кодогенерації в Protobuf дозволяє швидко реалізувати gRPC-клієнт для будь-якої мови програмування.

З іншого боку, такі нововведення роблять gRPC незручним для використання в якості звичайного HTTP API або REST API – через ряд обмежень цю систему не застосовують для комунікації між frontend та backend частинами програмного забезпечення. Тому й було прийнято рішення застосовувати обидва інструменти у додатку для управління конфігураціями: класичний HTTP-сервер відповідає за REST API та WebUI, а gRPC-сервер забезпечує комунікацію між вузлами та підтримує роботу інших внутрішніх компонентів ПЗ – він дозволяє оперувати декількома різними сервісами на одному сервері під одним мережевим портом.

Для розробки REST API було використано Gin – популярний веб-фреймворк для Golang. Він спрощує ряд типових операцій та має додаткові компоненти для більш гнучкого налаштування сервера й створення додаткових інтеграцій. В функції main створюється екземпляр об’єкта-маршрутизатора, якому передаються деякі базові налаштування, а вже потім він використовується

в якості одного з аргументів для функцій пакету web. В першу чергу налаштовується логування запитів, компонент для відновлення роботи сервера при виникненні критичних (рівень «panic») помилок та CORS-політики, що необхідні для більш коректного налаштування доступу до API. Описані кроки наведено на рисунку 3.7.

```
40  // Web Server
41  router := gin.New()
42  ginLogger := log.With().Str(key: "component", val: "gin").Logger()
43  router.Use(gin.LoggerWithConfig(gin.LoggerConfig{Output: ginLogger}))
44  router.Use(gin.Recovery())
45  router.Use(cors.Default())
```

Рисунок 3.7 – Лістинг коду для підготовки маршрутизатора HTTP-сервера

Крім того, в цей момент створюється об’єкт вузла кластера на основі попередньо отриманої конфігурації, що є частиною алгоритму по забезпеченню високої доступності. Цей об’єкт використовується для забезпечення роботи REST API, що наведено на рисунку 3.8.

```
47  // UI
48  err = web.RegisterUI(router)
49  if err ≠ nil {
50      log.Fatal().Err(err).Msg(msg: "Failed to start WebUI")
51  }
52
53  // Cluster
54  cl, err := cluster.CreateFromConfig(cfg)
55  if err ≠ nil {
56      log.Fatal().Err(err).Msg(msg: "Failed to create cluster from config")
57  }
58
59  // API
60  web.RegisterAPI(router, cl)
```

Рисунок 3.8 – Лістинг коду для створення об’єкта вузла кластера та підключення WebUI та REST API до маршрутизатора

Стандартна бібліотека Go містить функції для запаковування файлів у кінцевий виконуваний файл, що позбавляє необхідності додатково розповсюджувати ресурси WebUI. Крім того, можна отримати доступ до

запакованих ресурсів у вигляді віртуальної файлової системи. Таку систему можна напряму використовувати в маршрутизаторі для обслуговування статичних ресурсів, що наведено на рисунку 3.9.

```
11  func RegisterUI(r *gin.Engine) error {  1 usage  Andrii Myrhorodskyi
12      staticFS := fs.FS(embedres.WebUiRes)
13      staticContent, err := fs.Sub(staticFS, dir: "web/dist")
14      if err ≠ nil {
15          return fmt.Errorf("failed to make FS for static content: %w", err)
16      }
17
18      r.StaticFS( relativePath: "/ui", http.FS(staticContent))
19
```

Рисунок 3.9 – Лістинг функції RegisterUI з пакету web

Функція RegisterAPI приймає екземпляр вузла кластера, з якого вона може отримати більшу частину необхідних даних для забезпечення роботи API або ж виконувати певні дії на основі запитів користувача. Функція реєструє усі необхідні кінцеві точки доступу на маршрутизаторі та відповідні їм функції-обробники. Також застосовується детальне логування помилок та групування запитів на основі шляху. Лістинг фрагмента функції RegisterAPI зображено на рисунку 3.10.

```
23  func RegisterAPI(router *gin.Engine, cl *cluster.Cluster) {  1 usage
24      apiRouter := router.Group( relativePath: "/api")
25      logger := log.With().Str( key: "component",  val: "api").Logger()
26
27      clusterRouter := apiRouter.Group( relativePath: "/cluster")
28
29      clusterRouter.GET( relativePath: "/fsm", func(c *gin.Context) {
30          c.JSON( code: 200, cl.ClusterState.State)
31      })
```

Рисунок 3.10 – Фрагмент лістингу функції RegisterAPI з пакету web

Після реєстрації REST API та WebUI в функції main створюється екземпляр gRPC-сервера, на якому відразу реєструється сервіс комунікації

кластера, що є частиною алгоритму забезпечення високої доступності. Маршрутизатор з фреймворку Gin сумісний з веб-сервером стандартної бібліотеки Go, який і було застосовано в додатку. Крім того, в цей же момент створюється канал для системних сигналів, як-от SIGTERM, щоб можна було коректно обробити команду завершення виконання. Цей фрагмент наведено на рисунку 3.11.

```
62      // gRPC server
63      grpcServer := grpc.NewServer()
64      cl.Transport.Register(grpcServer)
65
66      // Control flow
67      webServer := &http.Server{Addr: cfg.Node.WebAddress, Handler: router}
68
69      c := make(chan os.Signal)
70      signal.Notify(c, os.Interrupt, syscall.SIGTERM)
```

Рисунок 3.11 – Лістинг коду для створення об’єктів-серверів та каналу для системних сигналів

Канали в Golang застосовуються для впорядкованого обміну даними між потоками. Golang має спеціальне ключове слово «go», яке дозволяє практично будь-яку функцію запустити в окремому потоці без додаткових налаштувань. Більш коректним терміном є слово «goroutine», що застосовується в офіційній документації – це полегшений потік виконання в Go, який є більш простим в використанні в порівнянні з класичним багатопотоковим програмуванням в інших мовах.

Канал – це FIFO-черга для змінних певного типу, яку можна застосовувати для зручного керування логікою виконання при використанні декількох потоків [43]: в даному випадку обидва сервери запускаються паралельно в окремих потоках, а канал блокує головний потік функції main в очікуванні сигналу SIGTERM. Аналогічним чином можна реалізувати паралелізм практично будь-якої частини програмного забезпечення. Запуск серверів в багатопотоковому режимі наведено на рисунку 3.12.

```
80    go func() {
81        sock, err := net.Listen( network: "tcp", cfg.Node.GRpcAddress)
82        if err ≠ nil {
83            log.Error().Err(err).Msg( msg: "Failed to listen gRPC port")
84            return
85        }
86        err = grpcServer.Serve(sock)
87        if err ≠ nil {
88            log.Error().Err(err).Msg( msg: "Failed to start gRPC server")
89            return
90        }
91    }()
```

Рисунок 3.12 – Лістинг коду для запуску серверів в окремих потоках

Крім того, канали можуть блокувати потік в очікуванні інформації та застосовуватись разом з конструкціями типу «switch-case» – в такому випадку можливо одночасно прослуховувати відразу декілька каналів та будувати різну логіку в залежності від того, який канал першим отримає дані. Приклад такої поведінки наведено на рисунку 3.13.

```
93    ticker := time.NewTicker(10 * time.Second)
94    quit := make(chan struct{})
95    go func() {
96        for {
97            select {
98            case ←ticker.C:
99                cl.UpdateState()
100
101           case ←quit:
102               ticker.Stop()
103               return
```

Рисунок 3.13 – Лістинг коду для обробки декількох каналів

В даному випадку створюється два канали: ticker, що спрацьовує з інтервалом в десять секунд, а також quit, що необхідний для коректного завершення роботи потоку. Перший канал автоматично буде отримувати повідомлення з регулярним інтервалом і за допомогою конструкції «switch-case» (в Go замість ключового слова «switch» використовується «select») буде запускати функцію cl.UpdateState. Вона відповідає за перевірку спільного стану

кластера, порівняння його з локальним станом і запуск додаткових дій у разі виявлення різниці між локальним і загальним станами. Таким чином, ця конструкція забезпечує функціонування фінальної частини алгоритму по забезпеченню високої доступності.

Остання частина функції main наведена на рисунку 3.14.

```
108    <-c
109    close(quit)
110    grpcServer.GracefulStop()
111    log.Info().Msg( msg: "Stopped gRPC server")
112    err = webServer.Shutdown(context.Background())
113    if err != nil && !errors.Is(err, http.ErrServerClosed) {
114        log.Error().Err(err).Msg( msg: "Failed to shutdown web server")
115    }
116    log.Info().Msg( msg: "Stopped web server")
```

Рисунок 3.14 – Лістинг коду для зупинки серверів та роботи програми

По-перше, ця частина коду виконується вже при зупинці програмного забезпечення – потік виконання блокується і очікує отримання сигналу SIGTERM. Як тільки сигнал отримано, канал quit закривається та почергово вимикаються обидва сервери – це спричиняє контрольоване завершення роботи всіх попередньо відкритих потоків. Останнім кроком виступає серіалізація локального стану вузла, що зберігається окремо від загального стану кластеру, який необхідний для режиму високої доступності.

Два інші алгоритми, що необхідні для реалізації методів по автоматичній інсталяції програм-агентів та створення серверів, було реалізовано в пакеті web в якості окремих точок доступу в REST API – в такому форматі ці методи можуть бути запущені як кінцевим користувачем, так і іншим компонентом ПЗ. Обидва методи очікують POST-запитів з вхідними даними, що необхідні для роботи, тому перший крок – це обробка вхідних даних, а саме їх десеріалізація з формату JSON в локальну структуру даних, що наведено на рисунку 3.15.

Після цього для автоматичної інсталяції програм-агентів починається цикл, що містить основні кроки алгоритму. Встановлюється з’єднання з

віддаленим сервером, на основі параметрів кластера генерується конфігурація для нового вузла та відбувається копіювання усіх необхідних файлів на сервер. Після запуску додатку на новому сервері відбувається перевірка доступності цього вузла в кластері

```
213 clusterRouter.POST( relativePath: "/install", func(c *gin.Context) {
214     connData := remote.ConnectionData{}
215     err := c.BindJSON(&connData)
216     if err ≠ nil {
217         logger.Error().Err(err).Msg( msg: "Failed to parse input JSON")
218         return
219     }
```

Рисунок 3.15 – Лістинг коду для обробки вхідних даних

Для виконання дій на віддаленому сервері використовується бібліотека Goph, що є більш функціональною обгорткою для стандартного SSH-клієнта, який можна застосовувати для роботи як з UNIX-сумісними системами, так і з ОС Windows з застосуванням OpenSSH. Перевірка доступності полягає в отриманні поточних членів кластера та пошуку серед них нового вузла – при успішному розгортанні вузол буде приєднано до кластера та почнеться реплікація загального статусу системи й приведення локального стану в бажаний стан. При виникненні помилок на будь-якому з кроків алгоритму передбачено їх логування та повернення відповідного статусу для запиту в REST API.

Фрагмент коду для інсталяції агентів наведено на рисунку 3.16.

```
221 for i := 0; i < connData.RetryCount; i++ {
222
223     conn, err := remote.Connect(&connData)
224     if err ≠ nil {
225         logger.Error().Err(err).Msg( msg: "Failed to connect to remote host")
226         return
227     }
228
229     clusterInfo := cl.GetInfo()
230     conf, err := cluster.GenerateConfig(clusterInfo, &connData)
231     if err ≠ nil {
232         logger.Error().Err(err).Msg( msg: "Failed to generate config")
233         return
```

Рисунок 3.16 – Фрагмент лістингу коду для інсталяції програм-агентів

Реалізація алгоритму автоматичного створення серверів з використанням хмарних провайдерів також починається з десеріалізації вхідних даних, після чого створюється клієнт для роботи з хмарою. Далі виконуються інші кроки алгоритму в відповідності до блок-схеми з підрозділу 2.5: перевірка та створення мережевої інфраструктури, підготовка серверів, тестування їх доступності та опціональне видалення ресурсів при виникненні помилок.

В рамках магістерської кваліфікаційної роботи для розробки та тестування було використано хмарного провайдера AWS, тому для реалізації компонентів алгоритму було використано офіційний AWS SDK для мови Go. Проте, алгоритм не покладається на особливості роботи того чи іншого хмарного провайдера, тому при введенні створюваного ПЗ в експлуатацію можливо швидко розробити інтеграцію з іншими хмарними рішеннями, як-от Microsoft Azure або Google Cloud Platform. Фрагмент лістингу коду для алгоритму створення серверів наведено на рисунку 3.17.

```
277 	cl, err := cloud.NewClient(&provisionInput.AccessConfig)
278 	if err ≠ nil {
279 		logger.Error().Err(err).Msg( msg: "Failed to create cloud client")
280 		return
281 	}
282 
283 	if provisionInput.NetworkConfig.Exists {
284 		networkReady, err := cl.IsExist(&provisionInput.NetworkConfig)
285 		if err ≠ nil {
286 			logger.Error().Err(err).Msg( msg: "Failed to check network config")
287 			return
288 		}
```

Рисунок 3.17 – Фрагмент лістингу коду алгоритму автоматичного створення серверів

Реалізація алгоритму для AWS покладається на створення хмарних ресурсів за допомогою сервісу AWS CloudFormation, що дозволяє описати інфраструктуру в текстовому вигляді в форматі YAML. Інші хмарні провайдери мають аналогічні сервіси: наприклад, Azure Automation та Google Cloud Deployment Manager. Таке рішення дозволяє спростити використання ПЗ для тих

користувачів, які вже знайомі з цими сервісами та синтаксисом опису інфраструктури в них, а також позбавляє необхідності розробляти власну обгортку навколо SDK для таких задач.

Приклад застосування SDK для створення серверів наведено на рисунку 3.18.

```
79  func (client *Client) PrepareInstances(serversConfig *ServersConfig, networkConfig *NetworkConfig)
80      if len(networkConfig.StackId) == 0 : nil, fmt.Errorf("network infra is not ready") ↵
83
84      stack, err := client.cloudformation.CreateStack(context.Background(), &cloudformation.CreateSt
85          StackName:   &serversConfig.StackName,
86          TemplateURL: &serversConfig.StackURL,
87      })
88      if err ≠ nil : nil, err ↵
91
92      serversConfig.StackId = *stack.StackId
93
94      output, err := client.cloudformation.ListStackResources(context.Background(), &cloudformation.
95          StackName: &serversConfig.StackName,
96      })
97      if err ≠ nil : nil, err ↵
```

Рисунок 3.18 – Лістинг коду методу створення серверів

Таким чином, в даному підрозділі було розглянуто програмну реалізацію основних алгоритмів та інших компонентів створюваного програмного продукту. Повний лістинг наведено у додатку Б.

3.4 Висновки

В даному розділі було проведено аналіз мов програмування C++, C#, Python та Go, а також середовищ розробки VS Code, LiteIDE та GoLand. Порівняння цих рішень за рядом критеріїв дозволило вибрати мову програмування Go та IDE JetBrains GoLand для розробки програмного засобу «Detrint». Також було детально розглянуто програмну реалізацію окремих компонентів продукту.

4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Аналіз методів тестування програмного забезпечення

Тестування програмного забезпечення – це один з процесів розробки ПЗ, що передбачає оцінку та перевірку відповідності програмного продукту поставленим вимогам. Основною метою тестування є виявлення дефектів, невідповідностей вимогам та інших потенційних проблем, що потребують виправлень перед повноцінним релізом програмного забезпечення для кінцевих користувачів.

Крім виявлення помилок, тестування дозволяє покращити ряд інших аспектів програмного продукту [44]:

– зменшення витрат на розробку – чим раніше проблему буде виявлено, тим дешевше обійдеться її виправлення; пізно знайдений дефект може вимагати додаткової трудомісткої підтримки та обслуговування кінцевих клієнтів;

– зворотній зв'язок та вдосконалення продукту – розробники та користувачі часто можуть по-різному бачити й сприймати програмне забезпечення, включаючи його функціонал, UI та UX, тому тестування може дати цінну інформацію про потенційні шляхи покращення продукту;

– безпека – захист конфіденційних даних та персональної інформації користувачів має першочергове значення при перевірці продукту на потенційні вразливості; несанкціонований доступ до даних може нести репутаційні, фінансові та правові ризики;

– конкурентна перевага та довіра користувачів – продукт з меншою кількістю дефектів на релізі може мати перевагу над аналогами через кращу підготовку до використання.

Тестування програмного забезпечення може по-різному інтегруватися в процес розробки в залежності від його типу. Зазвичай тестування класифікують наступним чином:

– функціональне тестування – сюди відносять компонентне (unit-тестування), інтеграційне та системне тестування, що мають на меті перевірити

основний функціонал програмного продукту на різних рівнях; часто таке тестування є автоматизованим та проводиться під час виконання різноманітних CI/CD-процесів;

– нефункціональне тестування – до цієї категорії відносять перевірки безпеки, швидкодії та надійності, а також різноманітні процеси тестування зручності й доступності для користувачів або ж тестування сумісності з різними пристроями;

– регресивне тестування – це повторна перевірка сценаріїв тестування, що до цього були успішні; така процедура потрібна для виявлення потенційних проблем в старому функціоналі при розробці оновлень;

– «Smoke» та «Sanity» тестування – процедури по швидкій перевірці основного функціоналу ПЗ для відповідно нестабільних та стабільних внутрішніх релізів, що визначають потребу в подальшому більш комплексному тестуванні.

Окрім наведеної класифікації існує велика кількість інших видів тестування, деякі з них можуть підпадати відразу під декілька категорій. Наприклад, популярним є тестування «чорного/білого/сірого ящика» – в такому випадку зміст сценаріїв тестування визначається ступенем знання системи та її побудови. Тестування «чорного ящика» передбачає повну відсутність такої інформації, тому цей вид перевірок найбільш наближений до реальних сценаріїв використання, коли як тестування «білого ящика» та «сірого ящика» застосовують відповідно повну або часткову інформацію про будову ПЗ.

Тестування методом «чорного ящика» має ряд переваг, як-от:

– імітація дій та сприйняття користувача – коректно складені сценарії тестування дозволяють виявити не тільки функціональні недоліки, а й проблеми в графічному інтерфейсі та досвіді використання;

– незалежність від коду – для виконання ручного тестування не потрібні додаткові знання, а у випадку розробки автоматизованих тестів немає прив’язки до оригінальної архітектури додатку або ж певного набору інструментів, тому QA-команда має більш ширший вибір технологічного стеку;

– перевірка вимог та документації – без доступу до коду сценарії для тестування розробляються на основі попередньо визначених вимог, специфікацій та інших документів, що дозволяє додатково їх перевірити та провести тестування документації.

З іншого боку, метод «чорного ящика» не враховує внутрішню структуру ПЗ та може погано підходити для виявлення низькорівневих дефектів [45]. Залежність від документації може стати проблемою для комплексних програмних продуктів, де підтримка вимог та специфікацій в актуальному стані має другорядний пріоритет – це може напряму вплинути на якість тестування. Крім того, метод «чорного ящика» погано підходить для тестування безпеки, адже для розробки ефективних сценаріїв необхідні хоча б базові знання про використані технології та рішення.

Метод «білого ящика» краще підходить для виявлення проблем інтеграції, безпеки та ефективності роботи, адже зі знанням про внутрішню побудову системи можливо створити такі сценарії тестування, що будуть найкращим чином перевіряти той чи інший компонент ПЗ. З іншого боку, цей метод вимагає більшої експертизи та досвіду від QA-спеціалістів, а сама процедура тестування може вимагати більшої кількості ресурсів та часу [45].

Метод «сірого ящика» – це гібрид двох попередніх підходів. Поєднання принципів «чорного ящика» та «білого ящика» при розробці плану тестування або ж при проектуванні окремих сценаріїв дозволяє краще покрити більшість аспектів створюваного ПЗ. Застосування цього методу на великих продуктах може мати аналогічні до «білого ящика» недоліки, а саме підвищені вимоги до спеціалістів та ресурсоємність, але це менш властиво невеликим програмним рішенням, як-от розроблюване ПЗ «Detrint».

Таким чином, в даному підрозділі було розглянуто базові поняття з тестування програмного забезпечення. Наведено мету тестування та розглянуто переваги його застосування. Описано популярні шляхи класифікації тестування ПЗ, а також більш детально розглянуто окремі методи тестування. Для перевірки працездатності створюваного продукту обрано метод «сірого ящика».

4.2 Тестування розробленого програмного продукту

При тестуванні розробленого продукту необхідно перевірити базовий функціонал ПЗ, а саме управління конфігураціями – це приведення системи в бажаний стан відповідно до вхідних даних користувача. Також варто протестувати працездатність всіх трьох розроблених методів.

На основі обраного для тестування методу «сірого ящика» та списку основних функцій ПЗ було сформовано список сценаріїв тестування:

1) перевірити функціонал запуску конфігураційних скриптів;

2) перевірити функціонал режиму високої доступності;

3) перевірити функціонал автоматичної інсталяції агентів;

4) перевірити функціонал автоматичного створення серверів за допомогою хмарного провайдера.

Для кожного зі сценаріїв тестування було розроблено один або декілька тестових випадків (тест-кейсів). Тестовий випадок (англ. test case) – це детальний набір кроків, вхідних даних, очікуваних результатів тощо, що необхідні для перевірки працездатності програмного забезпечення, компонента або окремої функції. Розглянемо більш детально розроблені тестові випадки.

Для перевірки функціоналу режиму високої доступності розроблено два тестових випадки. Тест-кейс №1 «Додавання нового вузла в систему»:

1) запустити застосунок на 1 або більше серверах з застосуванням конфігураційного файлу за-замовчуванням;

2) відкрити в браузері WebUI додатку, скориставшись IP-адресою будь-якого з серверів та портом за-замовчуванням (зазначений в конфігураційному файлі);

3) переключитись на вкладку «Редактор кластера»;

4) натиснути кнопку додавання нового вузла в кластер та вибрати пункт підключення існуючого вузла в кластер;

5) ввести адресу та порт будь-якого іншого сервера з запущеним ПЗ;

6) повторити кроки 4-5 з даними всіх інших серверів з розгорнутим програмним додатком;

7) перевірити наявність нових серверів в меню зі списком вузлів-членів кластера;

8) переключитись на вкладку «Переглядач логів»;

9) перевірити наявність логів до всіх доданих серверів, а також відсутність помилок.

Очікуваний результат: меню зі списком вузлів містить всі додані сервери, вкладка з логами містить повідомлення від всіх серверів-вузлів.

Результати виконання тест-кейсу №1 наведено на рисунку 4.1.

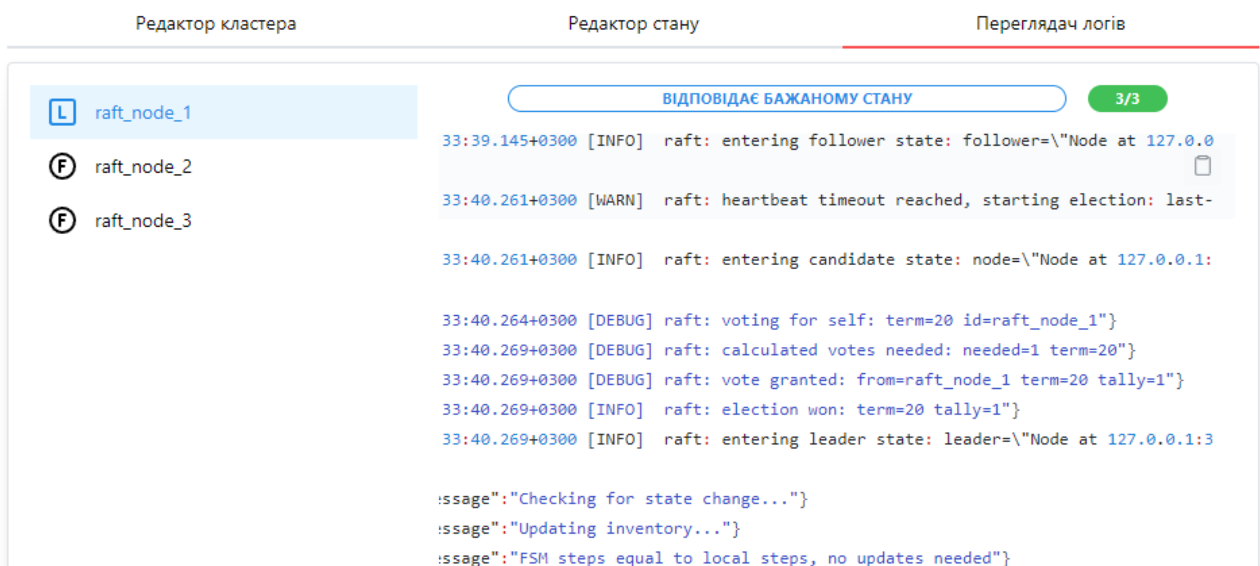


Рисунок 4.1 – Результати виконання тест-кейсу №1

Тест-кейс №2 «Аварійна зміна лідера кластеру»:

1) підготувати систему до використання відповідно до кроків 1-6 тест-кейсу №1;

2) переключитись на вкладку «Редактор кластера» та перевірити який сервер є поточним лідером кластеру;

3) на сервері, що відповідає поточному лідеру, завершити виконання додатку будь-яким нештатним шляхом (примусове завершення процесу, вимкнення або перезавантаження ОС тощо);

4) через 30 секунд оновити вкладку «Редактор кластера» та перевірити який сервер став новим лідером кластеру;

5) переключитись на вкладку «Переглядач логів»;

6) перевірити логи на наявність повідомлень про зміну лідера.

Очікуваний результат: будь-який з працюючих серверів отримає статус лідера кластеру, в логах присутні повідомлення про втрату зв'язку з поточним лідером та обрання нового.

Результати виконання тест-кейсу №2 наведено на рисунку 4.2.

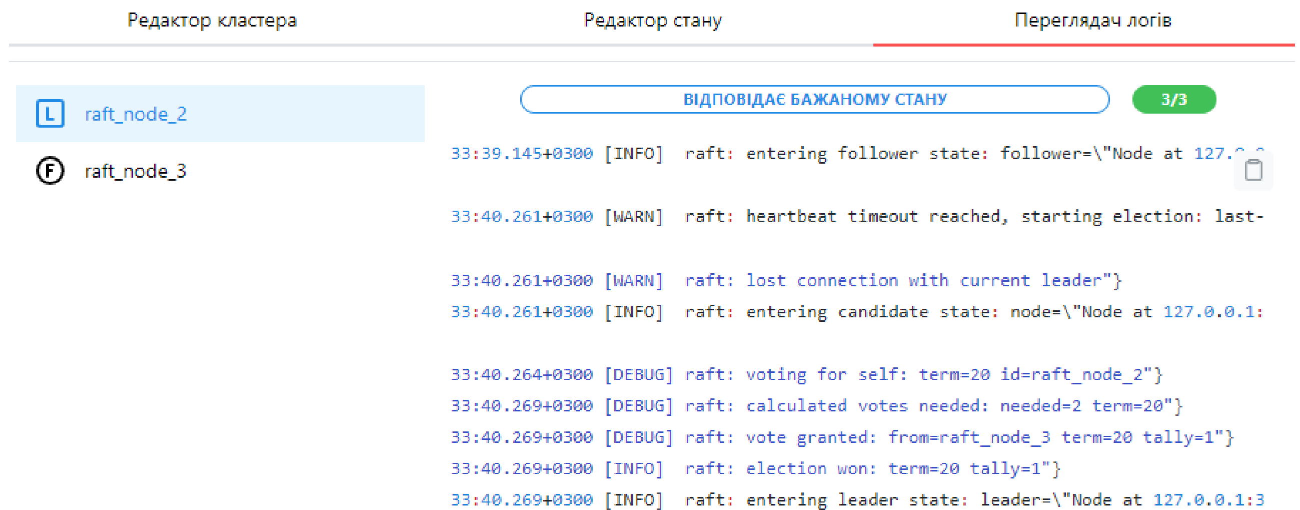


Рисунок 4.2 – Результати виконання тест-кейсу №2

Для перевірки функціоналу запуску конфігураційних скриптів було розроблено один тестовий випадок. Текст-кейс №3 «Запуск конфігураційних скриптів»:

1) підготувати систему до використання відповідно до кроків 1-6 тест-кейсу №1;

2) переключитись на вкладку «Редактор стану»;

3) натиснути кнопку редагування стану системи та завантажити попередньо підготовлений YAML-файл з новим бажаним станом системи;

4) перейти на вкладку «Переглядач логів»;

5) перевірити логи всіх серверів системи на відсутність помилок.

Очікуваний результат: логи містять відомості про виконання конфігураційних скриптів, для кожного сервера присутнє повідомлення про завершення приведення системи в бажаний стан.

Результати виконання тест-кейсу №3 наведено на рисунку 4.3.

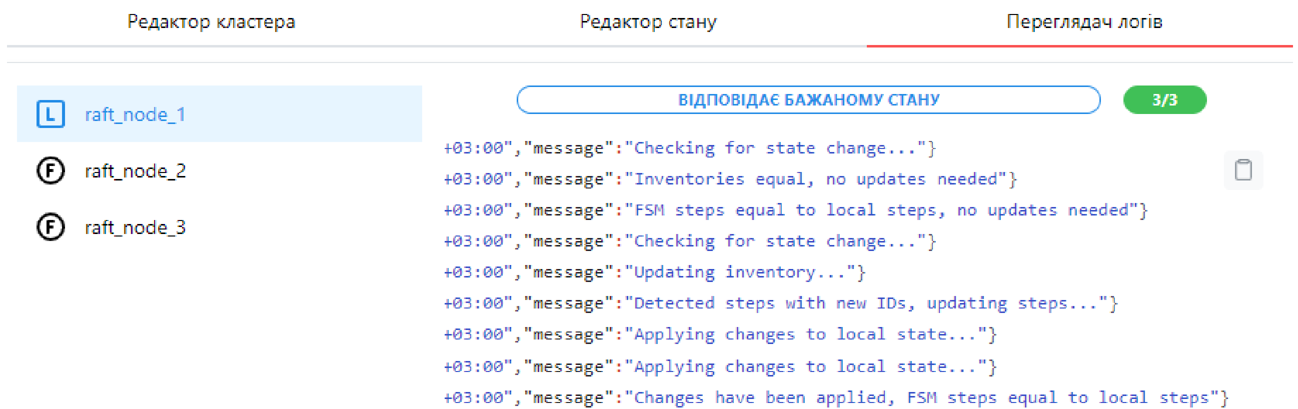


Рисунок 4.3 – Результати виконання тест-кейсу №3

Для перевірки функціоналу автоматичної інсталяції програм-агентів було розроблено один тестовий випадок. Тест-кейс №4 «Автоматична інсталяція агентів»:

1) підготувати систему до використання відповідно до кроків 1-6 тест-кейсу №1;

2) переключитись на вкладку «Редактор кластера»;

3) натиснути кнопку додавання нового вузла в кластер та вибрати пункт запуску вузла на існуючому сервері;

4) завантажити попередньо підготовлений YAML-файл з даними для підключення до сервера;

5) через 90 секунд оновити вкладку «Редактор кластера» та перевірити наявність нового вузла в списку серверів;

6) переключитись на вкладку «Переглядач логів»;

7) перевірити логи на наявність повідомлень про підключення нового вузла.

Очікуваний результат: новий сервер додано до кластеру в якості вузла-послідовника; логи містять інформаційні повідомлення про підключення нового вузла та реплікацію даних на нього.

Результати виконання тест-кейсу №4 наведено на рисунку 4.4.

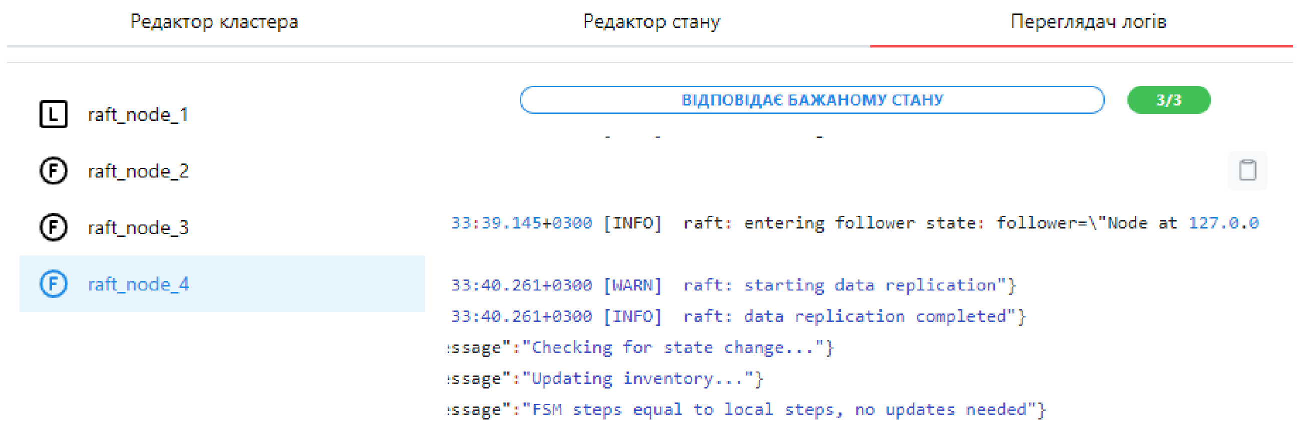


Рисунок 4.4 – Результати виконання тест-кейсу №4

Для перевірки функціоналу автоматичного створення серверів за допомогою хмарних провайдерів було підготовлено один тестовий випадок. Тест-кейс №5 «Створення сервера в AWS»:

1) підготувати систему до використання відповідно до кроків 1-6 тест-кейсу №1;

2) надати серверам системи доступ до AWS шляхом передачі даних для авторизації через зміну змінних середовища, копіювання файлів налаштувань або надання IAM-ролі серверу;

3) переключитись на вкладку «Редактор кластера»;

4) натиснути кнопку додавання нового вузла в кластер та вибрати пункт розгортання нових серверів;

5) завантажити попередньо-підготовлені YAML-файли конфігурації мережі та серверів для AWS;

6) переключитись на вкладку «Переглядач логів»;

7) перевірити логи на наявність інформаційних повідомлень про процес підготовки серверів та відсутність помилок;

8) після отримання повідомлень про розгортання всіх машин в інфраструктурі хмарного провайдера переключитись на вкладку «Редактор кластера»;

9) виконати кроки 3-7 тест-кейсу №4;

10) перевірити логи на наявність повідомлень від нових вузлів.

Очікуваний результат: логи містять інформаційні повідомлення про розгортання нових серверів та запуск агентів на них, нові сервери відображають на вкладці «Редактор кластера» в якості вузлів системи.

Результати виконання тест-кейсу №5 наведено на рисунку 4.5.

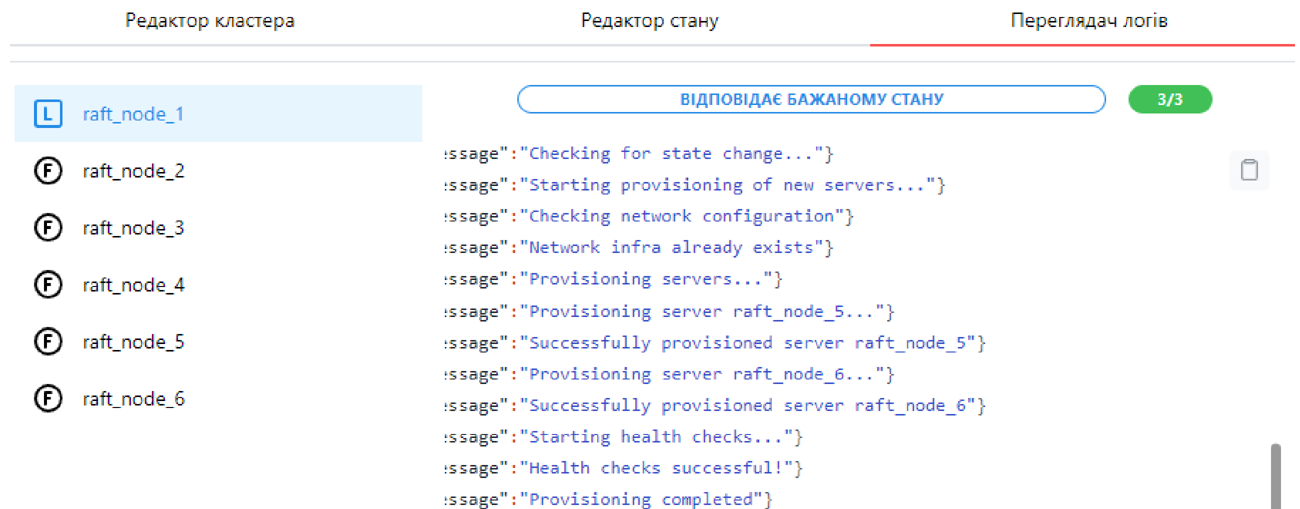


Рисунок 4.5 – Результати виконання тест-кейсу №5

Таким чином, було проведено тестування програмного забезпечення управління конфігураціями. За допомогою методу «сірого ящика» було розроблено ряд тестових випадків для перевірки працездатності основного функціоналу програмного продукту, включаючи реалізацію запропонованих покращених методів. Тестування не виявило проблем в основному функціоналі ПЗ, виконання усіх тест-кейсів було успішне.

4.3 Дослідження ефективності розробленого методу забезпечення високої доступності

Одна з властивостей високої доступності – це забезпечення безвідмовної роботи в аварійних ситуаціях, тому варто порівняти розроблений метод з іншими стратегіями аварійного відновлення. RTO визначає час на відновлення працездатності системи, а RPO – об’єм даних, який неможливо відновити через особливості тої чи іншої стратегії відновлення. Іншими словами, RTO та RPO визначаються час простою та втрати даних відповідно.

Конкретні значення цих показників сильно залежать від цифрової інфраструктури продукту, а також від масштабів розгорнутої системи. Крім того, кількість ресурсів та часу, що необхідні для реалізації тієї чи іншої стратегії відновлення, також відрізняються. Тому, в відкритому доступі складно знайти усереднені статистичні показники RTO/RPO – вони індивідуальні й залежать від конкретного продукту та готовності бізнесу інвестувати в аварійне відновлення.

Проте, є чіткі співвідношення між стратегіями відновлення, які дозволяють оцінити їх показники відносно одне одного. Найбільш популярними стратегіями аварійного відновлення є: «Backup and restore», «Pilot light», «Warm standby», «Multi-site active/active».

Розроблений вдосконалений метод забезпечення високої доступності по своїм характеристикам подібний до стратегій «Pilot light» та «Warm standby». Це дві схожі стратегії, що відрізняються кількістю попередньо розгорнутих ресурсів. «Pilot light» передбачає дублювання лише самої важливої інфраструктури, для якої регулярно проводиться реплікація даних, а решта вузлів або сервісів розгортаються лише у випадку аварійної ситуації. «Warm standby» передбачає повне дублювання системи та необхідної інфраструктури, але в зменшених масштабах – ця стратегія добре підходить для систем, що підтримують автоматичне масштабування. У випадку аварійної ситуації додаткові ресурси можуть бути створені відповідно до нового клієнтського навантаження. Ці дві стратегії мають схожі показники та зазвичай обходяться дорожче за «Backup and restore», але й показують кращі показники RTO та RPO – в середньому це десятки хвилин для «Pilot light» та лічені хвилини для «Warm standby».

Для використання запропонованого методу також необхідні додаткові сервери-вузли, які вже в залежності від призначення системи можуть або виконувати певну корисну роботу, або ж просто реплікувати критично важливі дані та очікувати. Тому, вартість застосування запропонованого методу в якості стратегії аварійного відновлення може бути близькою до застосування «Pilot light» або «Warm standby», але показники RTO та RPO повинні бути кращими.

Реплікація даних відбувається в момент обробки нового запиту на оновлення стану кластеру – лідер виконує зміни в своїй власній копії даних, а потім розсилає ці оновлення на інші вузли. Тільки після підтвердження про успішну реплікацію від більшості вузлів лідер зберігає всі ці змінив в якості постійних. Втрата даних може виникнути лише під час обробки нового запиту, коли лідер почав працювати з новими даними, але потрапив в аварійну ситуацію до закінчення реплікації даних. Відповідно, показник RPO в такому сценарії повинен становити не більше декількох секунд для малих об’ємів даних (наприклад, коли в загальному стані є лише базова інформація для підтримки системи в робочому стані) та не більше декількох хвилин для великих об’ємів (наприклад, коли реплікований стан використовують в якості бази даних).

Показник RTO не повинен перевищувати декількох десятків секунд – вузли регулярно комунікують з лідером для перевірки наявності нової інформації, тому виявлення непрацюючого лідера та його заміна на інший вузол відбувається швидко. Так як останні дані про стан системи регулярно реплікуються, то більшість вузлів можуть стати потенційними кандидатами для нового лідера – процес переключення включає в себе лише голосування за конкретний вузол та розсилання оновлення з даними нового лідера.

Для більш об’єктивної оцінки розробленого методу було застосовано апріорне ранжування. Було сформовано ряд факторів, що присутні в розглянутих стратегіях аварійного відновлення. Група з 5-ти експертів, що мають досвід в сфері DevOps, провели ранжування цих факторів. За отриманими даними можна визначити які показники є найбільш значущими та перевірити наскільки вони реалізовані в запропонованій розробці, щоб зрозуміти її актуальність.

Було обрано наступні фактори для ранжування:

– складність реалізації та підтримки;

– час простою в аварійній ситуації;

– частота реплікації даних або створення резервних копій;

– можливість додаткового використання резервних вузлів;

– автоматизація процесу відновлення.

Оцінки експертів та результати розрахунків наведено в таблиці 4.1. Після виставлення оцінок було обчислено суму рангів для кожного фактору за формулою:

$$\Delta_k = \sum_{m=1}^{m} a_{km}, \tag{4.1}$$

де m – кількість експертів (у випадку даного дослідження – це 5 осіб), а k – кількість факторів, що були використані при ранжуванні (в даному випадку – 5).

Приклад розрахунку для першого фактору:

$$\Delta_1 = 2 + 5 + 5 + 4 + 4 = 20.$$

Після цього було розраховано загальну суму рангів та середнє значення суми:

$$\overline{\Delta} = \frac{\sum_{k=1}^{k} \Delta_k}{k} = 20 + 9 + 12 + 22 + 12 = \frac{75}{5} = 15. \tag{4.2}$$

Далі розраховано відхилення суми рангів від середньої суми рангів для кожного фактору:

$$\Delta_k^{\sigma} = \Delta_k - \overline{\Delta}. \tag{4.3}$$

Приклад розрахунку для першого фактору:

$$\Delta_1^{\sigma} = 20 - 15 = 5.$$

Ці дані дозволяють розрахувати коефіцієнт згоди між експертами – занадто мале значення означає, що обрана група має занадто різнобічні погляди й не підходить для ранжування факторів. Спочатку було знайдено суму квадратів відхилення:

$$\mathrm{S} = \sum_{k=1}^{k} \left(\Delta_k^{\sigma}\right)^2 = 25 + 36 + 9 + 49 + 9 = 128. \tag{4.4}$$

І з використанням отриманої суми розраховано коефіцієнт узгодженості:

$$\mathrm{W} = \frac{12\mathrm{S}}{m^2\left(k^3 - k\right)} = \frac{12 \cdot 128}{5^2\left(5^3 - 5\right)} = 0{,}512. \tag{4.5}$$

Якщо значення коефіцієнту складає більше, ніж 0,5, як вийшло при розрахунках з даною командою та набором факторів, то можна вважати, що в оцінках експертів є певний рівень згоди.

Крім того, отримані дані можна використати для розрахунку критерія узгодженості Пірсона. Він застосовується для перевірки, чи отримана гіпотеза про ранжування факторів є випадковим співпадінням думок експертів, чи вона має певне підґрунтя. Може виникнути ситуація, коли оцінки експертів підпадають під закон розподілу ймовірностей і в такому випадку неможна довіряти отриманому значенню коефіцієнта узгодженості.

Для обчислення використано наступну формулу:

$$\chi^2 = \mathrm{W} \cdot m(k-1) = 0{,}512 \cdot 5(5-1) = 10{,}24\,, \tag{4.6}$$

де $(k-1)$ визначає число ступенів свободи. Табличне значення критерію Пірсона за умови наявності чотирьох ступенів свободи та рівня значимості 0,05 становить 9,5 [46], що менше отриманого значення. Це дозволяє прийняти гіпотезу про те, що співпадіння думок експертів не є випадковим на заданому рівні значимості.

Отримані значення Δ_k дозволяють провести ранжування факторів. Місця розподіляються по мірі зростання суми рангів, тобто перше місце відповідає найменшій сумі.

На основі місць можна розрахувати питому вагу кожного фактору – це дозволяє порівняти фактори відносно одне одного й визначити їх значущість на думку експертів. Розрахунок питомої ваги виконано за допомогою наступної формули:

$$q_k = \frac{2(k - \mathrm{M} + 1)}{k(k+1)}, \tag{4.7}$$

де M – це місце, що отримав фактор на основі ранжування.

Отримані дані дозволяють побудувати апріорну діаграму рангів, яка візуально показує найбільш важливі фактори з точки зору експертів при виборі методу або стратегії для забезпечення високої доступності. Фактори, чия сума рангів менша за середню, можна вважати більш важливими за решту.

Всі результати розрахунків наведено в таблиці 4.1.

Таблиця 4.1 – Вхідні дані та результати апріорного ранжування

Номер	Фактор	Експерти					Δ_k	Δ_k^σ	$\left(\Delta_k^\sigma\right)^2$	M	q_k
		1	2	3	4	5					
1	Складність реалізації та підтримки	2	5	5	4	4	20	5	25	4	0,13
2	Час простою в аварійній ситуації	3	2	1	1	2	9	-6	36	1	0,33
3	Частота реплікації даних	1	3	2	3	3	12	-3	9	2	0,27
4	Додаткове використання вузлів	5	4	3	5	5	22	7	49	5	0,07
5	Автоматизація процесу відновлення	4	1	4	2	1	12	-3	9	3	0,2
Сума		15	15	15	15	15	75	-	128	-	1

Отримана діаграма наведена на рисунку 4.6.

Згідно діаграми, найбільш важливими факторами при виборі методу або стратегії аварійного відновлення є час простою в аварійній ситуації, частота реплікації даних та автоматизація процесу відновлення. Їх сумарна питома вага становить 0,8.

Отримане ранжування дозволяє оцінити наскільки актуальним є запропонований метод забезпечення високої доступності відповідно до вимог і оцінок експертів. Розроблений метод має хороші показники часу простою та частоти реплікації даних – в залежності від ситуації ці параметри можуть співпадати або випереджати параметри, що показують стратегії «Pilot light» та «Warm standby».

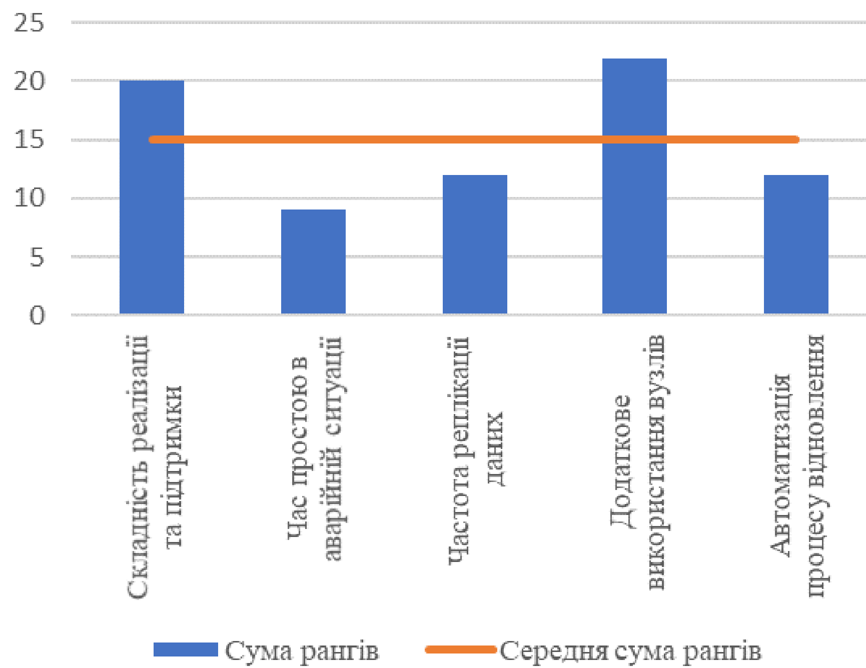


Рисунок 4.6 – Апріорна діаграма рангів

Процес відновлення працездатності системи у розробленому методі повністю автономний – вузли самостійно перевіряють можливість з'єднання з поточним лідером та ініціюють його зміну в випадку виникнення проблем. Крім того, метод надає можливість використовувати вузли-послідовники для самого процесу управління конфігураціями, а не просто використовує їх в якості пасивних реплік, тому сервери мають можливість виконувати додаткове корисне навантаження. Проте, цей фактор експерти оцінили в якості найменш важливого. Складність реалізації та підтримки не розглядається по аналогічній причині – цей фактор посідає четверте місце в ранжуванні і його оцінка може сильно варіюватися в залежності від варіантів застосування того чи іншого методу або стратегії для реальної системи.

Апріорне ранжування показало, що метод реалізує найбільш важливі за оцінками експертів фактори, а по показникам RTO та RPO він показує себе краще за деякі інші популярні стратегії аварійного відновлення. Розроблений метод забезпечення високої доступності для програмного забезпечення управління конфігураціями є актуальним та доцільним для використання.

4.4 Розробка інструкції користувача

«Detrint» – це програмне забезпечення для управління конфігураціями з режимом високої доступності та функціями автоматичної інсталяції програм-агентів і створення нових серверів в хмарі.

На відміну від класичного ПЗ для управління конфігураціями «Detrint» має лише один виконуваний файл, адже кожен вузол системи може виступати і в якості лідера, і в якості послідовника. Крім того, разом з виконуваним файлом поставляється YAML-файл з конфігурацією за замовчуванням.

Приклад конфігурації наведено на рисунку 4.7.

```
1  node:
2    grpc_address: 127.0.0.1:3000
3    web_address: 127.0.0.1:3001
4    local_id: raft_node_1
5    data_dir: ./data/
6    logs_file: logs.dat
7    stable_file: stable.dat
8    local_file: local.json
9    bootstrap: false
```

Рисунок 4.7 – Приклад конфігураційного файлу

Файл містить налаштування локального сховища даних, gRPC та Web серверів, а також параметри самого вузла. В більшості випадків необхідно працювати лише з трьома параметрами: «grpc_address», «web_address» та «bootstrap». Перші два визначають адрес та порт для запуску відповідно gRPC та Web серверів. За потреби ці параметри необхідно змінити в відповідності до мережевої конфігурації кінцевого сервера, на якому буде використовуватись ПЗ.

Параметр «bootstrap» визначає початковий стан вузла – при значенні «false» сервер відразу перейде в режим послідовника або ж спробує приєднатися до попередньо відомого кластера (за умови, що цей вузол уже працював до цього моменту в складі кластера). При значенні «true» вузол запустить процедуру ініціації кластера – такий сервер можна використовувати як самостійну машину

(кластер з лише одним вузлом) або відразу використовувати для підключення вузлів-послідовників.

При запуску виконуючого файлу ПЗ автоматично буде шукати файл «config.yaml» з конфігурацією в поточній директорії. Шлях та назву файлу можливо перезаписати за допомогою параметра «-config».

Після запуску ПЗ користувач може відкрити WebUI системи – для цього необхідно в браузері перейти по комбінації адреси та порту, що зазначені в конфігураційному файлі в параметрі «web_address». Якщо графічний інтерфейс не буде доступний або виникнуть проблеми в його роботі – необхідно перевірити логи виконання ПЗ на наявність повідомлень про потенційні проблеми. При запуску виконуваного файлу програмне забезпечення продовжує працювати в форматі інтерфейсу командного рядка, в цьому вікні можливо побачити всі логи поточного вузла, включаючи завантажену конфігурацію та всі процедури запуску.

Графічний інтерфейс користувача поділено на три основні зони-вкладки, а саме: «Редактор кластера», «Редактор стану» та «Переглядач логів». Перша вкладка «Редактор кластера» призначена для роботи з вузлами системи.

Приклад інтерфейсу вкладки «Редактор кластера» наведено на рисунку 4.8.

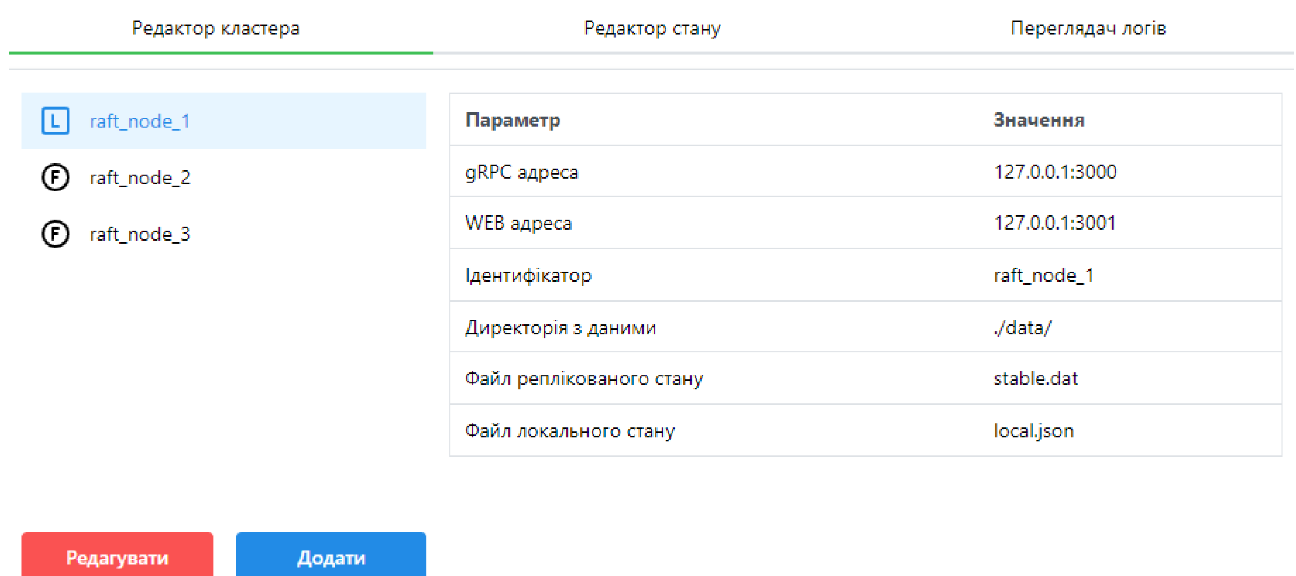


Рисунок 4.8 – Приклад інтерфейсу вкладки «Редактор кластера»

Вона дозволяє перевірити стан та роль поточного вузла, а також містить кнопки для додавання нових вузлів і редагування даних уже приєднаних в кластер серверів. При додаванні нових серверів у користувача є вибір:

– приєднати інший сервер з уже запущеним екземпляром «Detrint», щоб зробити його частиною поточного кластера;

– автоматично розгорнути на іншому сервері екземпляр «Detrint» та приєднати його до кластера;

– розгорнути за допомогою хмарного провайдера нові сервери для роботи системи.

Вкладка «Редактор стану» призначена для роботи з основною функцією програмного забезпечення – управління конфігураціями. Вона дає можливість переглянути загальний список вузлів системи та перевірити стан кожного з них.

Приклад інтерфейсу вкладки «Редактор стану» наведено на рисунку 4.9.

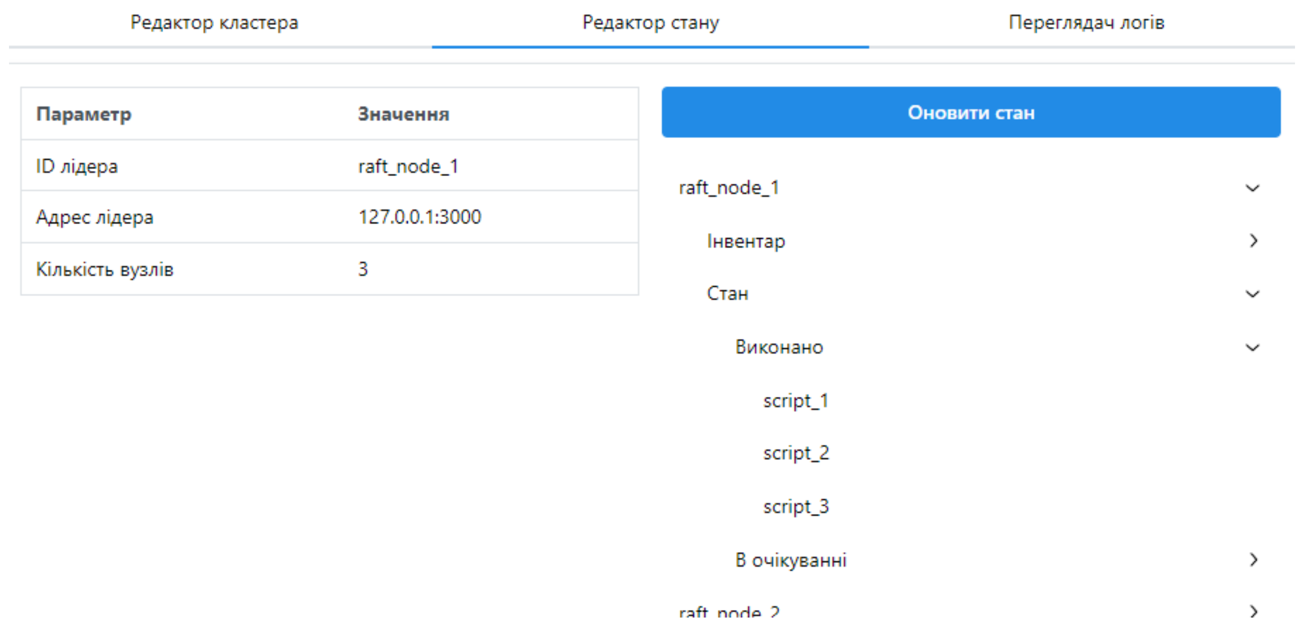


Рисунок 4.9 – Приклад інтерфейсу вкладки «Редактор стану»

Локальний стан вузла – це сукупність даних про те, які скрипти автоматизації виконувались чи не виконувались на сервері. Саме скрипти автоматизації і змінюють систему для досягнення бажаного стану. Наприклад, вони можуть виконуватись для інсталяції додаткового програмного

забезпечення, запуску інших інструментів, зміни параметрів ОС тощо. За допомогою кнопки редагування стану системи можливо ввести новий бажаний стан. Ці дані буде репліковано на всі вузли кластера та почнеться процедура порівняння нового бажаного стану з локальним станом кожного з вузлів – при знаходженні відмінностей буде запущено ті чи інші скрипти для внесення необхідних змін на сервері.

Вкладка «Переглядач логів» дозволяє більш детально ознайомитись з діями, які виконує кожен вузол. Приклад інтерфейсу вкладки «Переглядач логів» наведено на рисунку 4.10.

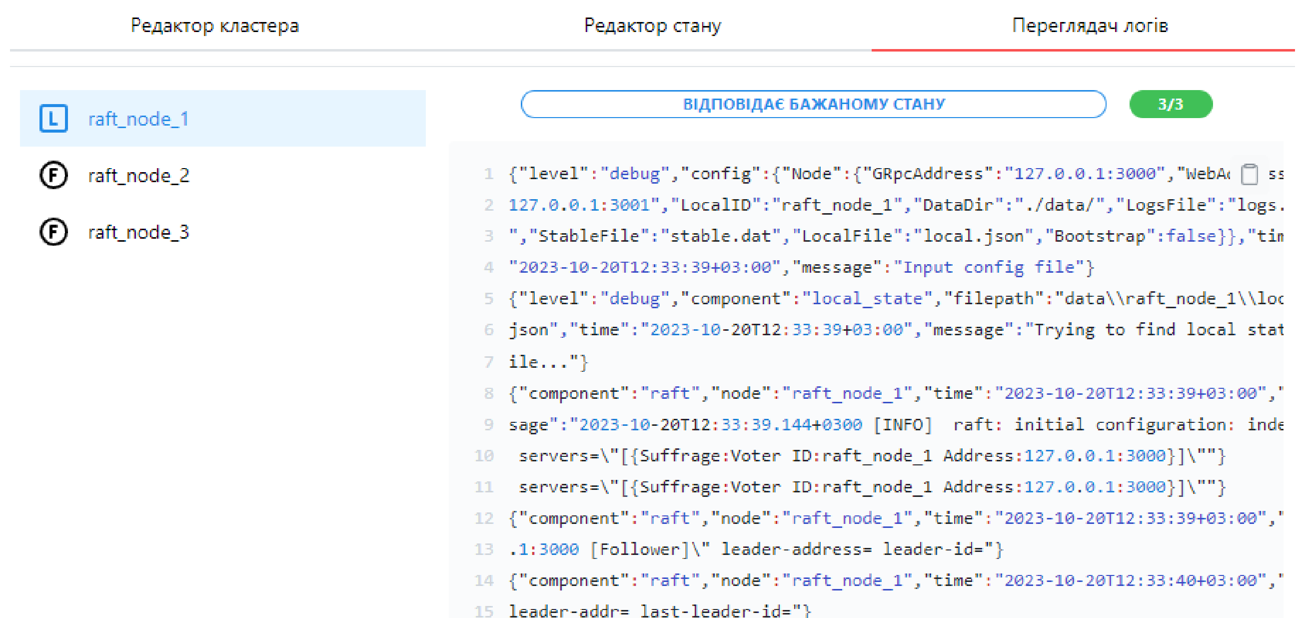


Рисунок 4.10 – Приклад інтерфейсу вкладки «Переглядач логів»

Меню в лівій частині інтерфейсу дозволяє обрати конкретний сервер, а зона в правій частині відповідає за відображення повідомлень цього вузла. Користувач може переглянути як повідомлення, що відповідають за сам процес управління конфігураціями й приведення системи в бажаний стан, так і службові логи, що стосуються загального функціонування вузла. Також присутні два додаткових індикатори в верхній частині вкладки – вони містять загальну «оцінку» стану вузла, щоб користувач міг швидко зорієнтуватися в його стані без детального дослідження логів.

Файли з вхідними даними, які користувач завантажує при зміні стану системи або ж при редагуванні кластера, мають бути в форматі YAML. Це простий формат для зберігання структур даних, що добре підходить для нетехнічних користувачів через простоту синтаксису. Приклад вхідних файлів наведено на рисунку 4.11.

```yaml
username: Administrator
address: Pa88w0rd
remote_path: D:/detrint/
authentication: basic
type: SSH
retry_count: 3
```

```yaml
- node_id: raft_node_2
  node_address: 192.168.0.24:3000
  inventory:
    - new_var: 'value'
    - new_var_2: 524352
- node_id: raft_node_3
  node_address: 192.168.0.25:3000
  steps:
    - id: ab2
      name: script_a
    - id: ab3
      name: script_b
```

Рисунок 4.11 – Приклад файлу для автоматичної інсталяції агента (зліва) та для зміни стану системи (справа)

Для зміни стану системи використовуються скрипти, що вже вбудовані в програмне забезпечення. Для їх використання необхідно лише зазначити назву потрібного скрипту в YAML файлі. Крім того, кінцевий користувач може використовувати свої власні скрипти на мові Python, що будуть виконані під час зміни стану системи за допомогою вбудованого в «Detrint» інтерпретатор. В якості вхідних даних використовують змінні, що об'являються в секції «inventory» документу.

Таким чином, було розроблено інструкцію користувача. Описано процес запуску програмного забезпечення, наведено приклад конфігураційного файлу, розглянуто його найбільш важливі параметри. Детально описано всі елементи інтерфейсу продукту, а також їх призначення. Наведено приклади вхідних файлів для різних функцій застосунку.

4.5 Системні вимоги

Застосунок розповсюджується в складі виконуваного файлу для відповідної операційної системи та набору файлів-прикладів у форматі YAML, як-от конфігураційний файл вузла, файл зміни стану, розширення кластера тощо.

Мінімальні та рекомендовані системні вимоги до апаратного та програмного забезпечення наведено в таблиці 4.2.

Таблиця 4.2 – Системні вимоги для «Detrint»

Параметр	Мінімальна конфігурація	Рекомендована конфігурація
CPU	Тактова частота 1,6 ГГц або більше, 2 ядра або більше, архітектура x86-64	Тактова частота 2,4 ГГц або більше, 4 ядра або більше, архітектура x86-64
RAM	512 МБ для дистрибутивів Linux; 4 ГБ для ОС Windows або MacOS	2 ГБ для дистрибутивів Linux; 8 ГБ для ОС Windows або MacOS
GPU	Інтегрований графічний пристрій	Інтегрований графічний пристрій
Накопичувач	HDD, 128 МБ вільного місця	SSD, 512 МБ вільного місця
ОС	Windows 7 або дистрибутив з ядром Linux 2.6.23 або macOS High Sierra 10.13	Windows 10 або дистрибутив з ядром Linux 5.17.6 або macOS Monterey 12

Таким чином, було розглянуто системні вимоги для використання програмного продукту «Detrint». Наведено мінімальну та рекомендовану конфігурацію апаратного та програмного забезпечення для роботи із застосунком.

4.6 Висновки

В даному підрозділі було розглянуто базові відомості про тестування програмного забезпечення. Для перевірки працездатності створюваного продукту було обрано метод «сірого ящика», розроблено набір з п’яти тестових випадків. Проведено тестування ПЗ, яке не виявило дефектів в основних функціях «Detrint». Розроблено інструкцію користувача, сформовано мінімальні та рекомендовані системні вимоги для використання застосунку.

5 ЕКОНОМІЧНА ЧАСТИНА

5.1 Проведення комерційного та технологічного аудиту науково-технічної розробки

Метою проведення комерційного і технологічного аудиту дослідження за темою «Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів» є оцінювання науково-технічного рівня та рівня комерційного потенціалу розробки, створеної в результаті науково-технічної діяльності.

Оцінювання науково-технічного рівня розробки та її комерційного потенціалу рекомендується здійснювати із застосуванням 5-ти бальної системи оцінювання за 12-ма критеріями, наведеними в табл. 5.1 [47].

Таблиця 5.1 – Рекомендовані критерії оцінювання науково-технічного рівня і комерційного потенціалу розробки та бальна оцінка

Бали (за 5-ти бальною шкалою)					
	0	1	2	3	4
Технічна здійсненність концепції					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено працездатність продукту в реальних умовах
Ринкові переваги (недоліки)					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів
4	Технічні та споживчі властивості продукту значно гірші, ніж в	Технічні та споживчі влас-тивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в	Технічні та споживчі властивості продукту значно кращі, ніж в
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів

Продовження таблиці 5.1

Бали (за 5-ти бальною шкалою)					
	0	1	2	3	4
Ринкові перспективи					
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практична здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використ. у виробництві
11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

Результати оцінювання науково-технічного рівня та комерційного потенціалу науково-технічної розробки потрібно звести, результати наведено в таблиці 5.2.

Таблиця 5.2 – Результати оцінювання науково-технічного рівня і комерційного потенціалу розробки експертами

Критерії	Експерт (ПІБ, посада)		
	1	2	3
	Бали:		
1. Технічна здійсненність концепції	4	4	4
2. Ринкові переваги (наявність аналогів)	4	4	4
3. Ринкові переваги (ціна продукту)	3	3	3
4. Ринкові переваги (технічні властивості)	4	3	3
5. Ринкові переваги (експлуатаційні витрати)	3	3	3
6. Ринкові перспективи (розмір ринку)	3	4	3
7. Ринкові перспективи (конкуренція)	3	3	4
8. Практична здійсненність (наявність фахівців)	3	3	3
9. Практична здійсненність (наявність фінансів)	3	3	3
10. Практична здійсненність (необхідність нових матеріалів)	3	3	3
11. Практична здійсненність (термін реалізації)	3	4	4
12. Практична здійсненність (розробка документів)	4	3	3
Сума балів	40	40	40
Середньоарифметична сума балів $СБ_с$	40		

За результатами розрахунків, наведених в таблиці 5.2, зробимо висновок щодо науково-технічного рівня і рівня комерційного потенціалу розробки. При цьому використаємо рекомендації, наведені в табл. 5.3 [47].

Таблиця 5.3 – Науково-технічні рівні та комерційні потенціали розробки

Середньоарифметична сума балів $СБ_с$, розрахована на основі висновків експертів	Науково-технічний рівень та комерційний потенціал розробки
41…48	Високий
31…40	Вище середнього
21…30	Середній
11…20	Нижче середнього
0…10	Низький

Згідно проведених досліджень рівень комерційного потенціалу розробки за темою «Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів» становить 40 балів, що, відповідно до таблиці 5.3, свідчить про комерційну важливість проведення даних досліджень (рівень комерційного потенціалу розробки вищий середнього).

5.2 Розрахунок витрат на проведення науково-дослідної роботи

Витрати, пов’язані з проведенням науково-дослідної роботи групуються за наступними статтями: витрати на оплату праці; відрахування на соціальні заходи; сировина та матеріали; розрахунок витрат на комплектуючі; спецустаткування для наукових (експериментальних) робіт; програмне забезпечення для наукових (експериментальних) робіт; амортизація обладнання, програмних засобів та приміщень; паливо та енергія для науково-виробничих цілей; службові відрядження; витрати на роботи, які виконують сторонні підприємства, установи і організації; інші витрати; накладні (загальновиробничі) витрати.

До статті «Витрати на оплату праці» належать витрати на виплату основної та додаткової заробітної плати керівникам відділів, лабораторій, секторів і груп, науковим, інженерно-технічним працівникам, конструкторам, технологам, креслярам, копіювальникам, лаборантам, робітникам, студентам, аспірантам та іншим працівникам, безпосередньо зайнятим виконанням конкретної теми, обчисленої за посадовими окладами, відрядними розцінками, тарифними ставками згідно з чинними в організаціях системами оплати праці.

Витрати на основну заробітну плату дослідників ($З_о$) розраховуємо у відповідності до посадових окладів працівників, за формулою [47]:

$$З_o = \sum_{i=1}^{k} \frac{M_{ni} \cdot t_i}{T_p}, \qquad (5.1)$$

де k – кількість посад дослідників залучених до процесу досліджень;

M_{ni} – місячний посадовий оклад конкретного дослідника, грн;

t_i – число днів роботи конкретного дослідника, дні;

T_p – середнє число робочих днів в місяці, T_p=21 день.

$$З_о = 20000,00 \cdot 63 / 21 = 60000 \text{ грн.}$$

Проведені розрахунки зведемо до таблиці 5.4.

Таблиця 5.4 – Витрати на заробітну плату дослідників

Найменування посади	Місячний посадовий оклад, грн	Оплата за робочий день, грн	Число днів роботи	Витрати на заробітну плату, грн
Керівник проекту	20000	952,38	63	60000,00
Інженер-розробник програмного забезпечення	18000	857,14	58	49714,29
UI/UX-дизайнер	15000	714,29	15	10714,29
Всього				120428,57

Витрати на основну заробітну плату робітників ($З_р$) за відповідними найменуваннями робіт НДР розраховуємо за формулою:

$$З_p = \sum_{i=1}^{n} C_i \cdot t_i \, , \qquad (5.2)$$

де C_i – погодинна тарифна ставка робітника відповідного розряду, за виконану відповідну роботу, грн/год;

t_i – час роботи робітника при виконанні визначеної роботи, год.

Погодинну тарифну ставку робітника відповідного розряду C_i можна визначити за формулою:

$$C_i = \frac{M_M \cdot K_i \cdot K_c}{T_p \cdot t_{зм}} \, , \qquad (5.3)$$

де M_M – розмір прожиткового мінімуму працездатної особи, або мінімальної місячної заробітної плати (в залежності від діючого законодавства), приймемо M_M=6700,00 грн;

K_i – коефіцієнт міжкваліфікаційного співвідношення для встановлення тарифної ставки робітнику відповідного розряду (табл. Б.2, додаток Б) [47];

K_c – мінімальний коефіцієнт співвідношень місячних тарифних ставок робітників першого розряду з нормальними умовами праці виробничих

об’єднань і підприємств до законодавчо встановленого розміру мінімальної заробітної плати.

$T_р$ – середнє число робочих днів в місяці, приблизно $T_р$ = 21 дні;

$t_{зм}$ – тривалість зміни, год.

$$С_1 = 6700,00 \cdot 1,5 \cdot 1,65 / (21 \cdot 8) = 98,71 \text{ грн.}$$

$$З_{р1} = 98,71 \cdot 6,00 = 592,23 \text{ грн.}$$

Дані наведено в таблиці 5.5.

Таблиця 5.5 – Величина витрат на основну заробітну плату робітників

Найменування робіт	Тривалість роботи, год	Розряд роботи	Тарифний коефіцієнт	Погодинна тарифна ставка, грн	Величина оплати на робітника, грн
Первинне налаштування робочих станцій	6	4	1,5	98,71	592,23
Налаштування локальної мережі для обладнання	12	4	1,5	98,71	1184,46
Налаштування і підключення серверу для середовища розробки	16	5	1,7	111,87	1789,86
Всього					3566,55

Додаткову заробітну плату розраховуємо як 10 … 12% від суми основної заробітної плати дослідників та робітників за формулою:

$$З_{дод} = (З_о + З_р) \cdot \frac{Н_{дод}}{100\%}, \qquad (5.4)$$

де $Н_{дод}$ – норма нарахування додаткової заробітної плати. Приймемо 12%.

$З_{дод}$ = (120428,57+ 3566,55) · 12 / 100% = 14879,42 грн.

Нарахування на заробітну плату дослідників та робітників розраховуємо як 22% від суми основної та додаткової заробітної плати дослідників і робітників за формулою:

$$З_н = (З_о + З_р + З_{дод}) \cdot \frac{Н_{зп}}{100\%} \qquad (5.5)$$

де $Н_{зп}$ – норма нарахування на заробітну плату. Приймаємо 22%.

Зн = (120428,57+ 3566,55+ 14879,42) · 22 / 100% = 30552,4 грн.

До статті «Сировина та матеріали» належать витрати на сировину, основні та допоміжні матеріали, інструменти, пристрої та інші засоби і предмети праці, які придбані у сторонніх підприємств, установ і організацій та витрачені на проведення досліджень.

Витрати на матеріали (*М*), у вартісному виражені розраховуються окремо по кожному виду матеріалів за формулою:

$$M = \sum_{j=1}^{n} Н_j \cdot Ц_j \cdot К_j - \sum_{j=1}^{n} В_j \cdot Ц_{вj} \ , \qquad (5.6)$$

де $Н_j$ – норма витрат матеріалу *j*-го найменування, кг;

n – кількість видів матеріалів;

$Ц_j$ – вартість матеріалу *j*-го найменування, грн/кг;

$К_j$ – коефіцієнт транспортних витрат, ($К_j = 1,1 \ ... \ 1,15$);

$В_j$ – маса відходів *j*-го найменування, кг;

$Ц_{вj}$ – вартість відходів *j*-го найменування, грн/кг.

$М_1$ = 3 · 200,00 · 1,1 - 0,000 · 0,00 = 660,0 грн.

Проведені розрахунки зведемо до таблиці 5.6.

Таблиця 5.6 – Витрати на матеріали

Найменування матеріалу, марка, тип, сорт	Ціна за од, грн	Норма витрат, од	Величина відходів, кг	Ціна відходів, грн/кг	Вартість витраченого матеріалу, грн
Офісний папір	200	3	0	0	660
Папір для записів	110	1	0	0	121
Канцелярське приладдя (набір офісного працівника)	175	2	0	0	385
Всього					1166

Витрати на комплектуючі *(Кв)*, які використовують при проведенні НДР відсутні.

До статті «Спецустаткування для наукових (експериментальних) робіт» належать витрати на виготовлення та придбання спецустаткування необхідного

для проведення досліджень, також витрати на їх проектування, виготовлення, транспортування, монтаж та встановлення.

Балансову вартість спецустаткування розраховуємо за формулою:

$$B_{спец} = \sum_{i=1}^{k} Ц_i \cdot C_{пр.i} \cdot K_i \quad , \tag{5.7}$$

де $Ц_i$ – ціна придбання одиниці спецустаткування даного виду, марки, грн;

$C_{пр.i}$ –кількість одиниць устаткування відповідного найменування, які придбані для проведення досліджень, шт.;

K_i – коефіцієнт, що враховує доставку, монтаж, налагодження устаткування тощо, ($K_i = 1,10…1,12$);

k – кількість найменувань устаткування.

$$B_{спец} = 18000 \cdot 3 \cdot 1,1 = 59400 \text{ грн.}$$

Отримані результати зведемо до таблиці 5.7.

Таблиця 5.7 – Витрати на придбання спецустаткування по кожному виду

Найменування устаткування	Кількість, шт	Ціна за одиницю, грн	Вартість, грн
Робоча станція (ПК)	3	18000	59400
Маршрутизатор	1	1700	1870
Сервер для середовища розробки та тестування	1	20000	22000
Всього			83270

До статті «Програмне забезпечення для наукових (експериментальних) робіт» належать витрати на розробку та придбання спеціальних програмних засобів і програмного забезпечення, (програм, алгоритмів, баз даних) необхідних для проведення досліджень, також витрати на їх проектування, формування та встановлення.

Балансову вартість програмного забезпечення розраховуємо за формулою:

$$B_{прг} = \sum_{i=1}^{k} Ц_{iпрг} \cdot C_{прг.i} \cdot K_i \quad , \tag{5.8}$$

де $Ц_{iпрг}$ – ціна придбання одиниці програмного засобу даного виду, грн;

$C_{прг.i}$ – кількість одиниць програмного забезпечення відповідного найменування, які придбані для проведення досліджень, шт.;

K_i – коефіцієнт, що враховує інсталяцію, налагодження програмного засобу тощо, ($K_i = 1,10 \ldots 1,12$);

k – кількість найменувань програмних засобів.

$$B_{прг} = 1300,00 \cdot 1 \cdot 1,12 = 1456 \text{ грн.}$$

Отримані результати зведемо до таблиці 5.8.

Таблиця 5.8 – Витрати на придбання програмних засобів по кожному виду

Найменування програмного засобу	Кількість, шт	Ціна за одиницю, грн	Вартість, грн
Операційна система Microsoft Windows 10 Pro for Workstations	3	15000	50400
Середовище розробки JetBrains GoLand	2	3600	8064
Графічний редактор Adobe Photoshop	1	1300	1456
Всього			59920

В спрощеному вигляді амортизаційні відрахування по кожному виду обладнання, приміщень та програмному забезпеченню, що застосовуються, розраховуємо з використанням прямолінійного методу амортизації за наступною формулою:

$$A_{обл} = \frac{Ц_б}{T_в} \cdot \frac{t_{вик}}{12}, \qquad (5.9)$$

де $Ц_б$ – балансова вартість обладнання, програмних засобів, приміщень тощо, які використовувались для проведення досліджень, грн;

$t_{вик}$ – термін використання обладнання, програмних засобів, приміщень під час досліджень, місяців;

$T_в$ – строк корисного використання обладнання, програмних засобів, приміщень тощо, років.

$$A_{обл} = (59400,00 \cdot 3) / (3 \cdot 12) = 4950,00 \text{ грн.}$$

Проведені розрахунки зведемо до таблиці 5.9.

Таблиця 5.9 – Амортизаційні відрахування по кожному виду обладнання

Найменування обладнання	Балансова вартість, грн	Строк корисного використання, років	Термін використання обладнання, місяців	Амортизаційні відрахування, грн
Робоча станція (ПК)	59400	3	3	4950,00
Маршрутизатор	1870	3	3	155,83
Сервер для середовища розробки та тестування	22000	5	3	1100,00
Всього				6205,83

Витрати на силову електроенергію (B_e) розраховуємо за формулою:

$$B_e = \sum_{i=1}^{n} \frac{W_{yi} \cdot t_i \cdot Ц_e \cdot K_{впі}}{\eta_i}, \quad (5.10)$$

де W_{yi} – встановлена потужність обладнання на визначеному етапі розробки, кВт;

t_i – тривалість роботи обладнання на етапі дослідження, год;

$Ц_e$ – вартість 1 кВт-години електроенергії, грн; (вартість електроенергії визначається за даними енергопостачальної компанії), приймемо $Ц_e$ = 7,50 грн;

$K_{впі}$ – коефіцієнт, що враховує використання потужності, $K_{впі} < 1$;

η_i – коефіцієнт корисної дії обладнання, $\eta_i < 1$.

$$B_e = 0{,}5 \cdot 180{,}0 \cdot 7{,}50 \cdot 0{,}95 / 0{,}97 = 661{,}08 \text{ грн.}$$

Проведені розрахунки зведемо до таблиці 5.10.

Таблиця 5.10 – Витрати на електроенергію

Найменування обладнання	Встановлена потужність, кВт	Тривалість роботи, год	Сума, грн
Блок живлення робочої станції	0,5	180	661,08
Блок живлення маршрутизатора	0,02	60	8,81
Блок живлення сервера	0,7	60	308,51
Всього			978,40

До статті «Службові відрядження» дослідної роботи належать витрати на відрядження штатних працівників, працівників організацій, які працюють за договорами цивільно-правового характеру, аспірантів, зайнятих розробленням досліджень, відрядження, пов'язані з проведенням випробувань машин та приладів, а також витрати на відрядження на наукові з'їзди, конференції, наради, пов'язані з виконанням конкретних досліджень.

Витрати за статтею «Службові відрядження» розраховуємо як 20…25% від суми основної заробітної плати дослідників та робітників за формулою:

$$В_{св} = (З_о + З_р) \cdot \frac{Н_{св}}{100\%}, \quad (5.11)$$

де $Н_{св}$ – норма нарахування за статтею «Службові відрядження», приймемо $Н_{св} = 20\%$.

$$В_{св} = (120428{,}57 + 3566{,}55) \cdot 20 / 100\% = 24799{,}03 \text{ грн.}$$

Витрати за статтею «Витрати на роботи, які виконують сторонні підприємства, установи і організації» розраховуємо як 30…45% від суми основної заробітної плати дослідників та робітників за формулою:

$$В_{сп} = (З_о + З_р) \cdot \frac{Н_{сп}}{100\%}, \quad (5.12)$$

де $Н_{сп}$ – норма нарахування за статтею «Витрати на роботи, які виконують сторонні підприємства, установи і організації», приймемо $Н_{сп} = 30\%$.

$$В_{сп} = (120428{,}57 + 3566{,}55) \cdot 30 / 100\% = 37198{,}54 \text{ грн.}$$

До статті «Інші витрати» належать витрати, які не знайшли відображення у зазначених статтях витрат і можуть бути віднесені безпосередньо на собівартість досліджень за прямими ознаками.

Витрати за статтею «Інші витрати» розраховуємо як 50…100% від суми основної заробітної плати дослідників та робітників за формулою:

$$І_в = (З_о + З_р) \cdot \frac{Н_{ів}}{100\%}, \quad (5.13)$$

де $Н_{ів}$ – норма нарахування за статтею «Інші витрати», приймемо $Н_{ів} = 50\%$.

$$І_в = (120428{,}57 + 3566{,}55) \cdot 50 / 100\% = 61997{,}56 \text{ грн.}$$

До статті «Накладні (загальновиробничі) витрати» належать: витрати, пов’язані з управлінням організацією; витрати на винахідництво та раціоналізацію; витрати на підготовку (перепідготовку) та навчання кадрів; витрати, пов’язані з набором робочої сили; витрати на оплату послуг банків; витрати, пов’язані з освоєнням виробництва продукції; витрати на науково-технічну інформацію та рекламу та ін.

Витрати за статтею «Накладні (загальновиробничі) витрати» розраховуємо як 100…150% від суми основної заробітної плати дослідників та робітників за формулою:

$$B_{нзв} = (З_о + З_р) \cdot \frac{H_{нзв}}{100\%} , \qquad (5.14)$$

де $H_{нзв}$ – норма нарахування за статтею «Накладні (загальновиробничі) витрати», приймемо $Н_{нзв}$ = 120%.

$$B_{нзв} = (120428{,}57 + 3566{,}55) \cdot 120 / 100\% = 148\ 794{,}15 \text{ грн.}$$

Витрати на проведення науково-дослідної роботи розраховуємо як суму всіх попередніх статей витрат за формулою:

$$B_{заг} = З_о + З_р + З_{дод} + З_н + M + K_в + B_{спец} + B_{прг} + A_{обл} + B_е + B_{св} + B_{сп} + I_в + B_{нзв} . \qquad (5.15)$$

$$B_{заг} = 593756{,}45 \text{ грн.}$$

Загальні витрати *ЗВ* на завершення науково-дослідної (науково-технічної) роботи та оформлення її результатів розраховується за формулою:

$$ЗВ = \frac{B_{заг}}{\eta} , \qquad (5.16)$$

де η - коефіцієнт, який характеризує етап (стадію) виконання науково-дослідної роботи, приймемо η=0,9.

$$ЗВ = 593756{,}45 / 0{,}9 = 659729{,}39 \text{ грн.}$$

5.3 Розрахунок економічної ефективності науково-технічної розробки при її можливій комерціалізації потенційним інвестором

В ринкових умовах узагальнюючим позитивним результатом, що його може отримати потенційний інвестор від можливого впровадження результатів

тієї чи іншої науково-технічної розробки, є збільшення у потенційного інвестора величини чистого прибутку.

Результати дослідження проведені за темою «Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів» передбачають комерціалізацію протягом 3-х років реалізації на ринку.

В цьому випадку майбутній економічний ефект буде формуватися на основі таких даних:

ΔN – збільшення кількості споживачів продукту, у періоди часу, що аналізуються, від покращення його певних характеристик;

1-й рік – 300 користувачів;

2-й рік – 400 користувачів;

3-й рік – 350 користувачів.

N – кількість споживачів які використовували аналогічний продукт у році до впровадження результатів нової науково-технічної розробки, приймемо 800 користувачів;

$Ц_б$ – вартість програмного продукту у році до впровадження результатів розробки, приймемо 11000 грн;

$\pm\Delta Ц_о$ – зміна вартості програмного продукту від впровадження результатів науково-технічної розробки, приймемо 2000 грн.

Можливе збільшення чистого прибутку у потенційного інвестора $\Delta П_i$ для кожного із 3-х років, протягом яких очікується отримання позитивних результатів від можливого впровадження та комерціалізації науково-технічної розробки, розраховуємо за формулою [47]:

$$\Delta П_i = (\pm\Delta Ц_о \cdot N + Ц_о \cdot \Delta N)_i \cdot \lambda \cdot \rho \cdot (1 - \frac{\vartheta}{100}), \quad (5.17)$$

де λ – коефіцієнт, який враховує сплату потенційним інвестором податку на додану вартість. У 2023 році ставка податку на додану вартість складає 20%, а коефіцієнт $\lambda = 0,8333$;

ρ – коефіцієнт, який враховує рентабельність інноваційного продукту.

Приймемо ρ=40%;

ϑ – ставка податку на прибуток, який має сплачувати потенційний інвестор, у 2023 році $\vartheta=18\%$;

Збільшення чистого прибутку 1-го року:

$\Delta П_1 =$(2000,00·800,00+13000,00·300)·0,83·0,4·(1-0,18/100%)= 1502732 грн.

Збільшення чистого прибутку 2-го року:

$\Delta П_2 =$(2000,00·800,00+13000,00·(300+400))·0,83·0,4·(1-0,18/100%)=

=2923496,8 грн.

Збільшення чистого прибутку 3-го року:

$\Delta П_3 =$(2000,00·800,00+130000,00·(300+400+350))·0,83·0,4·(1-0,18/100%)=

=4166666 грн.

Приведена вартість збільшення всіх чистих прибутків *ПП*, що їх може отримати потенційний інвестор від можливого впровадження та комерціалізації науково-технічної розробки:

$$ПП = \sum_{i=1}^{T} \frac{\Delta П_i}{(1+\tau)^t}, \qquad (5.18)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному з років, протягом яких виявляються результати впровадження науково-технічної розробки, грн;

T – період часу, протягом якого очікується отримання позитивних результатів від впровадження та комерціалізації науково-технічної розробки, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні, τ=0,15;

t – період часу (в роках) від моменту початку впровадження науково-технічної розробки до моменту отримання потенційним інвестором додаткових чистих прибутків у цьому році.

$ПП$=1502732/(1+0,15)^1+2923496,8/(1+0,15)^2+4166666/(1+0,15)^3=6256957,60 грн.

Величина початкових інвестицій PV, які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки:

$$PV = k_{інв} \cdot ЗВ, \qquad (5.19)$$

де $k_{інв}$ – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію, приймаємо $k_{інв}$ =2;

$ЗВ$ – загальні витрати на проведення науково-технічної розробки та оформлення її результатів, приймаємо 659729,39 грн.

$PV = k_{інв} \cdot ЗВ$ = 2 · 659729,39 = 1319458,776 грн.

Абсолютний економічний ефект $E_{абс}$ для потенційного інвестора від можливого впровадження та комерціалізації науково-технічної розробки становитиме:

$$E_{абс} = ПП - PV \qquad (5.20)$$

де $ПП$ – приведена вартість зростання всіх чистих прибутків від можливого впровадження та комерціалізації науково-технічної розробки, 6256957,60 грн;

PV – теперішня вартість початкових інвестицій, 1319458,776 грн.

$E_{абс} = ПП - PV$ = 6256957,60 - 1319458,776= 4937498,82 грн.

Внутрішня економічна дохідність інвестицій $E_в$, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки:

$$E_в = \sqrt[Тж]{1 + \frac{E_{абс}}{PV}} - 1, \qquad (5.21)$$

де $E_{абс}$ – абсолютний економічний ефект вкладених інвестицій, 4937498,82 грн;

PV – теперішня вартість початкових інвестицій, 1319458,776 грн;

$T_ж$ – життєвий цикл науково-технічної розробки, тобто час від початку її розробки до закінчення отримування позитивних результатів від її впровадження, 3 роки.

$E_в = \sqrt[Тж]{1 + \frac{E_{абс}}{PV}} - 1$ = (1+4937498,82/1319458,776)$^{1/3}$= 0,68.

Мінімальна внутрішня економічна дохідність вкладених інвестицій $\tau_{мін}$:

$$\tau_{мін} = d + f \,, \qquad (5.22)$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2023 році в Україні d =0,11;

f – показник, що характеризує ризикованість вкладення інвестицій, приймемо 0,18.

$$\tau_{мін} = 0{,}11+0{,}18 = 0{,}29$$

Отримане значення 0,29 < 0,68 свідчить про те, що внутрішня економічна дохідність інвестицій $E_в$, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки вища мінімальної внутрішньої дохідності.

Тобто інвестувати в науково-дослідну роботу за темою «Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів» доцільно.

Період окупності інвестицій $T_{ок}$ які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки розраховуються за формулою:

$$T_{ок} = \frac{1}{E_в} \,, \qquad (5.23)$$

де $E_в$ – внутрішня економічна дохідність вкладених інвестицій.

$$T_{ок} = 1 \,/\, 0{,}68 = 1{,}47 \text{ року.}$$

$T_{ок} <$ 3-х років, що свідчить про комерційну привабливість науково-технічної розробки і може спонукати потенційного інвестора профінансувати впровадження даної розробки та виведення її на ринок.

5.4 Висновки

Таким чином, згідно проведених досліджень рівень комерційного потенціалу розробки за темою «Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання

та масштабування електронних ресурсів» становить 40 балів, що, свідчить про комерційну важливість проведення даних досліджень (рівень комерційного потенціалу розробки вищий середнього).

Також термін окупності становить 1,47 р., що менше 3-х років, що свідчить про комерційну привабливість науково-технічної розробки і може спонукати потенційного інвестора профінансувати впровадження даної розробки та виведення її на ринок. Тобто інвестувати в науково-дослідну роботу за темою «Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів» доцільно.

За результатами економічної частини можна зробити висновок про доцільність проведення науково-дослідної роботи за темою «Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів».

ВИСНОВКИ

У магістерській кваліфікаційній роботі було розроблено методи та алгоритми для підвищення ефективності процесів розгортання та масштабування електронних ресурсів, робота яких демонстрована за допомогою розробленого програмного забезпечення для управління конфігураціями.

Проведено аналіз сучасного стану питання та порівняння існуючих аналогів програмного забезпечення для управління конфігураціями, досліджено їх переваги та недоліки. Проаналізовано існуючі методи для забезпечення високої доступності системи, автоматичної інсталяції програм-агентів та інтеграції з хмарними провайдерами. За результатами аналізу було прийнято рішення про розробку власного програмного додатку, що буде реалізовувати покращені методи для підвищення ефективності процесів розгортання та масштабування електронних ресурсів. Проведено постановку задач розробки.

Проведено розробку методів та алгоритмів програмного продукту. Подальшого розвитку отримав метод для забезпечення високої доступності, який, на відміну від існуючих, було адаптовано та розширено шляхом застосування технік кластеризації, балансування навантаження та алгоритму консенсусу для застосування в ПЗ для управління конфігураціями.

Удосконалено метод автоматичної інсталяції програм-агентів на керовані сервери, що на відміну від інших рішень застосовує гібридну комунікацію з вузлами системи та дозволяє автоматизувати типові дії кінцевого користувача програмного забезпечення по підготовці кластера.

Подальшого розвитку отримав метод автоматичного створення серверів з використанням хмарних провайдерів, що на відміну від аналогів дозволяє не тільки створити нові хмарні ресурси, а й відразу застосовувати їх у складі інфраструктури програмної системи для управління конфігураціями, що дозволило автоматизувати розгортання кластера. Проведено розробку блок-схем алгоритмів для відповідних методів та розробку графічного інтерфейсу користувача.

Виконано варіантний аналіз і обґрунтування вибору засобів та середовища розробки для реалізації програмного продукту. За результатами аналізу було обрано мову програмування Go та середовище розробки JetBrains GoLand. Виконано програмну реалізацію додатку для управління конфігураціями з розробленими удосконаленими методами.

Проведено аналіз методів тестування програмного забезпечення. За результатами аналізу було обрано метод «сірого ящика» для перевірки працездатності розробленого продукту. Розроблено ряд тестових випадків для тестування функції ПЗ, перевірка працездатності не виявила недоліків.

Проведено дослідження ефективності розробленого методу забезпечення високої доступності, що включало в себе порівняння методу з популярними стратегіями аварійного відновлення за параметрами RTO та RPO, а також апріорне ранжування за рядом обраних факторів. Дослідження показало, що розроблений метод забезпечення високої доступності є актуальним та доцільним для використання.

Розроблено інструкцію користувача та сформовано мінімальну та рекомендовану конфігурації апаратного забезпечення для використання ПЗ. Проведено оцінювання комерційного потенціалу розробки, яке показало, що розробка має достатній рівень комерційного потенціалу. Задачі магістерської кваліфікаційної роботи виконано в повному обсязі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Franke U. IT service outage cost: case study and implications for cyber insurance [Електронний ресурс] / Ulrik Franke // The geneva papers on risk and insurance - issues and practice. – 2020. – Т. 45, № 4. – С. 760–784. – Режим доступу: https://doi.org/10.1057/s41288-020-00177-4 (дата звернення: 29.11.2023). – Назва з екрана.

2. Olawuyi J. Configuration management / J. Olawuyi, Mercy Benson-Emenike, Oju Onuoha // West Africa journal of science, technology and social. – 2023. – № 1. – С. 97–105.

3. Unleashing full potential of Ansible framework: university labs administration [Електронний ресурс] / Pavel Masek [та ін.] // 2018 22nd conference of open innovations association (FRUCT), Jyvaskyla, 15–18 трав. 2018 р. – [Б. м.], 2018. – Режим доступу: https://doi.org/10.23919/fruct.2018.8468270 (дата звернення: 29.11.2023). – Назва з екрана.

4. From monolithic systems to microservices: a comparative study of performance [Електронний ресурс] / Freddy Tapia [та ін.] // Applied sciences. – 2020. – Т. 10, № 17. – С. 5797. – Режим доступу: https://doi.org/10.3390/app10175797 (дата звернення: 29.11.2023). – Назва з екрана.

5. Миргородський А. В. Аналіз методів для управління конфігураціями при розгортанні електронних ресурсів / Андрій Вікторович Миргородський, Оксана Володимирівна Романюк // Електронні інформаційні ресурси: створення, використання, доступ : Міжнар. науково-практ. Інтернет-конф., Суми/Вінниця, 28–29 листоп. 2022 р. – Суми/Вінниця, 2022. – С. 152–156.

6. Миргородський А. В. Розробка розподілених систем з використанням алгоритму консенсусу Raft [Електронний ресурс] / Андрій Вікторович Миргородський, Оксана Володимирівна Романюк // LII Науково-технічна конференція факультету інформаційних технологій та комп'ютерної інженерії, Вінниця, 21–23 черв. 2023 р. – Вінниця, 2023. – Режим доступу:

https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2023/paper/view/17164/14546 (дата звернення: 30.11.2023). – Назва з екрана.

7. Миргородський А. В. Розробка методу забезпечення високої доступності для програмного забезпечення управління конфігураціями / Андрій Вікторович Миргородський, Оксана Володимирівна Романюк // Оптико-електронні інформаційно-енергетичні технології. – 2023. – Т. 46, № 2. – С. 45-54.

8. The cloud's cloudy moment: a systematic survey of public cloud service outage [Електронний ресурс] / Zheng Li [та ін.] // International journal of cloud computing and services science (IJ-CLOSER). – 2013. – Т. 2, № 5. – Режим доступу: https://doi.org/10.11591/closer.v2i5.5125 (дата звернення: 30.11.2023). – Назва з екрана.

9. Elasticity in cloud computing: state of the art and research challenges [Електронний ресурс] / Yahya Al-Dhuraibi [та ін.] // IEEE transactions on services computing. – 2018. – Т. 11, № 2. – С. 430–447. – Режим доступу: https://doi.org/10.1109/tsc.2017.2711009 (дата звернення: 30.11.2023). – Назва з екрана.

10. Aiello B. Configuration management best practices: practical methods that work in the real world / Bob Aiello, Leslie Sachs. – Upper Saddle River, NJ : Addison-Wesley Professional, 2010. – 272 с.

11. Olubisi I. What is Ansible? A tool to automate parts of your job [Електронний ресурс] / Idris Olubisi // freeCodeCamp.org. – Режим доступу: https://www.freecodecamp.org/news/what-is-ansible/ (дата звернення: 01.12.2023). – Назва з екрана.

12. Afreen S. What is Puppet and how does it work? [Електронний ресурс] / Sana Afreen // Simplilearn.com. – Режим доступу: https://www.simplilearn.com/tutorials/puppet-tutorial/what-is-puppet (дата звернення: 01.12.2023). – Назва з екрана.

13. Seshachala S. Automation, provisioning & configuration management (CHEF) [Електронний ресурс] / Sudhi Seshachala // DevOps.com. – Режим доступу:

https://devops.com/automation-provisioning-configuration-management-chef/ (дата звернення: 01.12.2023). – Назва з екрана.

14. Heller M. Why use SaltStack for automation and orchestration [Електронний ресурс] / Martin Heller // InfoWorld. – Режим доступу: https://www.infoworld.com/article/3277970/why-use-saltstack-for-automation-and-orchestration.html (дата звернення: 01.12.2023). – Назва з екрана.

15. Research on high availability architecture of cloud platform [Електронний ресурс] / Lai Xinming [та ін.] // Journal of physics: conference series. – 2019. – № 1345. – Режим доступу: https://doi.org/10.1088/1742-6596/1345/2/022044 (дата звернення: 01.12.2023). – Назва з екрана.

16. Clustering techniques for software engineering [Електронний ресурс] / Shohag Barman [та ін.] // Indonesian journal of electrical engineering and computer science. – 2016. – Т. 4, № 2. – С. 465. – Режим доступу: https://doi.org/ 10.11591/ijeecs.v4.i2.pp465-472 (дата звернення: 01.12.2023). – Назва з екрана.

17. Nwobodo I. Cloud computing: a detailed relationship to grid and cluster computing [Електронний ресурс] / Ikechukwu Nwobodo // International journal of future computer and communication. – 2015. – Т. 4, № 2. – С. 82–87. – Режим доступу: https://doi.org/10.7763/ijfcc.2015.v4.361 (дата звернення: 01.12.2023). – Назва з екрана.

18. Andry J. F. Using backup and restore automation from disaster in university information systems [Електронний ресурс] / Johanes Fernandes Andry, Honni Po // 2nd international conference on innovative research across disciplines (ICIRAD 2017), Denpasar, Bali-Indonesia, 26 серп. 2017 р. – Paris, France, 2017. – Режим доступу: https://doi.org/10.2991/icirad-17.2017.1 (дата звернення: 01.12.2023). – Назва з екрана.

19. Bondi A. B. Characteristics of scalability and their impact on performance [Електронний ресурс] / André B. Bondi // The second international workshop, Ottawa, Ontario, Canada. – New York, New York, USA, 2000. – Режим доступу: https://doi.org/10.1145/350391.350432 (дата звернення: 01.12.2023). – Назва з екрана.

20. Rajak R. Load balancing techniques in cloud platform: a systematic study [Електронний ресурс] / Ranjit Rajak, Anjali Choudhary, Mohammad Sajid // International journal of experimental research and review. – 2023. – Т. 30. – С. 15–24. – Режим доступу: https://doi.org/10.52756/ijerr.2023.v30.002 (дата звернення: 01.12.2023). – Назва з екрана.

21. Stecca M. Sticky session support in auto scaling IaaS systems [Електронний ресурс] / Michele Stecca, Luca Bazzucco, Massimo Maresca // 2011 IEEE world congress on services (SERVICES), Washington, DC, USA, 4–9 лип. 2011 р. – [Б. м.], 2011. – Режим доступу: https://doi.org/10.1109/services.2011.27 (дата звернення: 01.12.2023). – Назва з екрана.

22. Gaol F. L. Design and development of the application monitoring the use of server resources for server maintenance [Електронний ресурс] / Ford Lumban Gaol, Steven Santoso, Tokuro Matsuo // Open engineering. – 2022. – Т. 12, № 1. – С. 524–538. – Режим доступу: https://doi.org/10.1515/eng-2022-0055 (дата звернення: 01.12.2023). – Назва з екрана.

23. Remote access protocols for Desktop-as-a-Service solutions [Електронний ресурс] / Eduardo Magaña [та ін.] // Plos one. – 2019. – Т. 14, № 1. – С. 1–28. – Режим доступу: https://doi.org/10.1371/journal.pone.0207512 (дата звернення: 01.12.2023). – Назва з екрана.

24. Kanade V. Understanding the working and benefits of SSH [Електронний ресурс] / Vijay Kanade // Spiceworks. – Режим доступу: https://www.spiceworks.com/it-security/network-security/articles/what-is-ssh/ (дата звернення: 01.12.2023). – Назва з екрана.

25. Merimovich B. Cloud automation with WinRM vs SSH [Електронний ресурс] / Barak Merimovich // Cloudify. – Режим доступу: https://cloudify.co/blog/cloud-automation-with-winrm-vs-ssh/ (дата звернення: 01.12.2023). – Назва з екрана.

26. Froehlich A. What are the advantages and disadvantages of CLI and GUI? | TechTarget [Електронний ресурс] / Andrew Froehlich // Networking. – Режим доступу: https://www.techtarget.com/searchnetworking/answer/What-are-the-

advantages-and-disadvantages-of-CLI-and-GUI (дата звернення: 01.12.2023). – Назва з екрана.

27. Domantas G. What is CLI? [Електронний ресурс] / G. Domantas // Hostinger Tutorials. – Режим доступу: https://www.hostinger.com/tutorials/what-is-cli (дата звернення: 01.12.2023). – Назва з екрана.

28. Campbell B. Terraform In-Depth [Електронний ресурс] / Bradley Campbell // The definitive guide to AWS infrastructure automation. – Berkeley, CA, 2019. – С. 123–203. – Режим доступу: https://doi.org/10.1007/978-1-4842-5398-4_4 (дата звернення: 01.12.2023). – Назва з екрана.

29. Redundancy mechanisms applied in cloud computing infrastructures [Електронний ресурс] / Rosangela Melo [та ін.] // 2018 13th Iberian conference on information systems and technologies (CISTI), Caceres, 13–16 черв. 2018 р. – [Б. м.], 2018. – Режим доступу: https://doi.org/10.23919/cisti.2018.8399255 (дата звернення: 02.12.2023). – Назва з екрана.

30. Jin C. Understanding security group usage in a public IaaS cloud [Електронний ресурс] / Cheng Jin, Abhinav Srivastava, Zhi-Li Zhang // IEEE INFOCOM 2016 - IEEE conference on computer communications, San Francisco, CA, USA, 10–14 квіт. 2016 р. – [Б. м.], 2016. – Режим доступу: https://doi.org/10.1109/infocom.2016.7524508 (дата звернення: 02.12.2023). – Назва з екрана.

31. Indu I. Identity and access management in cloud environment: mechanisms and challenges [Електронний ресурс] / I. Indu, P. M. Rubesh Anand, Vidhyacharan Bhaskar // Engineering science and technology, an international journal. – 2018. – Т. 21, № 4. – С. 574–588. – Режим доступу: https://doi.org/10.1016/j.jestch.2018.05.010 (дата звернення: 02.12.2023). – Назва з екрана.

32. Scott S. Working with AWS networking & Amazon VPC [Електронний ресурс] / Stuart Scott // Cloud Academy. – Режим доступу: https://cloudacademy.com/blog/aws-networking-amazon-vpc/ (дата звернення: 02.12.2023). – Назва з екрана.

33. Collins T. What is user interface (UI)? (types & features) [Електронний ресурс] / Tom Collins // BrowserStack. – Режим доступу: https://www.browserstack.com/guide/what-is-user-interface (дата звернення: 02.12.2023). – Назва з екрана.

34. Iglberger K. C++ software design: design principles and patterns for highquality software / K. Iglberger. – Sebastopol : O'Reilly Media, 2022. – 435 с.

35. Skeet J. C# in depth / J. Skeet. – 4-те вид. – Shelter Island : Manning, 2019. – 528 с.

36. Romano F. Learn Python Programming: An in-depth introduction to the fundamentals of Python / Fabrizio Romano, Heinrich Kruger. – 3-тє вид. – Birmingham : Packt Publishing, 2021. – 554 с.

37. Donovan A. The Go programming language / A. Donovan, B. W. Kernighan. – Boston : Addison-Wesley Professional, 2015. – 400 с.

38. Belton J. Why you Need to Learn Go if you're interested in DevOps [Електронний ресурс] / J. Belton // Medium. – Режим доступу: https://medium.com/@joelbelton/why-you-need-to-learn-go-if-youre-interested-in-devops-19839e7dd295 (дата звернення: 03.12.2023). – Назва з екрана.

39. Fan B. VSCode vs Jetbrains – revisited [Електронний ресурс] / Brandon Fan // Shade. – Режим доступу: https://www.shade.inc/posts/vscode-vs-jetbrains-revisited-update-sep-2023 (дата звернення: 03.12.2023). – Назва з екрана.

40. GitHub - visualfc/liteide: LiteIDE is a simple, open source, cross-platform Go IDE. [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/visualfc/liteide#features (дата звернення: 03.12.2023). – Назва з екрана.

41. Obregon A. Supercharge your Go development with JetBrains GoLand [Електронний ресурс] / Alexander Obregon // Medium. – Режим доступу: https://medium.com/@AlexanderObregon/supercharge-your-go-development-with-jetbrains-goland-5fcd5fb85956 (дата звернення: 03.12.2023). – Назва з екрана.

42. Tyson M. Intro to gRPC: the REST alternative [Електронний ресурс] / Matthew Tyson // InfoWorld. – Режим доступу:

https://www.infoworld.com/article/3704988/intro-to-grpc-the-rest-alternative.html (дата звернення: 03.12.2023). – Назва з екрана.

43. Kamani S. An introduction to channels in Go (Golang) [Електронний ресурс] / Soham Kamani // Soham Kamani. – Режим доступу: https://www.sohamkamani.com/golang/channels/ (дата звернення: 03.12.2023). – Назва з екрана.

44. Graham D. Foundations of software testing ISTQB certification / Dorothy Graham, Rex Black, Erik Veenendaal. – 4-те вид. – Andover : Cengage Learning EMEA, 2019. – 288 с.

45. Ehmer M. A comparative study of white box, black box and grey box testing techniques / Mohd Ehmer, Farmeena Khan // International journal of advanced computer science and applications. – 2012. – Т. 3, № 6.

46. Singhal R. Chi-square test and its application in hypothesis testing [Електронний ресурс] / Richa Singhal, Rakesh Rana // Journal of the practice of cardiovascular sciences. – 2015. – Т. 1, № 1. – С. 69. – Режим доступу: https://doi.org/10.4103/2395-5414.157577 (дата звернення: 04.12.2023). – Назва з екрана.

47. Методичні вказівки до виконання економічної частини магістерських кваліфікаційних робіт / уклад.: В. О. Козловський, О. Й. Лесько, В. В. Кавецький. – Вінниця : ВНТУ, 2021. – 42 с.

ДОДАТКИ

122

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії

УЗГОДЖЕНО
Керівник відділу розробки
ПЗ ТОВ «ДСофтвер»
Малик Р.О.
«25» вересня 2023 року

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
д.т.н., проф. Романюк О.Н.

«19» вересня 2023 року

Технічне завдання
на магістерську кваліфікаційну роботу «Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів» за спеціальністю 121 – Інженерія програмного забезпечення

Керівник магістерської кваліфікаційної роботи:
_____________ к.т.н., доц. каф. ПЗ О.В. Романюк
«19» вересня 2023 р.

Виконав:
_______ студент гр. 3ПІ-22м А.В. Миргородський
«19» вересня 2023 р

Вінниця – 2023 року

1. Найменування та галузь застосування

Магістерська кваліфікаційна робота: «Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів».

Галузь застосування – автоматизація процесів управління конфігураціями, розгорнення і масштабування електронних ресурсів.

2. Підстава для розробки

Підставою для виконання магістерської кваліфікаційної роботи (МКР) є індивідуальне завдання на МКР та наказ №247 від 18 вересня 2023 року ректора по ВНТУ про закріплення тем МКР.

3. Мета та призначення розробки

Метою роботи є підвищення ефективності процесів розгортання та масштабування електронних ресурсів.

Призначення роботи – розробка методів і засобів для підвищення ефективності процесів розгортання та масштабування електронних ресурсів за допомогою програмного забезпечення управління конфігураціями.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись МКР.

1. Aiello B. Configuration management best practices: practical methods that work in the real world / Bob Aiello, Leslie Sachs. – Upper Saddle River, NJ : Addison-Wesley Professional, 2010. – 272 с.

2. Merimovich B. Cloud automation with WinRM vs SSH [Електронний ресурс] / Barak Merimovich // Cloudify. – Режим доступу: https://cloudify.co/blog/cloud-automation-with-winrm-vs-ssh/ (дата звернення: 01.12.2023). – Назва з екрана.

3. Redundancy mechanisms applied in cloud computing infrastructures [Електронний ресурс] / Rosangela Melo [та ін.] // 2018 13th Iberian conference on information systems and technologies (CISTI), Caceres, 13–16 черв. 2018 р. – [Б. м.], 2018. – Режим доступу: https://doi.org/10.23919/cisti.2018.8399255 (дата звернення: 02.12.2023). – Назва з екрана.

4. Donovan A. The Go programming language / A. Donovan, B. W. Kernighan. – Boston : Addison-Wesley Professional, 2015. – 400 с.

5. Graham D. Foundations of software testing ISTQB certification / Dorothy Graham, Rex Black, Erik Veenendaal. – 4-те вид. – Andover : Cengage Learning EMEA, 2019. – 288 с.

5. Технічні вимоги

Модель розробки – ітеративна; метод передачі повідомлень між серверами – виклик віддалених процедур (RPC); вхідні дані – текстові файли в форматі YAML з даними для виконання методів або описом бажаного стану системи; вихідні дані – логи виконання методів та приведення системи в бажаний стан; середовище розробки – JetBrains GoLand; мова програмування – Go.

6. Конструктивні вимоги

Конструкція пристрою повинна відповідати естетичним та ергономічним вимогам, повинна бути зручною в обслуговуванні та керуванні.

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

– пояснювальна записка до МКР;

– технічне завдання;

– лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів магістерської кваліфікаційної роботи	Строк виконання етапів роботи
1	Аналіз стану питання та постановка задач дослідження	20.09.2023 – 01.10.2023
2	Розробка методів та алгоритмів програмного продукту	02.10.2023 – 09.10.2023
3	Розробка програмних компонент застосунку	10.10.2023 – 19.10.2023
4	Дослідження ефективності запропонованих методів та тестування програмного забезпечення	20.10.2023 – 12.11.2023
5	Економічна частина	13.11.2023 – 01.12.2023

10. Порядок контролю та прийняття

Виконання етапів магістерської кваліфікаційної роботи контролюється керівником згідно з графіком виконання роботи. Прийняття магістерської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком

126

ДОДАТОК Б
Протокол перевірки МКР на плагіат

ПРОТОКОЛ ПЕРЕВІРКИ НАВЧАЛЬНОЇ (КВАЛІФІКАЦІЙНОЇ) РОБОТИ

Назва роботи: **Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів.**

Тип роботи: кваліфікаційна робота
Підрозділ: кафедра програмного забезпечення, ФІТКІ, 3ПІ-22м
Науковий керівник: к.т.н. доц. Романюк О. В.

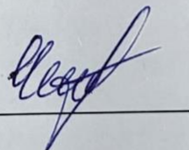
Unicheck	
Оригінальність	**96,9%**
Схожість	3,1%

Аналіз звіту подібності

■ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**

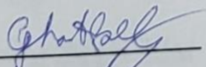
□ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

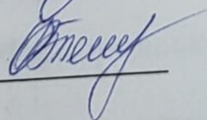
□ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку  Черноволик Г. О.

Опис прийнятого рішення: **допустити до захисту**

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck.

Автор роботи  Миргородський А.В.

Керівник роботи  Романюк О.В.

ДОДАТОК В
Акт впровадження на підприємстві

УЗГОДЖЕНО
Керівник відділу розробки
ПЗ ТОВ «ДСофтвер»
Малик Р.О.
«___» __________ 2023 року

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
д.т.н., проф. Романюк О.Н.

«01» грудня 2023 року

АКТ ВПРОВАДЖЕННЯ № ___
результатів науково-дослідних робіт

Замовник: ТОВ «ДСофтвер»
(найменування організації)

Цим актом підтверджується, що результат роботи «Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів»,
(найменування теми)

що виконав студент групи ЗПІ-22м ВНТУ Миргородський А.В.,
(виконавець)

за договором про творчу співдружність без взаємних грошових розрахунків, на громадських засадах відповідно до теми затвердженої наказом №247 від «18» _вересня__ 2023 р.

виконана з 19.09.2023 по 05.12.2023,
(строки виконання)

впроваджено у ТОВ «ДСофтвер»
(найменування організації, де здійснювалося впровадження)

1. Вид впроваджених результатів: експлуатація програмного забезпечення
(експлуатація виробу, роботи, технології)
2. Характеристика масштабу впровадження: одиничне
(унікальне, одиничне, партія, масове, серійне)
3. Форма впровадження: дослідний зразок
4. Новизна результатів науково-дослідної роботи: якісно нові
(піонерські, принципово нові, якісно нові, модифікації, модернізація старих розробок)
5. Впроваджені: в системі розгортання цифрової інфраструктури
6. Соціальний та науково-технічний ефект: науково-технічних напрямків
(охорона навколишнього середовища, поліпшення й оздоровлення умов праці, удосконалення структури керування, науково-технічних напрямків, спеціальне призначення)

Від виконавця:
Студент групи ЗПІ-22м
____________ Миргородський А.В.
Керівник: к.т.н., доцент кафедри ПЗ

Від ТОВ «ДСофтвер»
Керівник відділу розробки ПЗ
____________ Малик Р. О.
____________ Романюк О. В.

ДОДАТОК Г

Лістинг коду

cmd/cli/main.go

```
package main

import (
	"context"
	"detrint_v2/pkg/cluster"
	"detrint_v2/pkg/config"
	"detrint_v2/pkg/web"
	"errors"
	"flag"
	"github.com/gin-contrib/cors"
	"github.com/gin-gonic/gin"
	"github.com/rs/zerolog/log"
	"google.golang.org/grpc"
	"gopkg.in/yaml.v3"
	"net"
	"net/http"
	"os"
	"os/signal"
	"syscall"
	"time"
)

func main() {
	// Config
	configFilename := flag.String("config", "config.yaml", "yaml-file with node
configuration")

	bytes, err := os.ReadFile(*configFilename)
	if err != nil {
		log.Fatal().Err(err).Msg("Failed to read config file")
	}

	var cfg config.Config
	err = yaml.Unmarshal(bytes, &cfg)
	if err != nil {
		log.Fatal().Err(err).Msg("Failed to unmarshal config file")
	}

	log.Debug().Any("config", cfg).Msg("Input config file")

	// Web Server
	router := gin.New()
	ginLogger := log.With().Str("component", "gin").Logger()
	router.Use(gin.LoggerWithConfig(gin.LoggerConfig{Output: ginLogger}))
	router.Use(gin.Recovery())
	router.Use(cors.Default())

	// UI
	err = web.RegisterUI(router)
	if err != nil {
		log.Fatal().Err(err).Msg("Failed to start WebUI")
	}

	// Cluster
	cl, err := cluster.CreateFromConfig(cfg)
```

```
if err != nil {
   log.Fatal().Err(err).Msg("Failed to create cluster from config")
}

// API
web.RegisterAPI(router, cl)

// gRPC server
grpcServer := grpc.NewServer()
cl.Transport.Register(grpcServer)

// Control flow
webServer := &http.Server{Addr: cfg.Node.WebAddress, Handler: router}

c := make(chan os.Signal)
signal.Notify(c, os.Interrupt, syscall.SIGTERM)

go func() {
   err := webServer.ListenAndServe()
   if err != nil {
      log.Error().Err(err).Msg("Failed to start web server")
      return
   }
}()

go func() {
   sock, err := net.Listen("tcp", cfg.Node.GRpcAddress)
   if err != nil {
      log.Error().Err(err).Msg("Failed to listen gRPC port")
      return
   }
   err = grpcServer.Serve(sock)
   if err != nil {
      log.Error().Err(err).Msg("Failed to start gRPC server")
      return
   }
}()

ticker := time.NewTicker(10 * time.Second)
quit := make(chan struct{})
go func() {
   for {
      select {
      case <-ticker.C:
         cl.UpdateState()

      case <-quit:
         ticker.Stop()
         return
      }
   }
}()

<-c
close(quit)
grpcServer.GracefulStop()
log.Info().Msg("Stopped gRPC server")
err = webServer.Shutdown(context.Background())
if err != nil && !errors.Is(err, http.ErrServerClosed) {
   log.Error().Err(err).Msg("Failed to shutdown web server")
}
log.Info().Msg("Stopped web server")

shutdownFuture := cl.Raft.Shutdown()
```

```
    err = shutdownFuture.Error()
    if err != nil {
        log.Error().Err(err).Msg("Failed to shutdown Raft communication")
    }
    log.Info().Msg("Stopped Raft communication")

    err = cl.Save()
    if err != nil {
        log.Error().Err(err).Msg("Failed to shutdown cluster")
    }
}
```

pkg/config/config.go

```
package config

type Config struct {
    Node Node `yaml:"node"`
}

type Node struct {
    GRpcAddress string `yaml:"grpc_address"`
    WebAddress  string `yaml:"web_address"`
    LocalID     string `yaml:"local_id"`
    DataDir     string `yaml:"data_dir"`
    LogsFile    string `yaml:"logs_file"`
    StableFile  string `yaml:"stable_file"`
    LocalFile   string `yaml:"local_file"`
    Bootstrap   bool   `yaml:"bootstrap"`
}
```

pkg/cluster/cluster.go

```
package cluster

import (
    "detrint_v2/pkg/config"
    "detrint_v2/pkg/remote"
    "detrint_v2/pkg/state"
    "fmt"
    transport "github.com/Jille/raft-grpc-transport"
    "github.com/google/uuid"
    "github.com/hashicorp/raft"
    raftboltdb "github.com/hashicorp/raft-boltdb"
    "github.com/rs/zerolog/log"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
    "net"
    "os"
    "path/filepath"
)

type Cluster struct {
    Raft         *raft.Raft
    Transport    *transport.Manager
    ClusterState *state.ClusterFSM
    LocalState   *state.LocalState
    Config       *config.Config
}

func CreateFromConfig(cfg config.Config) (*Cluster, error) {
```

```
    raftLogger := log.With().Str("component", "raft").Str("node",
cfg.Node.LocalID).Logger()

    // Config
    raftConfig := raft.DefaultConfig()
    raftConfig.LocalID = raft.ServerID(cfg.Node.LocalID)
    raftConfig.LogOutput = raftLogger

    // Local Data Storage
    baseDir := filepath.Join(cfg.Node.DataDir, cfg.Node.LocalID)
    err := os.MkdirAll(baseDir, os.ModePerm)
    if err != nil {
       return nil, fmt.Errorf("failed to create data dir: %w", err)
    }

    logStore, err := raftboltdb.NewBoltStore(filepath.Join(baseDir,
cfg.Node.LogsFile))
    if err != nil {
       return nil, fmt.Errorf("failed to create log store: %w", err)
    }

    stableStore, err := raftboltdb.NewBoltStore(filepath.Join(baseDir,
cfg.Node.StableFile))
    if err != nil {
       return nil, fmt.Errorf("failed to create stable store: %w", err)
    }

    snapshotStore, err := raft.NewFileSnapshotStore(baseDir, 5, raftLogger)
    if err != nil {
       return nil, fmt.Errorf("failed to create snapshot store: %w", err)
    }

    localState, err := state.NewLocalState(filepath.Join(baseDir,
cfg.Node.LocalFile))
    if err != nil {
       return nil, fmt.Errorf("failed to create/read local state file: %w", err)
    }

    // Transport
    grpcTransport := transport.New(
       raft.ServerAddress(cfg.Node.GRpcAddress),

[]grpc.DialOption{grpc.WithTransportCredentials(insecure.NewCredentials())},
    )

    // FSM & node
    fsm := &state.ClusterFSM{}

    r, err := raft.NewRaft(raftConfig, fsm, logStore, stableStore,
snapshotStore, grpcTransport.Transport())
    if err != nil {
       return nil, fmt.Errorf("failed to construct Raft node: %w", err)
    }

    if cfg.Node.Bootstrap {
       cfg := raft.Configuration{
          Servers: []raft.Server{
             {
                Suffrage: raft.Voter,
                ID:       raft.ServerID(cfg.Node.LocalID),
                Address:  raft.ServerAddress(cfg.Node.GRpcAddress),
             },
          },
       }
```

```
        f := r.BootstrapCluster(cfg)
        err := f.Error()
        if err != nil {
            return nil, fmt.Errorf("failed to bootstrap cluster: %w", err)
        }
    }

    cluster := &Cluster{
        Raft:         r,
        Transport:    grpcTransport,
        ClusterState: fsm,
        LocalState:   localState,
        Config:       &cfg,
    }

    return cluster, nil
}

func (c *Cluster) UpdateState() {
    c.LocalState.Update(c.ClusterState, c.Config.Node.LocalID)
}

func (c *Cluster) Save() error {
    baseDir := filepath.Join(c.Config.Node.DataDir, c.Config.Node.LocalID)
    return c.LocalState.Save(filepath.Join(baseDir, c.Config.Node.LocalFile))
}

func (c *Cluster) GetInfo() *config.Config {
    return c.Config
}

type HealthCheckResults struct {
    Success bool
}

func (c *Cluster) HealthCheck(newServer *config.Config) (*HealthCheckResults,
error) {
    f := c.Raft.GetConfiguration()
    err := f.Error()
    if err != nil {
        return nil, err
    }

    for _, server := range f.Configuration().Servers {
        if string(server.Address) == newServer.Node.GRpcAddress {
            return &HealthCheckResults{Success: true}, nil
        }
    }

    return &HealthCheckResults{Success: false}, nil
}

func GenerateConfig(clusterInfo *config.Config, data *remote.ConnectionData)
(*config.Config, error) {
    host, _, err := net.SplitHostPort(data.Address)
    if err != nil {
        return nil, fmt.Errorf("failed to extract host & port from connection
data: %w", err)
    }

    random, err := uuid.NewRandom()
    if err != nil {
        return nil, fmt.Errorf("failed to generate ID for new server: %w", err)
```

```go
	}

	return &config.Config{
		Node: config.Node{
			GRpcAddress: fmt.Sprintf("%s:3000", host),
			WebAddress:  fmt.Sprintf("%s:3001", host),
			LocalID:     random.String(),
			DataDir:     clusterInfo.Node.DataDir,
			LogsFile:    clusterInfo.Node.LogsFile,
			StableFile:  clusterInfo.Node.StableFile,
			LocalFile:   clusterInfo.Node.LocalFile,
			Bootstrap:   false,
		},
	}, nil
}
```

pkg/cloud/cloud.go

```go
package cloud

import (
	"context"
	"fmt"
	"github.com/aws/aws-sdk-go-v2/config"
	"github.com/aws/aws-sdk-go-v2/service/cloudformation"
	"github.com/aws/aws-sdk-go-v2/service/cloudformation/types"
	"github.com/aws/aws-sdk-go-v2/service/ec2"
	"github.com/go-ping/ping"
)

type AccessConfig struct {
	Region string
}

type Client struct {
	cloudformation *cloudformation.Client
	ec2            *ec2.Client
}

type ServersConfig struct {
	StackId   string
	StackName string
	StackURL  string
}

type NetworkConfig struct {
	Exists    bool
	StackId   string
	StackName string
	StackURL  string
}

func NewClient(cfg *AccessConfig) (*Client, error) {
	awsConfig, err := config.LoadDefaultConfig(context.Background(),
config.WithRegion(cfg.Region))
	if err != nil {
		return nil, fmt.Errorf("failed to prepare client config: %w", err)
	}
	cloudformationClient := cloudformation.NewFromConfig(awsConfig)
	ec2Client := ec2.NewFromConfig(awsConfig)
	return &Client{cloudformation: cloudformationClient, ec2: ec2Client}, nil
}
```

```
func (client *Client) IsExist(networkConfig *NetworkConfig) (bool, error) {
    if networkConfig.Exists {
       return true, nil
    }

    output, err := client.cloudformation.DescribeStacks(context.Background(),
&cloudformation.DescribeStacksInput{
       StackName: &networkConfig.StackName,
    })
    if err != nil {
       return false, fmt.Errorf("failed to get stack info: %w", err)
    }

    if output.Stacks[0].StackStatus == types.StackStatusCreateComplete {
       return true, nil
    }

    return false, nil
}

func (client *Client) PrepareNetwork(networkConfig *NetworkConfig) error {
    stack, err := client.cloudformation.CreateStack(context.Background(),
&cloudformation.CreateStackInput{
       StackName:   &networkConfig.StackName,
       TemplateURL: &networkConfig.StackURL,
    })
    if err != nil {
       return err
    }

    networkConfig.StackId = *stack.StackId
    return nil
}

func (client *Client) PrepareInstances(serversConfig *ServersConfig,
networkConfig *NetworkConfig) ([]string, error) {
    if len(networkConfig.StackId) == 0 {
       return nil, fmt.Errorf("network infra is not ready")
    }

    stack, err := client.cloudformation.CreateStack(context.Background(),
&cloudformation.CreateStackInput{
       StackName:   &serversConfig.StackName,
       TemplateURL: &serversConfig.StackURL,
    })
    if err != nil {
       return nil, err
    }

    serversConfig.StackId = *stack.StackId

    output, err :=
client.cloudformation.ListStackResources(context.Background(),
&cloudformation.ListStackResourcesInput{
       StackName: &serversConfig.StackName,
    })
    if err != nil {
       return nil, err
    }

    var instances []string
    for _, res := range output.StackResourceSummaries {
       if *res.ResourceType == "AWS::EC2::Instance" {
          instances = append(instances, *res.PhysicalResourceId)
```

```
        }
    }

    return instances, nil
}

func (client *Client) ConnectivityCheck(serversList []string) (bool, error) {
    instances, err := client.ec2.DescribeInstances(context.Background(),
&ec2.DescribeInstancesInput{
        InstanceIds: serversList,
    })
    if err != nil {
        return false, err
    }

    for _, reservation := range instances.Reservations {
        for _, instance := range reservation.Instances {
            address := instance.PublicIpAddress

            pinger, err := ping.NewPinger(*address)
            if err != nil {
                return false, err
            }

            pinger.Count = 4
            err = pinger.Run()
            if err != nil {
                return false, err
            }
        }
    }

    return true, nil
}

func (client *Client) DeleteResources(name string) error {
    _, err := client.cloudformation.DeleteStack(context.Background(),
&cloudformation.DeleteStackInput{
        StackName: &name,
    })
    if err != nil {
        return err
    }

    return nil
}
```

pkg/remote/remote.go

```
package remote

import (
    "detrint_v2/pkg/config"
    "fmt"
    "github.com/melbahja/goph"
    "github.com/pkg/sftp"
    "gopkg.in/yaml.v3"
    "os"
    "path"
)

type ConnectionType int8
```

```go
const (
    SSH ConnectionType = iota
)

type ConnectionData struct {
    Username       string
    Address        string
    RemotePath     string
    Authentication goph.Auth
    Type           ConnectionType
    RetryCount     int
}

type Client struct {
    c          *goph.Client
    remotePath string
}

func Connect(data *ConnectionData) (*Client, error) {
    if data.Type == SSH {
       client, err := goph.New(data.Username, data.Address, data.Authentication)
       if err != nil {
          return nil, err
       } else {
          return &Client{c: client, remotePath: data.RemotePath}, nil
       }
    }
    return nil, nil
}

func (client *Client) CopyInstallFiles() error {
    executable, err := os.Executable()
    if err != nil {
       return fmt.Errorf("failed to get path for current executable: %w", err)
    }
    filename := path.Base(executable)

    err = client.c.Upload(executable, path.Join(client.remotePath, filename))
    if err != nil {
       return fmt.Errorf("failed to upload file to remote: %w", err)
    }

    return nil
}

func (client *Client) CopyConfig(cfg *config.Config) error {
    bytes, err := yaml.Marshal(cfg)
    if err != nil {
       return fmt.Errorf("failed to marshall config to YAML format: %w", err)
    }

    sftpClient, err := client.c.NewSftp()
    if err != nil {
       return fmt.Errorf("failed to open SFTP connection with remote: %w", err)
    }
    defer func(sftpClient *sftp.Client) {
       _ = sftpClient.Close()
    }(sftpClient)

    file, err := sftpClient.Create(path.Join(client.remotePath, "config.yaml"))
    if err != nil {
       return fmt.Errorf("failed to create config file on remote: %w", err)
    }
    defer func(file *sftp.File) {
```

```
        _ = file.Close()
    }(file)

    _, err = file.Write(bytes)
    if err != nil {
        return fmt.Errorf("failed to write to file on remote: %w", err)
    }

    return nil
}

func (client *Client) BootstrapAgent() error {
    executable, err := os.Executable()
    if err != nil {
        return fmt.Errorf("failed to get executable name: %w", err)
    }
    filename := path.Base(executable)

    cmd, err := client.c.Command(path.Join(client.remotePath, filename))
    if err != nil {
        return fmt.Errorf("failed to prepare command for remote execution: %w",
err)
    }
    err = cmd.Run()
    if err != nil {
        return fmt.Errorf("failed to run command on remote host: %w", err)
    }

    return nil
}
```

pkg/state/fsm.go

```
package state

import (
    "github.com/hashicorp/raft"
    "github.com/rs/zerolog/log"
    "google.golang.org/protobuf/proto"
    "io"
    "sync"
)

type CurrentState struct {
    Mtx   sync.RWMutex
    State DetrintState
}

func (c *CurrentState) Apply(l *raft.Log) interface{} {
    c.Mtx.Lock()
    defer c.Mtx.Unlock()

    err := proto.Unmarshal(l.Data, &c.State)
    if err != nil {
        log.Error().Err(err).Msg("Failed to unmarshal state in Apply method")
    }

    return nil
}

func (c *CurrentState) Snapshot() (raft.FSMSnapshot, error) {
    c.Mtx.RLock()
    defer c.Mtx.RUnlock()
```

```
    return &Snapshot{State: *proto.Clone(&c.State).(*DetrintState)}, nil
}

func (c *CurrentState) Restore(snapshot io.ReadCloser) error {
    bytes, err := io.ReadAll(snapshot)
    if err != nil {
        return err
    }

    c.Mtx.Lock()
    err = proto.Unmarshal(bytes, &c.State)
    c.Mtx.Unlock()
    if err != nil {
        return err
    }

    return nil
}

type Snapshot struct {
    State DetrintState
}

func (s *Snapshot) Persist(sink raft.SnapshotSink) error {
    bytes, err := proto.Marshal(&s.State)
    if err != nil {
        _ = sink.Cancel()
        return err
    }
    _, err = sink.Write(bytes)
    if err != nil {
        _ = sink.Cancel()
        return err
    }

    return sink.Close()
}

func (s *Snapshot) Release() {

}
```

pkg/state/local_state.go

```
package state

import (
    "encoding/json"
    "errors"
    "fmt"
    "github.com/rs/zerolog/log"
    "io"
    "os"
    "reflect"
    "sync"
)

type LocalState struct {
    Mtx            sync.RWMutex           `json:"-"`
    Inventory      map[string]interface{} `json:"inventory"`
    PendingSteps   []Step                 `json:"pending_steps"`
    ProcessedSteps []Step                 `json:"processed_steps"`
```

```
}

func NewLocalState(filepath string) (*LocalState, error) {
    logger := log.With().Str("component", "local_state").Logger()

    logger.Debug().Str("filepath", filepath).Msg("Trying to find local state
file...")

    _, err := os.Stat(filepath)
    if errors.Is(err, os.ErrNotExist) {
        logger.Debug().Msg("Local state file is not exist, working with clean
state")

        return &LocalState{}, nil
    } else if err != nil {
        return nil, fmt.Errorf("can't check if file with local state exists: %w",
err)
    }

    jsonFile, err := os.Open(filepath)
    if err != nil {
        return nil, fmt.Errorf("can't open file with local state: %w", err)
    }
    defer func(jsonFile *os.File) {
        err := jsonFile.Close()
        if err != nil {
            logger.Debug().Msg("Failed to close JSON file with local state")
        }
    }(jsonFile)

    jsonBytes, err := io.ReadAll(jsonFile)
    if err != nil {
        return nil, fmt.Errorf("failed to read JSON file content: %w", err)
    }

    newState := LocalState{}
    err = json.Unmarshal(jsonBytes, &newState)
    if err != nil {
        return nil, fmt.Errorf("failed to unmarshall JSON data: %w", err)
    }

    return &newState, nil
}

func (l *LocalState) Save(filepath string) error {
    logger := log.With().Str("component", "local_state").Logger()
    logger.Debug().Str("filepath", filepath).Msg("Saving local state to
file...")

    bytes, err := json.MarshalIndent(l, "", "  ")
    if err != nil {
        return fmt.Errorf("failed to marshall local state: %w", err)
    }

    file, err := os.Create(filepath)
    if err != nil {
        return fmt.Errorf("failed to create file for local state: %w", err)
    }
    defer func(file *os.File) {
        _ = file.Close()
    }(file)

    _, err = file.Write(bytes)
    if err != nil {
```

```
        return fmt.Errorf("failed to write local state to file: %w", err)
    }

    logger.Debug().Str("filepath", filepath).Msg("Local state saved to JSON
file!")
    return nil
}

func (l *LocalState) Update(fsm *ClusterFSM, nodeId string) {
    logger := log.With().Str("component", "local_state").Str("node",
nodeId).Logger()

    logger.Info().Msg("Checking for state change...")

    newInventory, ok := fsm.State.Inventory[nodeId]
    if !ok {
        logger.Info().Msg("Inventory emptied")
        l.Mtx.Lock()
        l.Inventory = map[string]interface{}{}
        l.Mtx.Unlock()
    } else {
        equal := reflect.DeepEqual(newInventory, l.Inventory)
        if equal {
            logger.Info().Msg("Inventories equal, no updates needed")
        } else {
            logger.Info().Msg("Updating inventory...")
            l.Mtx.Lock()
            l.Inventory = newInventory
            l.Mtx.Unlock()
        }
    }

    newSteps, ok := fsm.State.PendingSteps[nodeId]
    if !ok {
        logger.Info().Msg("No new steps detected")
    } else {
        equal := reflect.DeepEqual(newSteps, l.PendingSteps)
        if equal {
            logger.Info().Msg("FSM steps equal to local steps, no updates needed")
        } else {
            // Set of existing steps ids
            existingIds := map[string]int8{}
            for i := range l.PendingSteps {
                existingIds[l.PendingSteps[i].Id] = 0
            }

            // Check new steps
            var checkedSteps []Step
            for i := range newSteps {
                _, ok := existingIds[newSteps[i].Id]
                if !ok {
                    checkedSteps = append(checkedSteps, newSteps[i])
                }
            }

            if len(checkedSteps) > 0 {
                logger.Info().Msg("Detected steps with new IDs, updating steps...")
                l.Mtx.Lock()
                l.PendingSteps = append(l.PendingSteps, checkedSteps...)
                l.Mtx.Unlock()
            }
        }
    }
}
```

pkg/state/new_state.go

```
package state

import (
    "bytes"
    "encoding/gob"
    "github.com/hashicorp/raft"
    "io"
    "sync"
)

type ClusterFSM struct {
    Mtx   sync.RWMutex
    State ClusterState
}

type ClusterState struct {
    Inventory    map[string]map[string]interface{} `yaml:"inventory"
json:"inventory"`
    PendingSteps map[string][]Step                 `yaml:"pending_steps"
json:"pending_steps"`
}

type Step struct {
    Name   string `yaml:"name" json:"name"`
    Id     string `yaml:"id" json:"id"`
    Status string `yaml:"status" json:"status"`
}

func (c *ClusterFSM) Apply(log *raft.Log) interface{} {
    buff := bytes.NewBuffer(log.Data)
    dec := gob.NewDecoder(buff)

    c.Mtx.Lock()
    defer c.Mtx.Unlock()

    err := dec.Decode(&c.State)
    if err != nil {
       return err
    }

    return nil
}

func (c *ClusterFSM) Snapshot() (raft.FSMSnapshot, error) {
    var buff bytes.Buffer
    enc := gob.NewEncoder(&buff)

    c.Mtx.Lock()
    defer c.Mtx.Unlock()

    err := enc.Encode(c.State)
    if err != nil {
       return nil, err
    }

    return &ClusterStateSnapshot{Data: buff.Bytes()}, nil
}

func (c *ClusterFSM) Restore(snapshot io.ReadCloser) error {
    dec := gob.NewDecoder(snapshot)

    c.Mtx.Lock()
```

```
    err := dec.Decode(&c.State)
    c.Mtx.Unlock()
    if err != nil {
        return err
    }

    return nil
}

type ClusterStateSnapshot struct {
    Data []byte
}

func (c *ClusterStateSnapshot) Persist(sink raft.SnapshotSink) error {
    _, err := sink.Write(c.Data)
    if err != nil {
        _ = sink.Cancel()
        return err
    }
    return sink.Close()
}

func (c *ClusterStateSnapshot) Release() {}
```

pkg/state/state.go

```
package state

import (
    "context"
    "github.com/hashicorp/raft"
    "google.golang.org/protobuf/proto"
    "time"
)

type DetrintStateManagerServer struct {
    UnimplementedStateManagerServer
    Node *raft.Raft
}

func (s *DetrintStateManagerServer) UpdateStateSet(_ context.Context, req
*UpdateStateSetRequest) (*UpdateStateSetResponse, error) {
    bytes, err := proto.Marshal(&DetrintState{StateSet: req.GetNewSet()})
    if err != nil {
        return &UpdateStateSetResponse{
            Result:  OperationResult_ERROR,
            Message: err.Error(),
        }, nil
    }
    s.Node.Apply(bytes, 5*time.Second)

    return &UpdateStateSetResponse{Result: OperationResult_SUCCESS, Message:
"New state applied"}, nil
}
```

pkg/web/api.go

```
package web

import (
    "bytes"
```

```
    "detrint_v2/pkg/cloud"
    "detrint_v2/pkg/cluster"
    "detrint_v2/pkg/remote"
    "detrint_v2/pkg/state"
    "encoding/gob"
    "github.com/gin-gonic/gin"
    "github.com/hashicorp/raft"
    "github.com/rs/zerolog/log"
    "gopkg.in/yaml.v3"
    "io"
)

type ProvisionInput struct {
    AccessConfig  cloud.AccessConfig
    ServersConfig cloud.ServersConfig
    NetworkConfig cloud.NetworkConfig
}

func RegisterAPI(router *gin.Engine, cl *cluster.Cluster) {
    apiRouter := router.Group("/api")
    logger := log.With().Str("component", "api").Logger()

    clusterRouter := apiRouter.Group("/cluster")

    clusterRouter.GET("/fsm", func(c *gin.Context) {
       c.JSON(200, cl.ClusterState.State)
    })

    clusterRouter.GET("/local", func(c *gin.Context) {
       c.JSON(200, cl.LocalState)
    })

    clusterRouter.GET("/fsm/update", func(c *gin.Context) {
       var buff bytes.Buffer
       enc := gob.NewEncoder(&buff)
       err := enc.Encode(state.ClusterState{Inventory:
map[string]map[string]interface{}{
          "raft_node_1": {
             "Test 1": "A",
             "Test 2": "B",
          },
       }})
       if err != nil {
          logger.Error().Err(err).Msg("Failed to encode data")
          c.JSON(500, gin.H{"error": "failed to encode new data"})
          return
       }

       future := cl.Raft.Apply(buff.Bytes(), 0)
       err = future.Error()
       if err != nil {
          logger.Error().Err(err).Msg("Failed to apply data")
          c.JSON(500, gin.H{"error": "failed to apply data"})
          return
       }
    })

    clusterRouter.POST("/state/update", func(c *gin.Context) {
       header, err := c.FormFile("file")
       if err != nil {
          logger.Error().Err(err).Msg("Failed to get file from request")
          c.JSON(400, gin.H{"error": "failed to get file from request"})
          return
       }
```

```
        file, err := header.Open()
        if err != nil {
           logger.Error().Err(err).Msg("Failed to open provided file")
           c.JSON(500, gin.H{"error": "failed to open provided file"})
           return
        }

        content, err := io.ReadAll(file)
        if err != nil {
           logger.Error().Err(err).Msg("Failed to read file content")
           c.JSON(500, gin.H{"error": "failed to read file content"})
           return
        }

        var newState state.ClusterState
        err = yaml.Unmarshal(content, &newState)
        if err != nil {
           logger.Error().Err(err).Msg("Failed to unmarshal input data")
           c.JSON(500, gin.H{"error": "failed to unmarshal input data"})
           return
        }

        // Data verification
        for node := range newState.PendingSteps {
           existingIds := map[string]int8{}
           for i := range newState.PendingSteps[node] {
              id := newState.PendingSteps[node][i].Id
              _, ok := existingIds[id]
              if ok {
                 // ID duplication
                 logger.Error().Msg("Pending steps have id duplication")
                 c.JSON(400, gin.H{"error": "pending steps have id duplication"})
                 return
              }
              existingIds[id] = 0

              if len(newState.PendingSteps[node][i].Status) != 0 {
                 // Not-empty status
                 logger.Error().Msg("Pending steps have not-empty status")
                 c.JSON(400, gin.H{"error": "pending steps have not-empty
status"})
                 return
              }
           }
        }

        // Apply changes
        var buff bytes.Buffer
        enc := gob.NewEncoder(&buff)
        err = enc.Encode(newState)
        if err != nil {
           logger.Error().Err(err).Msg("Failed to encode data")
           c.JSON(500, gin.H{"error": "failed to encode new data"})
           return
        }

        future := cl.Raft.Apply(buff.Bytes(), 0)
        err = future.Error()
        if err != nil {
           logger.Error().Err(err).Msg("Failed to apply data")
           c.JSON(500, gin.H{"error": "failed to apply data"})
           return
        }
```

```
        c.JSON(200, newState)
    })

    clusterRouter.GET("/state/all", func(c *gin.Context) {
        leaderAddress, leaderId := cl.Raft.LeaderWithID()
        if len(leaderAddress) == 0 {
            leaderAddress = "-"
        }
        if len(leaderId) == 0 {
            leaderId = "-"
        }

        f := cl.Raft.GetConfiguration()
        err := f.Error()
        if err != nil {
            logger.Error().Err(err).Msg("Failed to get cluster configuration")
            c.JSON(500, gin.H{"error": "failed to get cluster configuration"})
            return
        }
        clusterConf := f.Configuration()

        c.JSON(200, gin.H{
            "leader_address": leaderAddress,
            "leader_id":      leaderId,
            "nodes_count":    len(clusterConf.Servers),
        })
    })

    clusterRouter.GET("/stats", func(c *gin.Context) {
        c.JSON(200, cl.Raft.Stats())
    })

    clusterRouter.GET("/config", func(c *gin.Context) {
        f := cl.Raft.GetConfiguration()
        err := f.Error()
        if err != nil {
            logger.Error().Err(err).Msg("Failed to get cluster configuration")
            c.JSON(500, gin.H{"error": "failed to get cluster configuration"})
            return
        }
        clusterConfig := f.Configuration()
        clusterConfigJson := struct {
            Servers []struct {
                Role    string `json:"role"`
                Id      string `json:"id"`
                Address string `json:"address"`
            } `json:"servers"`
        }{}

        for _, server := range clusterConfig.Servers {
            clusterConfigJson.Servers = append(clusterConfigJson.Servers, struct {
                Role    string `json:"role"`
                Id      string `json:"id"`
                Address string `json:"address"`
            }{Role: server.Suffrage.String(), Id: string(server.ID), Address:
string(server.Address)})
        }

        c.JSON(200, clusterConfigJson)
    })

    clusterRouter.POST("/add", func(c *gin.Context) {
        newPeer := struct {
            Id      string `json:"id"`
```

```
			Address string `json:"address"`
		}{}

		err := c.BindJSON(&newPeer)
		if err != nil {
			logger.Error().Err(err).Msg("Failed to parse input JSON")
			return
		}

		f := cl.Raft.AddVoter(raft.ServerID(newPeer.Id),
raft.ServerAddress(newPeer.Address), 0, 0)
		err = f.Error()
		if err != nil {
			c.JSON(500, gin.H{"error": "failed to add peer"})
			logger.Error().Err(err).Msg("Failed to add peer")
			return
		}
	})

	clusterRouter.POST("/install", func(c *gin.Context) {
		connData := remote.ConnectionData{}
		err := c.BindJSON(&connData)
		if err != nil {
			logger.Error().Err(err).Msg("Failed to parse input JSON")
			return
		}

		for i := 0; i < connData.RetryCount; i++ {

			conn, err := remote.Connect(&connData)
			if err != nil {
				logger.Error().Err(err).Msg("Failed to connect to remote host")
				return
			}

			clusterInfo := cl.GetInfo()
			conf, err := cluster.GenerateConfig(clusterInfo, &connData)
			if err != nil {
				logger.Error().Err(err).Msg("Failed to generate config")
				return
			}

			err = conn.CopyInstallFiles()
			if err != nil {
				logger.Error().Err(err).Msg("Failed to copy install files")
				return
			}

			err = conn.CopyConfig(conf)
			if err != nil {
				logger.Error().Err(err).Msg("Failed to copy config files")
				return
			}

			err = conn.BootstrapAgent()
			if err != nil {
				logger.Error().Err(err).Msg("Failed to bootstrap agent")
				return
			}

			results, err := cl.HealthCheck(conf)
			if err != nil {
				logger.Error().Err(err).Msg("Failed to check new server health")
				return
```

```
            }

            if results.Success {
               c.JSON(200, gin.H{"status": "success"})
               return
            }
         }

         c.JSON(200, gin.H{"status": "failed"})
      })

      clusterRouter.POST("/provision", func(c *gin.Context) {
         provisionInput := ProvisionInput{}
         err := c.BindJSON(&provisionInput)
         if err != nil {
            logger.Error().Err(err).Msg("Failed to parse input JSON")
            return
         }

         cl, err := cloud.NewClient(&provisionInput.AccessConfig)
         if err != nil {
            logger.Error().Err(err).Msg("Failed to create cloud client")
            return
         }

         if provisionInput.NetworkConfig.Exists {
            networkReady, err := cl.IsExist(&provisionInput.NetworkConfig)
            if err != nil {
               logger.Error().Err(err).Msg("Failed to check network config")
               return
            }

            if !networkReady {
               logger.Error().Err(err).Msg("Provided network config is not ready
for usage")
               return
            }
         } else {
            err := cl.PrepareNetwork(&provisionInput.NetworkConfig)
            if err != nil {
               logger.Error().Err(err).Msg("Failed to prepare network infra")
               return
            }
         }

         serversList, err := cl.PrepareInstances(&provisionInput.ServersConfig,
&provisionInput.NetworkConfig)
         if err != nil {
            logger.Error().Err(err).Msg("Failed to prepare instances infra")
            return
         }

         serversReady, err := cl.ConnectivityCheck(serversList)
         if err != nil {
            logger.Error().Err(err).Msg("Connectivity check failed")
            return
         }

         if !serversReady {
            err := cl.DeleteResources(provisionInput.ServersConfig.StackName)
            if err != nil {
               logger.Error().Err(err).Msg("Failed to delete servers resources")
               return
            }
```

```
			if !provisionInput.NetworkConfig.Exists {
				err := cl.DeleteResources(provisionInput.NetworkConfig.StackName)
				if err != nil {
					logger.Error().Err(err).Msg("Failed to delete network
resources")
					return
				}
			}
		}

		logger.Info().Bool("servers_ready", serversReady).Strs("servers",
serversList).Msg("Provisioning finished")
	})
}
```

pkg/web/static.go

```
package web

import (
	embedres "detrint_v2"
	"fmt"
	"github.com/gin-gonic/gin"
	"io/fs"
	"net/http"
)

func RegisterUI(r *gin.Engine) error {
	staticFS := fs.FS(embedres.WebUiRes)
	staticContent, err := fs.Sub(staticFS, "web/dist")
	if err != nil {
		return fmt.Errorf("failed to make FS for static content: %w", err)
	}

	r.StaticFS("/ui", http.FS(staticContent))

	return nil
}
```

ДОДАТОК Д

Ілюстративна частина

ІЛЮСТРАТИВНА ЧАСТИНА

РОЗРОБКА МЕТОДІВ І ПРОГРАМНИХ ЗАСОБІВ УПРАВЛІННЯ КОНФІГУРАЦІЯМИ ДЛЯ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ПРОЦЕСІВ РОЗГОРТАННЯ ТА МАСШТАБУВАННЯ ЕЛЕКТРОННИХ РЕСУРСІВ

Рисунок Д.1 – Титульний слайд

Рисунок Д.2 – Актуальність теми

Мета, об'єкт та предмет дослідження

- **Мета дослідження** - підвищення ефективності процесів розгортання та масштабування електронних ресурсів.
- **Об'єкт дослідження** - процеси управління конфігураціями при розгортанні та масштабуванні електронних ресурсів.
- **Предмет дослідження** - методи та засоби підвищення ефективності процесів розгортання та масштабування електронних ресурсів.

Рисунок Д.3 – Мета, об'єкт та предмет дослідження

Задачі дослідження

Відповідно до поставленої мети дослідження необхідно виконати такі завдання:

- провести аналіз існуючих методів підвищення ефективності процесів розгортання та масштабування електронних ресурсів;
- розробити метод і алгоритм для забезпечення високої доступності ПЗ для управління конфігураціями;
- розробити метод і алгоритм для автоматичної інсталяції програм-агентів на керовані сервери;
- розробити метод і алгоритм для автоматичного створення серверів з використанням хмарних провайдерів;
- розробити графічний інтерфейс користувача для створюваного ПЗ;
- розробити програмний застосунок для управління конфігураціями;
- провести тестування програмного застосунку;
- дослідити ефективність методу забезпечення високої доступності;
- розробити інструкцію користувача.

Рисунок Д.4 – Задачі дослідження

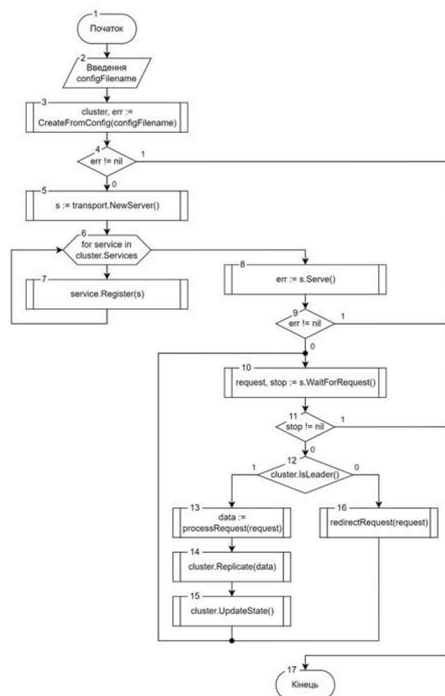
Рисунок Д.5 – Наукова новизна

Рисунок Д.6 – Практична цінність одержаних результатів

Порівняльний аналіз аналогів

Критерій	Ansible	Puppet	Chef	SaltStack	Detrint
Підтримка режиму високої доступності	-	-	-	+	+
Можливість працювати без встановлення агентів на сервери	+	-	-	-	-
Автоматична інсталяція агентів	+/-	-	-	-	+
Підтримка non-UNIX-подібних ОС в якості керуючого сервера	-	-	-	-	+
Підтримка роботи з хмарними провайдерами	+/-	+/-	+/-	+	+
Автономне підтримування бажаного стану системи	-	+	+	-	-
Підсумок	2	1,5	1,5	2	4

Рисунок Д.7 – Порівняльний аналіз аналогів



Метод і блок-схема алгоритму забезпечення високої доступності

Особливості роботи методу:

- синхронізація стану вузлів за допомогою алгоритму консенсусу;
- автоматична зміна лідера кластера в аварійній ситуації;
- розподілення задач по управлінню конфігураціями між доступними вузлами.

Рисунок Д.8 – Метод і блок-схема алгоритму забезпечення високої доступності

Метод і блок-схема алгоритму автоматичної інсталяції програм-агентів

Особливості роботи методу:

- генерація конфігурації нового вузла;
- автоматичне підключення до наявного кластера;
- перевірка стану вузла (health checks).

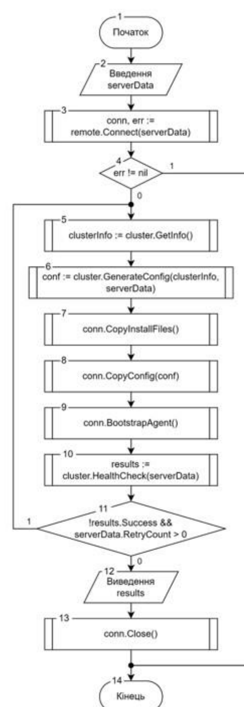


Рисунок Д.9 – Метод і блок-схема алгоритму автоматичної інсталяції програм-агентів

Метод і блок-схема алгоритму автоматичного створення серверів з використанням хмарних провайдерів

Особливості роботи методу:

- підготовка і перевірка мережевих ресурсів;
- підготовка і перевірка віртуальних машин або серверів;
- контроль створених ресурсів в аварійних ситуаціях.

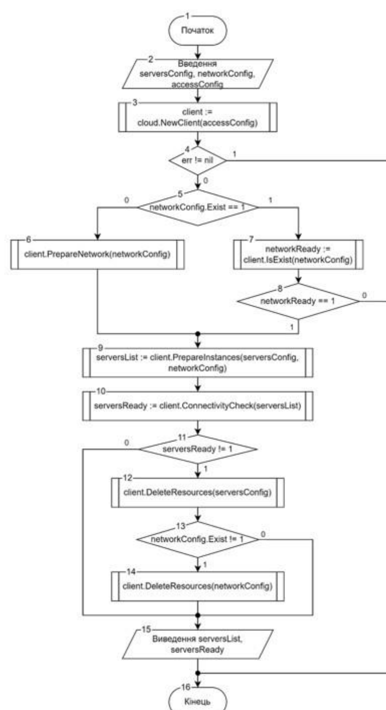


Рисунок Д.10 – Метод і блок-схема алгоритму автоматичного створення серверів з використанням хмарних провайдерів

Структура графічного інтерфейсу користувача

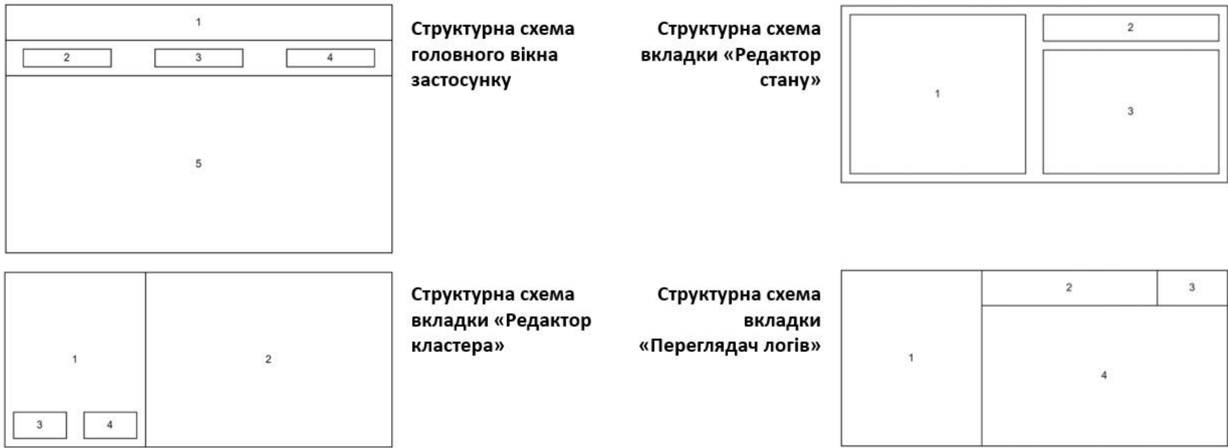


Рисунок Д.11 – Структура графічного інтерфейсу користувача

Тестування програмного забезпечення

Тестування не виявило проблем в основному функціоналі ПЗ, виконання усіх тест-кейсів було успішне

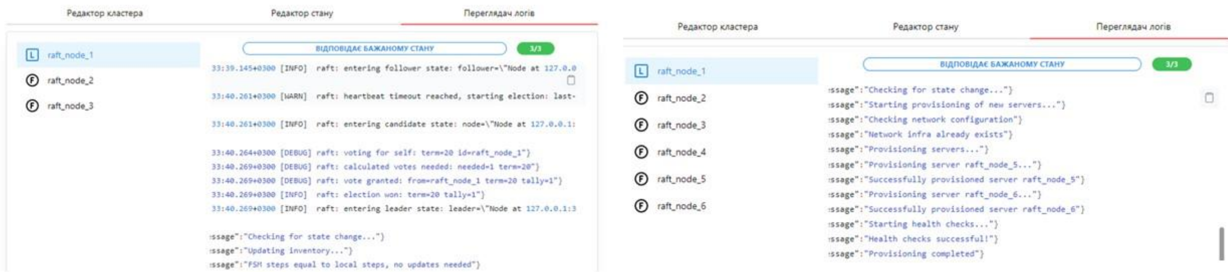
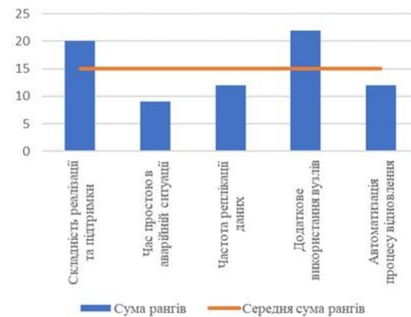


Рисунок Д.12 – Тестування програмного забезпечення

Дослідження ефективності розробленого методу забезпечення високої доступності

Номер	Фактор	Експерти					Δ_k	Δ_k^σ	$\left(\Delta_k^\sigma\right)^2$	M	q_k
		1	2	3	4	5					
1	Складність реалізації та підтримки	2	5	5	4	4	20	5	25	4	0,13
2	Час простою в аварійній ситуації	3	2	1	1	2	9	-6	36	1	0,33
3	Частота реплікації даних	1	3	2	3	3	12	-3	9	2	0,27
4	Додаткове використання вузлів	5	4	3	5	5	22	7	49	5	0,07
5	Автоматизація процесу відновлення	4	1	4	2	1	12	-3	9	3	0,2
	Сума	15	15	15	15	15	75	-	128	-	1



Апріорне ранжування показало, що метод реалізує найбільш важливі за оцінками експертів фактори, а по показникам RTO та RPO він показує себе краще за деякі інші популярні стратегії аварійного відновлення

Рисунок Д.13 – Дослідження ефективності розробленого методу забезпечення високої доступності

Економічна частина

- Згідно проведених досліджень рівень комерційного потенціалу розробки становить 40 балів, що свідчить про комерційну важливість проведення даних досліджень (рівень комерційного потенціалу розробки вищий середнього).
- Загальні витрати ЗВ на завершення науково-дослідної (науково-технічної) роботи та оформлення її результатів складає *ЗВ=659729,39 грн*
- Період окупності інвестицій, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки складає $T_{ок}$*=1,47 року*
- Можна зробити висновок про доцільність проведення науково-дослідної роботи за темою «Розробка методів і програмних засобів управління конфігураціями для підвищення ефективності процесів розгортання та масштабування електронних ресурсів».

Рисунок Д.14 – Економічна частина

Впровадження, апробація та публікації результатів роботи

Результати роботи **впроваджено** в ТОВ "ДСофтвер" в системі розгортання цифрової інфраструктури.

Також результати роботи доповідалися на:

- Міжнародній науково-практичній Інтернет-конференції «Електронні інформаційні ресурси: створення, використання, доступ» (Суми/Вінниця, 2022);
- LII Науково-технічній конференції факультету інформаційних технологій та комп'ютерної інженерії (Вінниця, 2023).

Основні результати досліджень опубліковано в 3 наукових працях, **у тому числі 1 стаття у фаховому виданні України**, 2 – у матеріалах конференцій.

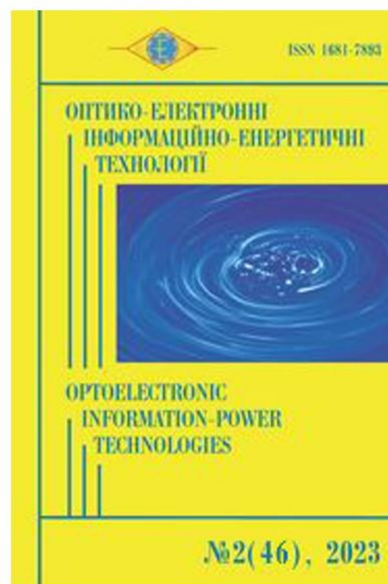


Рисунок Д.15 – Впровадження, апробація та публікації результатів роботи

Дякую за увагу

Рисунок Д.16 – Фінальний слайд