МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

кафедра «Інформаційні системи та мережі»

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи на тему:

Інформаційна система для впровадження GitOps методології у процес конфігурації серверів

Студента групи _____ІТ-41, Верхутіна, Д.Є_____
(шифр, прізвище та ініціали)

Керівник роботи _____ (__Андрій ВАСИЛЮК__)
Консультант _____ (_____)
_____ (_____)
Нормоконтроль _____ (__Андрій ВАСИЛЮК__)

Завідувач кафедри ІСМ _____ (**Дмитро ДОСИН**)
«_04__» __червня_____ **2025 р.**

ЛЬВІВ – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Інститут комп'ютерних наук та інформаційних технологій
Кафедра «Інформаційні системи та мережі»
Спеціальність 126 "Інформаційні системи та технології"
Перший (бакалаврський) рівень вищої освіти
ОПП "Інтелектуальні інформаційні системи"

«ЗАТВЕРДЖУЮ»
Завідувач кафедри ІСМ _____
«_04_» __червня____2025 р.

ЗАВДАННЯ

на бакалаврську кваліфікаційну роботу студента групи ІТ-41

Верхутіна Даніїла

(прізвище, ім'я, по батькові)

1. Тема роботи Інформаційна система для впровадження GitOps методології у процес конфігурації серверів

затверджена наказом по НУ «ЛП» від «_11_»__березня__2025р. №_866-4-08_____

2. Термін здачі студентом закінченої роботи ____30.05.2025р._____

3. Вихідні дані для роботи: ___ресурси глобальної мережі Інтернет, літературні джерела.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які належить розробити): аналіз існуючих систем для конфігурації з використанням GitOps методології, розробка концептуальної модель інформаційної системи для використання GitOps методології у процесі конфігурації серверів, вибір зручних, надійних, актуальних та оптимальних технологій і середовищ для розробки проєку, розробити інформаційну систему для впровадження GitOps методології у процес конфігурації серверів.

5. Перелік графічного матеріалу: дерево цілей, діаграми IDEF0, скріншоти роботи програми.

6. Перелік програмних продуктів, які належить використати в процесі розроблення роботи (проекту): AllFusion Process Modeler, VS Code, Ansible, Git, Redis, Postman, UTM.

7. Консультування роботи, із зазначенням розділів роботи

Розділ	Консультанти	Підпис, дата	
		завдання видав	завдання отримав

8. Дата, коли видано завдання _____24.02.2025 р._____

Керівник _____
(підпис)

Завдання отримав до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Етапи кваліфікаційної роботи	Термін виконання етапів роботи	Примітки
1	Огляд існуючих програмних рішень та аналіз літературних джерел	25.02-5.03.2025	Виконано
2	Системний аналіз об’єкта дослідження	6-25.03.2025	Виконано
3	Створення основної концепції системи	26.03-1.04.2025	Виконано
4	Пошук програмного рішення для розроблення системи	2-7.04.2025	Виконано
5	Розробка системи і проведення тестів щодо її роботи	8.04-10.05.2025	Виконано
6	Оформлення кваліфікаційної роботи	10-30.05.2025	Виконано

Студент _____
(підпис)

Керівник роботи _____
(підпис)

ЗМІСТ

ВСТУП

Актуальність теми:

Сьогодні GitOps вважається однією з найбільш популярних методологій керування інфраструктурою, особливо в середовищах, де активно використовується Kubernetes [1]. І це не дивно: підхід GitOps дає змогу досягти стабільності, передбачуваності та автоматичності в управлінні конфігурацією. Але попри його потенціал, є одна дуже помітна проблема - всі GitOps-інструменти і всі приклади впровадження стосуються виключно Kubernetes. Якщо ж говорити про звичайні сервери, де інфраструктура побудована без кластерів, без контейнеризації, а на основі звичайної операційної системи та конфігураційних інструментів на кшталт Ansible, то GitOps тут майже не застосовується.

Це парадоксально, бо сама ідея GitOps універсальна. Вона взагалі не залежить від Kubernetes. Її суть - у тому, що весь стан інфраструктури описується у Git-репозиторії, і всі зміни проходять через звичні для розробників процеси: пул реквести, код-рев'ю, історія комітів, контроль версій. Потім автоматичний агент або система застосовує ці зміни у продакшн. І це ідеально підходить для будь-якої інфраструктури. Але чомусь у реальному світі GitOps фактично монополізував Kubernetes, і всі інструменти, що називаються GitOps-сумісними, - ArgoCD, FluxCD, Jenkins X - орієнтовані лише на роботу з Kubernetes API. І тут виникає дуже велика прогалина: за межами кластерів ця методологія просто випадає з поля зору [2].

Про це прямо говорить Майкл Леван, інженер і ентузіаст Kubernetes, який багато пише про GitOps і DevOps:

"Realistically, whatever is in source control, GitOps would manage. The ultimate goal with GitOps is that the entire system is managed declaratively, not just the apps.

GitOps is pretty much a cleaner version of Configuration Management." [3].

І це дуже влучне зауваження. Ідея є, інструменти є, але ніхто не адаптує їх під серверне середовище, яке досі залишається актуальним для тисяч компаній.

Наприклад, багато інфраструктур в банківському секторі, державних установах, навіть в освітніх і наукових організаціях все ще побудовані на класичних серверах. Там активно використовується Ansible як основний інструмент для автоматизації [4]. Але зміни в таких середовищах все одно часто вносяться вручну або хаотично, через окремі конфігурації, які запускають вручну, без історії і контролю. І от саме тут GitOps міг би дати величезну перевагу - централізоване джерело правди, перевірка всіх змін через Git, автоматичне застосування змін, можливість відкотитись у будь-який момент, переглянути хто і що змінив.

Проєкт і полягає в тому, щоб реалізувати GitOps не в Kubernetes, а в класичному середовищі з Ansible. Це буде GitOps-агент, який відслідковує зміни в репозиторії, автоматично запускає Ansible-конфігурації, має логіку для валідації, обробки помилок, відстеження статусу. Це все зробить роботу з Ansible набагато надійнішою, безпечнішою і прозорішою, якраз за філософією GitOps.

За даними CNCF, ще у 2021 році тільки 27% компаній використовували GitOps, а вже в 2023 - більше ніж 50% команд з Kubernetes мають у себе GitOps-реалізацію [5]. Це показує, що методологія активно набирає популярність. Але цікаво інше: у звітах не згадується жоден інструмент, який би працював поза Kubernetes. Тобто по суті, методологія захопила лише одну нішу, а решта потенційно величезного ринку залишається порожньою.

Можна сказати, що впровадження GitOps для керування серверною конфігурацією - це наступний логічний крок у розвитку DevOps-підходів. Адже DevOps вже вийшов за межі просто CI/CD - він про надійність, автоматизацію, контроль, і все це GitOps дає просто з коробки. І якщо донести цю ідею до серверного середовища, особливо в поєднанні з Ansible, то можна досягти нової якості керування інфраструктурою.

Таким чином, тема не просто актуальна - вона знаходиться на стику технологій, які вже стали стандартом, і тих, які ще тільки мають відкрити свій потенціал. Це можливість показати, що GitOps - це не тільки про Kubernetes, що це не про моду, а

про інженерну культуру, яку можна і треба масштабувати на всі рівні інфраструктури. І інформаційна система - це крок у цьому напрямку.

Мета і задачі дослідження:

Розроблення інформаційної системи для впровадження GitOps методології у процес конфігурації серверів, яка може встановити потрібні сервіси на Linux сервер-агент, відслідковувати зміни Github репозиторію з Ansible проєктом, автоматично впроваджувати внесені зміни, зберігати інформацію про синхронізацію з репозиторієм та статуси процесів, надати можливість клієнтам по API звертатись для відстеження статусів та запуску синхронізації вручну.

Цілі роботи:

- Проаналізувати предметну область, відібравши та прочитавши відповідні літературні та інтернет- джерела.
- Дослідити реалізацію платформ-аналогів, врахувавши їх переваги та недоліки.
- Виконати системний аналіз проєкту, завдяки дереву цілей та діаграмі процесів.
- Зробити обґрунтований вибір мови програмування, технологій, підходів для максимально ефективної роботи програми та зручності процесу розробки.
- Реалізувати інформаційну систему, протестувати її, зробити висновки та методи покращення продукту.

Об’єкт дослідження:

Процес впровадження GitOps методології у процес конфігурації серверів.

Предмети дослідження:

Методи і засоби, які необхідні для провадження GitOps методології у процес конфігурації серверів.

Практичне значення одержаних результатів:

Реалізована інформаційна система має високе практичне значення, оскільки відкриває можливість впровадження GitOps-методології у сфері конфігурації

звичайних Linux-серверів, без залучення Kubernetes, що досі залишається основною платформою для GitOps-рішень. У той час як більшість сучасних інструментів GitOps орієнтовані виключно на контейнери та кластерні системи, запропоноване рішення дозволяє автоматизувати керування інфраструктурою, побудованою на Ansible, з використанням всіх переваг Git як єдиного джерела правди. Система вміє самостійно виявляти зміни у GitHub-репозиторії з Ansible-проєктом, автоматично застосовувати ці зміни на сервер-агент, встановлювати потрібні сервіси, зберігати детальну історію синхронізацій та їх статусів, а також надає зовнішній API, що дозволяє контролювати процес, ініціювати ручну синхронізацію та інтегруватись з іншими системами. Такий підхід дозволяє зменшити кількість помилок, пов’язаних із людським фактором, підвищити надійність процесів конфігурації, а також зробити інфраструктуру більш прозорою, передбачуваною та контрольованою. Він буде корисним як DevOps-інженерам, так і звичайним адміністраторам, які прагнуть перейти до більш структурованого, гнучкого та сучасного способу роботи з конфігурацією серверів, не змінюючи при цьому кардинально існуючу архітектуру інфраструктури.

РОЗДІЛ 1

Аналітичний огляд літературних та інших джерел

1.1. Основна концепція дослідження

Перші інструменти для автоматизації процесів у сфері керування серверною інфраструктурою були створені для того, щоб спростити адміністраторам та інженерам виконання рутинних задач, пов’язаних із налаштуванням, оновленням та моніторингом серверів. Проте з часом, у міру масштабування інфраструктур та впровадження DevOps-підходів, виникла потреба у ще більш структурованому способі роботи, зокрема, у впровадженні змін через систему контролю версій, з можливістю перегляду історії, контролю конфігурацій та автоматизації [6]. Саме на цю потребу відповідає методологія GitOps, яка активно застосовується в середовищах Kubernetes. Проте за межами Kubernetes, де використовується класична архітектура з фізичними або віртуальними Linux-серверами, GitOps досі не був належно реалізований. У таких середовищах адміністраторам доводилось запускати Ansible-конфігурації вручну, зберігати конфігурації локально або фрагментовано у різних системах, що ускладнювало контроль змін і створювало ризики помилок.

Рішення, яке розробляється в межах цього проєкту, покликане вперше застосувати GitOps підхід до конфігурації Linux-серверів на базі Ansible, автоматизувавши повний цикл: від виявлення змін у GitHub-репозиторії до застосування їх на сервері-агенті. Таке рішення дає змогу значно полегшити доступ до управління конфігурацією, підвищити рівень автоматизації, зменшити залежність від людського фактора та підвищити прозорість усіх процесів. Воно також забезпечує централізований контроль, можливість інтеграції з іншими сервісами через API, а також зберігає повну історію дій, дозволяючи швидко виявляти та відстежувати зміни.

Загальна мета цієї системи полягає в тому, щоб надати адміністраторам та інженерам зручний і надійний інструмент для керування інфраструктурою, який

базується на сучасних принципах GitOps, але адаптований до серверів без Kubernetes, що робить її унікальним та актуальним рішенням на сьогоднішньому ринку.

Інформаційна система для впровадження GitOps методології у процес конфігурації серверів також може включати низку додаткових функцій, які значно підвищують її цінність та зручність використання для системних адміністраторів і DevOps-інженерів. Серед таких функцій можна виділити:

- Можливість інтеграції з GitHub для автоматичного відстеження змін у репозиторії з Ansible-проєктом.
- Механізм тригерів для запуску синхронізації одразу після оновлення конфігурацій.
- Функціональність журналювання процесів збереження статусів синхронізації та помилок.
- REST API для взаємодії клієнтів з платформою.

Чому ж така система буде затребуваною та залишатиметься актуальною? Для цього можна виокремити кілька ключових причин:

Автоматизація - замість ручного запуску скриптів і перевірки стану серверів, система автоматично виконує ці дії при будь-яких змінах, тим самим заощаджуючи час спеціалістів та зменшуючи ризик людських помилок.

Централізація - адміністратори отримують єдиний інтерфейс для контролю всієї конфігурації серверів, що особливо важливо при масштабуванні інфраструктури.

Інтеграція з DevOps-циклами - рішення логічно вбудовується у сучасні практики CI/CD, дозволяючи впроваджувати інфраструктурні зміни разом із кодом застосунків, що забезпечує цілісність та контрольованість усіх змін.

Трасування та прозорість - завдяки збереженню історії синхронізацій та логів, користувачі можуть легко відслідковувати, коли і які зміни були застосовані.

Гнучкість та масштабованість - завдяки API можна будувати власні рішення на базі цієї системи, інтегруючи її з внутрішніми порталами або сторонніми сервісами.

Незалежність від Kubernetes - на відміну від більшості існуючих GitOps-інструментів, ця система не вимагає Kubernetes-кластера, що робить її ідеальним рішенням для компаній, які працюють із класичними віртуальними або фізичними Linux-серверами.

Актуальність і новизна - враховуючи стрімке зростання популярності GitOps у хмарних середовищах (згідно з опитуванням CNCF, понад 51% організацій використовують або планують використовувати GitOps у найближчі роки), розширення цієї методології на класичну серверну інфраструктуру є логічним та необхідним етапом еволюції [5].

Таким чином, подібна інформаційна система стає не просто черговим інструментом для автоматизації, а ключовим елементом сучасної IT-інфраструктури, що відповідає принципам безперервної інтеграції, прозорості та контрольованості змін, зберігаючи при цьому адаптивність до традиційних середовищ, де Kubernetes поки що не застосовується.

1.2. Основні перспективи GitOps у конфігурації серверів

Ця частина більш детально розкриє всі функції та особливості проєкту з впровадження GitOps методології у процес конфігурації серверів.

1.2.1 Централізоване управління конфігурацією через Git

Централізоване управління конфігурацією через Git - це одна з ключових ідей GitOps, яка переносить всю логіку управління серверною інфраструктурою в репозиторій Git [7]. Це означає, що вся інформація про те, які сервіси мають бути встановлені на серверах, які конфігурації потрібно застосувати, які зміни потрібно внести - усе зберігається в одному центральному місці, у Git-репозиторії з Ansible-проєктом.

Цей підхід дає змогу уникнути хаосу, пов’язаного з ручною конфігурацією серверів або підтримкою розрізнених сценаріїв. Адміністратору чи DevOps-інженеру

більше не потрібно пам’ятати, що саме він змінював на якому сервері - вся історія змін є у Git, із зазначенням автора, часу і суті коміту. Це також дуже зручно в командній роботі: кілька учасників можуть одночасно працювати з одним конфігураційним проєктом, не створюючи конфліктів або дублювань.

Таким чином, Git стає єдиним джерелом правди (single source of truth) для всієї інфраструктури. Це значно спрощує аудит, контроль якості, відкат до попереднього стану у разі помилки, а також спрощує процес навчання нових учасників у команді - достатньо дати їм доступ до репозиторію.

1.2.2. Автоматизація розгортання та оновлень інфраструктури

Ще одна ключова перспектива GitOps-підходу, яка безпосередньо реалізована у проєкті. Завдяки поєднанню Git та Ansible, зміни у конфігурації серверів перестають бути ручним або напівручним процесом і стають повністю автоматизованими. Як тільки зміни вносяться у Git-репозиторій, наприклад, додається новий сервіс або змінюється параметр існуючого, система автоматично відстежує цей коміт і запускає процес синхронізації з цільовими серверами.

Цей процес виключає необхідність вручну підключатися до серверів, запускати Ansible-конфігурації або турбуватися про актуальність налаштувань. Оновлення відбуваються одразу після внесення змін у репозиторій, що значно підвищує швидкість реагування на вимоги бізнесу, безпекові оновлення чи інші необхідності. Окрім цього, автоматизація мінімізує людський фактор, зменшується ймовірність помилок, випадкових пропусків або нестабільних налаштувань.

У перспективі це означає, що керування інфраструктурою може здійснюватися взагалі без прямої взаємодії з серверами, а усе через Git, що перетворює розгортання на просту дію типу `git push`.

1.2.3. Автоматизація розгортання та оновлень інфраструктури

Це одна з найсильніших сторін GitOps-підходу, яка реалізована у системі як фундаментальний принцип. У традиційному сценарії адміністрування серверів часто буває складно визначити, хто, коли і чому вніс ту чи іншу зміну в конфігурацію. Але при використанні Git як єдиного джерела правди кожна зміна в інфраструктурі фіксується у вигляді коміту, що містить всю історію, автора, час і опис дії. Таким чином, замість неструктурованого хаосу отримуємо контрольований, аудитований та прозорий процес.

Ключовим інструментом тут виступають pull request-и (PR). Зміни вносяться не напряму, а через запити на злиття, що дозволяє провести код-рев'ю. Інші члени команди можуть переглянути зміни, перевірити їх на відповідність вимогам, виявити потенційні помилки або неочевидні наслідки ще до того, як зміни потраплять до продакшену. Це забезпечує високий рівень якості змін, підвищує безпеку й сприяє обміну знаннями в команді.

У реалізації проєкту використовується єдиний сервер-агент, який під'єднується до GitHub репозиторію, відслідковує зміни та ініціює синхронізацію конфігурації на цільовому сервері. Це забезпечує централізовану точку управління, що значно спрощує моніторинг і діагностику. До того ж, кожна дія агента логуватиметься - тобто в системі буде журнал станів, де можна подивитись, коли відбувалась остання синхронізація, які саме зміни були застосовані, чи пройшов процес успішно, а якщо ні - що пішло не так.

Завдяки такій архітектурі прозорість не є просто побічним ефектом, а стає основною функціональністю. Усі дії мають логічне пояснення, фіксуються, аналізуються і, при необхідності, можуть бути відтворені чи відкочені. Це надзвичайно важливо для команд DevOps, адже дозволяє підтримувати стабільність, швидко реагувати на інциденти, і найголовніше, не втрачати контроль над тим, що саме відбувається у їх системі..

1.2.4. Наявність API для керування синхронізацією конфігурацій

API закладає фундамент для гнучкої інтеграції, автоматизації та масштабування. Вже реалізовано три базові точки доступу: перевірка обраного стану синхронізації, ініціація ручного тригера синхронізації та отримання журналу синхронізацій. Цього достатньо для MVP версії, щоб ззовні без необхідності прямого підключення до сервера отримувати актуальну інформацію про конфігурацію і запускати оновлення за потреби.

У перспективі це відкриває шлях до інтеграції з UI, де можна буде візуально переглядати стан серверів, конфігурацій, бачити лог-файли, статус останнього оновлення або повідомлення про помилки. Це значно полегшить життя адміністраторам, які не хочуть або не мають змоги постійно працювати з терміналом.

Крім того, така архітектура дозволяє під'єднувати CLI-клієнт, щоб напряму взаємодіяти з системою через консольні команди, наприклад, sync-now, status, або logs, і отримувати потрібну інформацію миттєво, не заходячи на сам сервер.

У більш розширеному варіанті API може також надати можливість аналізу конфігурації та стану системи: перевірка активних процесів, споживання ресурсів, активні служби, доступні оновлення - усе це без потреби SSH-доступу до сервера. Таким чином, навіть маючи лише шаблон і оболонку зараз, у майбутньому система може еволюціонувати у повноцінний централізований інтерфейс для адміністрування серверів, що поєднує простоту, контроль і гнучкість.

1.2.5. Підвищення рівня безпеки інфраструктури

Насамперед, його забезпечує рольовий доступ (RBAC) до всіх аспектів управління конфігурацією делегований GitHub, що виступає в ролі єдиного центру авторизації. Це означає, що всі зміни проходять через Git-процеси, такі як Pull Request-и, які передбачають обов'язкове рев'ю змін перед їх прийняттям. Відсутність окремих внутрішніх систем авторизації спрощує управління доступом. Всі права визначаються ролями GitHub-акаунтів або команд, які вже використовуються в межах

організації. Таким чином, немає потреби створювати окремі облікові записи або вручну налаштовувати списки дозволів у різних системах.

Ще однією перевагою є те, що агент синхронізації розташовується безпосередньо у внутрішній мережі, де знаходяться сервери, що конфігуруються. Це повністю виключає необхідність відкриття портів або прямого мережевого доступу ззовні. Усі дії виконуються за принципом "pull", де агент самостійно ініціює перевірку змін у Git-репозиторії, і, в разі потреби, застосовує їх до серверів. Така модель виключає зовнішні підключення і робить інфраструктуру менш вразливою до атак.

Також важливо, що немає необхідності надавати адміністраторам прямий доступ до серверів або передавати їм будь-які паролі, приватні ключі чи інші секрети. Усі зміни відбуваються виключно через контрольовані зміни коду в Git, а доступ до них регламентується політиками GitHub. Аутентифікація здійснюється через Personal Access Tokens, що дозволяє уникнути зберігання чутливих даних у відкритому вигляді.

До того ж, завдяки логуванню всіх подій, пов’язаних із синхронізацією станів, кожен застосований коміт має повну історію: коли, ким і з яких змін він був застосований. Це створює високий рівень прозорості, і в разі помилки можна швидко визначити її джерело, повернутись до стабільного стану або провести аудит дій.

У майбутньому можна додати ще кілька корисних речей: наприклад, вбудовану перевірку змін на відповідність певним політикам безпеки (типу CIS або SOC2), можливість безпечно працювати з секретами (через Vault або подібні рішення), інтеграцію з MFA, і ще кращий аудит з детальною інформацією по всіх діях. Також було б класно розвинути API та додати окремий інтерфейс для зручної роботи з ним. Це дозволило б автоматизувати ще більше речей без втрати контролю.

1.3. Аналіз відомих програмних рішень

Перш ніж створювати або впроваджувати власну систему, завжди варто звернути увагу на існуючі рішення, які вже встигли зарекомендувати себе у сфері. Це не лише дозволяє уникнути повторення чужих помилок, а й дає змогу краще зрозуміти, які підходи є ефективними, а що, навпаки, може створювати зайву складність. Це особливо актуально, адже йдеться про реалізацію GitOps-підходу до управління серверною інфраструктурою напряму, що сьогодні стрімко розвивається й має багато готових інструментів.

У цьому розділі розглянуто три найбільш помітні й впливові продукти у світі GitOps: ArgoCD, FluxCD та SpaceLift. Вони різні за своєю структурою, архітектурою, а також підходом до автоматизації й інтеграцій, однак усі вони базуються на одній ключовій ідеї - контроль за станом інфраструктури має здійснюватися через систему контролю версій, переважно Git. Кожне з цих рішень по-своєму цікаве. Наприклад, ArgoCD орієнтований на Kubernetes, має зручний UI та дуже активну спільноту. FluxCD, навпаки, більш легкий і модульний, без надлишкового інтерфейсу, що надає більше гнучкості. А от SpaceLift - це приклад комерційного, більш «дорослого» рішення, яке об’єднує GitOps із політиками доступу, Terraform'ом, інтеграцією з GitHub, а також можливістю ручної модерації змін.

Для мене було важливо не просто побіжно згадати ці інструменти, а розібратись у їхньому підході до синхронізації з Git, зручності інтеграції в CI/CD процеси, наявності або відсутності UI, можливостей API, безпеки та гнучкості управління доступом. Усе це буде розглянуто в рамках аналізу кожного окремого інструменту, а наприкінці зробити коротке порівняння, яке дозволить виділити найсильніші сторони кожного та зрозуміти, які ідеї можуть бути корисними при реалізації проєкту.

1.3.1. ArgoCD

ArgoCD - це, без перебільшення, одне з найвідоміших і найпоширеніших рішень у сфері GitOps [8]. Його розроблено спеціально для роботи з Kubernetes, і саме на

цьому він фокусується. Одразу після першого знайомства помітно, що продукт орієнтований на реальне використання в командах: є зручний веб-інтерфейс, можна швидко підключити репозиторій з описами маніфестів і майже одразу отримати розгортання в кластері.

Кожен застосунок (Application) в ArgoCD - це окрема одиниця розгортання, яка має свій стан, історію змін, журнали подій і можливість порівнювати актуальний стан у кластері з бажаним (записаним у Git). Також дуже зручно, що можна бачити, які саме файли змінилися, хто автор коміту, і за потреби вручну підтвердити розгортання. Це значно спрощує аналіз ситуацій, коли щось іде не так, особливо у командній роботі [9].

Однією з головних переваг ArgoCD є його інтеграція з Git як єдиним джерелом правди. Уся конфігурація системи (у тому числі, самих застосунків) також може зберігатися в Git, що дозволяє досягнути повної відповідності з GitOps-підходом. Додатково, ArgoCD підтримує роботу з Helm, Kustomize, Ksonnet, plain YAML та іншими інструментами, що робить його досить універсальним.

Ще одна річ, яка часто відрізняє ArgoCD від інших інструментів - це саме його графічний інтерфейс [10]. Він зручний, інформативний, з візуальним представленням залежностей ресурсів, де можна одразу побачити статус кожного поду, сервісу або job'и. У деяких випадках це замінює необхідність заходити у кластер взагалі. З досвіду інтеграції ArgoCD у банківські проєкти UI ArgoCD став вирішальним фактором також і для розробників, оскільки всю потрібну інформацію про стан сервісів на інфраструктурі без зайвих кроків з SSH-ключами, VPN, доступів, знань Linux і т.п. Навіть для почтаківців саме використання ArgoCD є інтуїтивно зрозумілим та легким.

Однак ArgoCD має і свої особливості. Наприклад, щоб мати повний контроль, часто доводиться писати додаткову конфігурацію: ролі доступу, правила синхронізації, гачки (hooks) тощо. Ще один момент, він постійно моніторить стан, і якщо щось у кластері було змінено вручну (тобто не через Git), то система вважатиме

це «дрейфом» і може відновити стару версію. Це добре з точки зору цілісності, але потребує певної культури роботи з кластером.

З точки зору безпеки, ArgoCD інтегрується з SSO-провайдерами (наприклад, GitHub, GitLab, OIDC), що дозволяє контролювати доступи централізовано. Крім того, можна точно вказати, хто і які застосунки має право створювати або змінювати, що дуже корисно в середовищі з декількома командами.

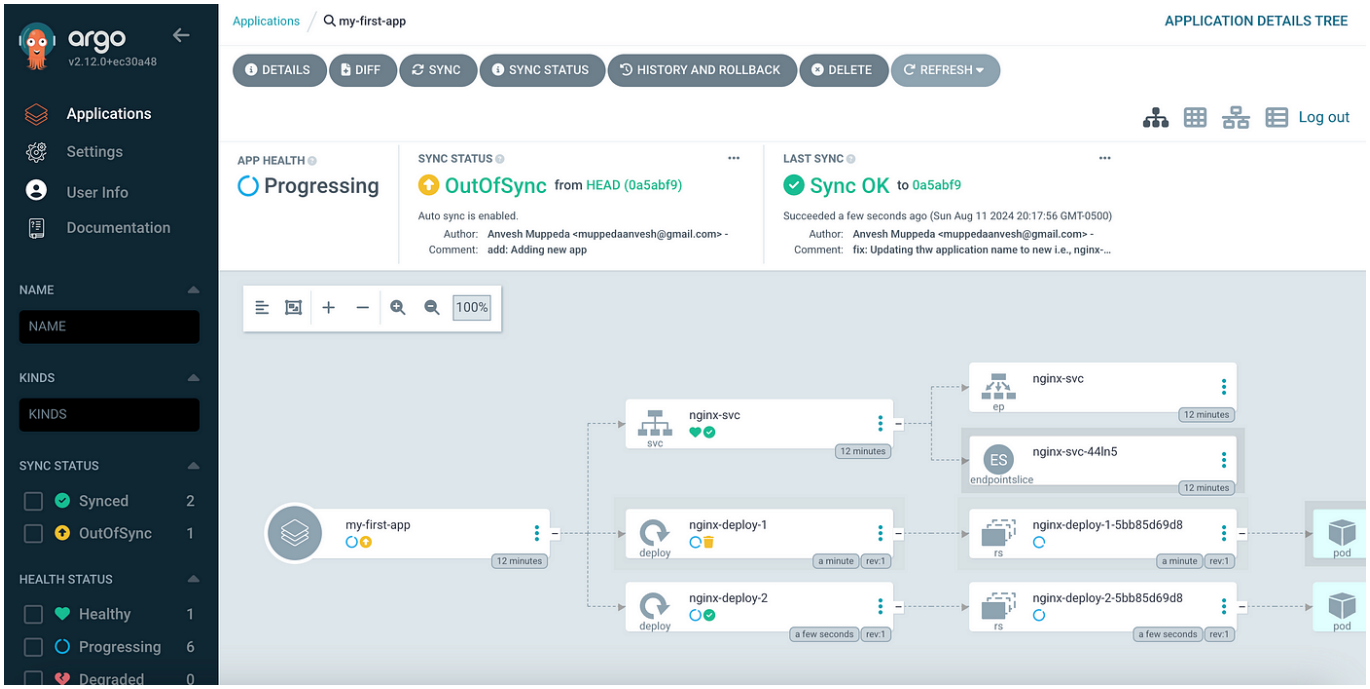

Рис. 1.1 Вигляд UI-інтерфейсу ArgoCD

У підсумку, ArgoCD - це стабільний, надійний і зручний інструмент для управління Kubernetes-інфраструктурою в стилі GitOps. Він має багато можливостей, хоча для менш досвідчених користувачів деякі з них можуть потребувати часу на освоєння. Але саме завдяки своїй функціональності, активному розвитку та зростаючій спільноті ArgoCD залишається одним із найпопулярніших варіантів для GitOps-управління на Kubernetes.

1.3.2. FluxCD

FluxCD - це ще один дуже потужний і популярний інструмент GitOps, який часто згадують поряд з ArgoCD. Але якщо ArgoCD більше про інтерфейс і централізоване

керування з UI, то FluxCD - це про максимальну автоматизацію, розширюваність і інтеграцію в CI/CD-ланцюжок. Flux здається більш "інженерним" продуктом без зайвого інтерфейсу, але з великою кількістю гнучких можливостей під капотом.

Flux працює як набір контролерів, які встановлюються у Kubernetes-кластер і безперервно стежать за репозиторієм з конфігурацією. Щойно в Git вносяться зміни - Flux реагує й оновлює кластер. При цьому контролери розділені по ролях: є окремо для роботи з Git, з Helm-чартами, з образами контейнерів тощо. Завдяки цьому можна будувати досить гнучкі пайплайни, підлаштовуючи поведінку під свої задачі [11].

У нього немає вбудованого графічного інтерфейсу, як у ArgoCD, але є підтримка інтеграції з Weave GitOps UI, простим веб-інтерфейсом для базового управління [12]. Хоча здебільшого все керується або через Git, або через CLI. І якщо чесно, це змушує тримати руку на пульсі, бо ти більше взаємодієш з системою через команди та зміни коду у репозиторії.

Що стосується безпеки, Flux дуже добре інтегрується з Kubernetes RBAC і працює тільки з тим, що дозволено у межах ролі сервіс-аккаунта. Додатково, для автентифікації з Git (особливо GitHub/GitLab) він використовує токени або SSH-ключі, і важливо, що все це зберігається як Kubernetes-секрети, а не просто в якихось конфігураційних файлах. Ще один момент, Flux не вимагає прямого доступу до кластеру для розробників. Достатньо зміни в Git - і все розгортається або оновлюється автоматично.

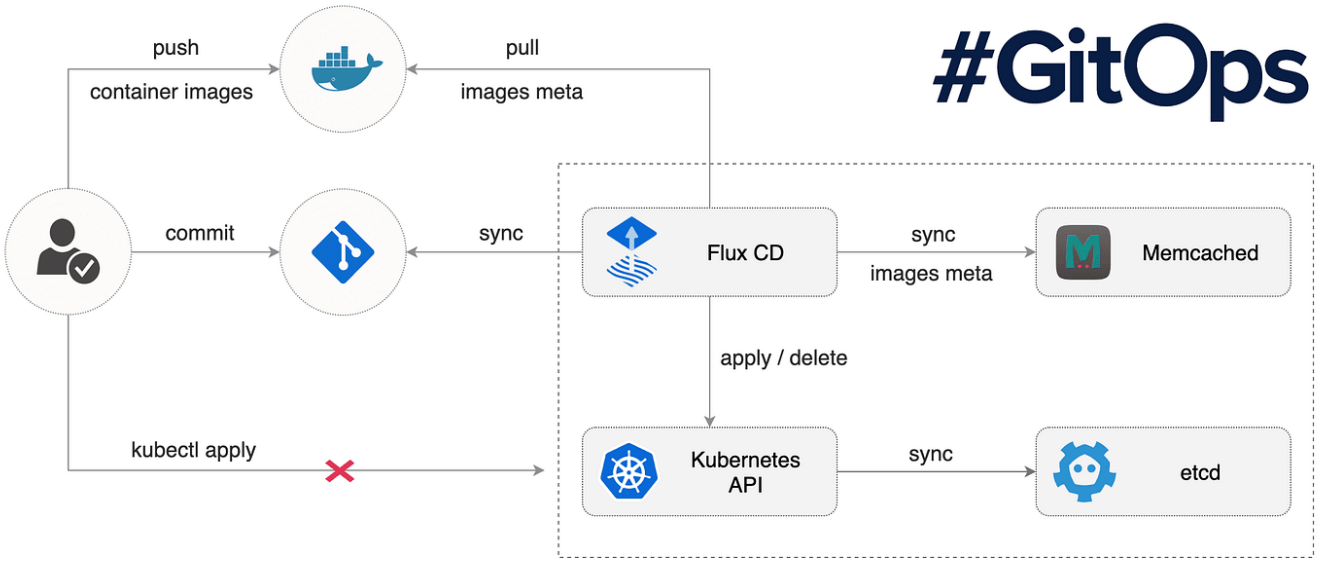

Рис. 1.2 Схематичне відображення роботи FluxCD

Ще одна сильна сторона Flux - це підтримка Image Update Automation. Він може автоматично перевіряти нові версії Docker-образів у реєстрі та створювати Pull Request'и до Git з оновленням тегів. Тобто є така собі "самооновлювана" система, де навіть версії образів можна тримати під контролем Git, не оновлюючи вручну [13].

Проте, якщо дивитись чесно, то для початку Flux трохи важчий в освоєнні, ніж ArgoCD. Через відсутність повноцінного GUI, спочатку складно зрозуміти, що саме він робить у певний момент часу, особливо якщо щось не працює. Але з часом, коли розбираєшся в структурі і логіці, починаєш цінувати саме гнучкість і модульність. Все працює як годинник - непомітно, але точно [14].

1.3.3 Spacelift

Spacelift - це сучасна платформа для управління інфраструктурою як кодом, яка підтримує різні інструменти, зокрема Terraform, Pulumi, CloudFormation, Kubernetes, а нещодавно й Ansible. Хоча Spacelift часто згадується у контексті GitOps, його не можна назвати класичною GitOps-платформою на кшталт ArgoCD чи FluxCD. Проте, він пропонує низку функцій, які дозволяють реалізувати елементи GitOps-підходу,

поєднуючи їх із можливостями розширеної автоматизації, контролю доступу та аналітики.

Одна з головних особливостей Spacelift - це глибока інтеграція з Git-репозиторіями. Вся конфігурація зберігається у Git, а зміни ініціюють автоматичні перевірки або запуск інфраструктурних "runs". Також платформа підтримує політики попереднього перегляду (preview plans), ручні затвердження змін, та фіксовані "gatekeeper"-механізми, що дозволяють команді повністю контролювати процес внесення змін [15].

Ще одним ключовим аспектом є безпека: Spacelift не потребує відкритого доступу до серверів, адже агент працює у захищеному середовищі, а аутентифікація та RBAC реалізовані через GitHub, GitLab або інші VCS-провайдери. Це дозволяє централізовано керувати правами доступу без необхідності роздавати окремі ключі чи паролі.

Spacelift також має потужне API, що відкриває шлях до подальшої автоматизації: від запуску "runs" вручну, до інтеграції з внутрішніми UI або CLI-клієнтами. Завдяки цьому платформа підходить не лише для DevOps-команд, а й для великих організацій, де потрібен контроль, гнучкість та масштабованість.

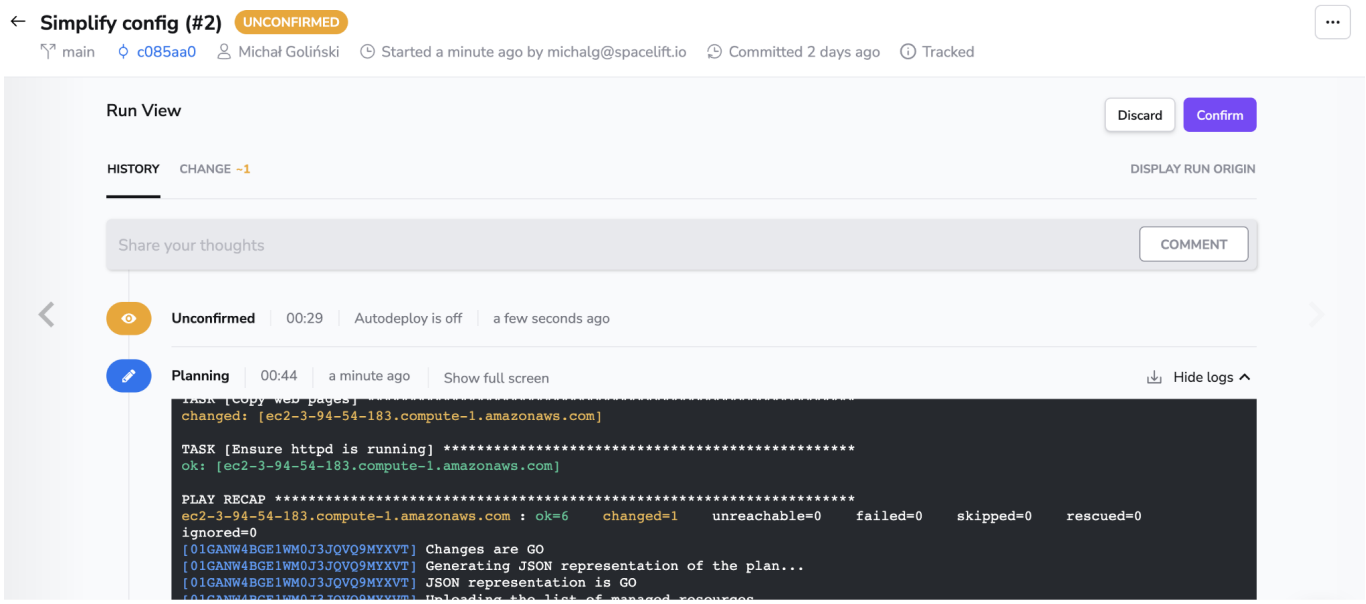

Рис. 1.3 Запуск Ansible playbook у Spacelift

Важливо зазначити, що Spacelift - комерційний продукт, який надає обмежену безкоштовну версію, але повноцінне використання передбачає оплату. Це створює певні обмеження для невеликих команд або освітніх ініціатив, які шукають безкоштовні рішення.

З точки зору мого проєкту, навіть попри наявність підтримки Ansible, Spacelift виступає радше як більш загальна платформа для IaC-процесів з GitOps-елементами, ніж як вузькоспеціалізований GitOps-агент для серверної конфігурації. Планована система, на відміну від Spacelift, зосереджується саме на автоматизованому конфігуруванні серверів із використанням Ansible у чистому GitOps-форматі з легковажним агентом, простим API та нативною інтеграцією в уже існуючі середовища без потреби у складних зовнішніх сервісах чи підписках [16].

1.3.4 Порівняльна характеристика

У поданій нижче таблиці 1.1 можна знайти згруповану порівняльну характеристику інформаційному системи роботи з наступними аналогами: ArgoCD, FluxCD, Spacelift.

Таблиця 1.1

Порівняльна характеристика з аналогами

Критерій / Рішення	ArgoCD	FluxCD	Spacelift	Розроблювана система
GitOps-підхід	Так, класичний	Так, класичний	Частковий GitOps	Так, чистий GitOps з легким агентом
Підтримка Ansible	Ні	Часткова (через додаткові плагіни)	Так (вбудована підтримка Ansible)	Так, нативна підтримка
Призначення	K8s-кластери	K8s-кластери	IaC платформа загального призначення	Конфігурація серверів через Ansible

Продовження табл 1.1

Цільова платформа	Kubernetes	Kubernetes	Cloud / IaC	Будь-які сервери (віртуальні, bare-metal, локальні)
Простота запуску	Відносно складний (потрібен K8s)	Відносно складний (потрібен K8s)	SaaS або складна інфраструктура	Максимально простий запуск: агент + Git
RBAC	Через Kubernetes	Через Kubernetes	Через GitHub / GitLab	Через GitHub (Git-права), не потрібно нічого налаштовувати
Потреба у відкритому доступі	Так, доступ до кластерів	Так, доступ до кластерів	Частково (через webhook або агент)	Ні, агент самостійно перевіряє оновлення з Git, без зовнішніх з'єднань
Ціна	Безкоштовно	Безкоштовно	Комерційна ліцензія	Вільне використання, без підписок
Можливість інтеграції з API	Обмежена інтеграція з API	Є, через Flux CLI / API	Так, потужний API	Є API (основа вже готова), можливість додавання UI/CLI
UI	Вбудований	Інші pen-source проєкти	Вбудований UI	Можлива інтеграція
Надійність та безпека	Висока, але потребує ручного налаштування	Висока, але потребує ручного налаштування	Висока, централізоване RBAC	Висока, без паролів/ключів, агент працює у внутрішній мережі серверів

Усі три розглянуті рішення мають потужні можливості, проте жодне з них не орієнтоване безпосередньо на GitOps-конфігурацію серверів через Ansible.

ArgoCD і FluxCD - чудові інструменти для управління Kubernetes-кластерами, проте їхнє використання для серверної конфігурації (особливо з Ansible) вимагає складних обхідних рішень. Вони більше підходять для інфраструктури, побудованої на Kubernetes.

Spacelift має вбудовану підтримку Ansible та пропонує більш гнучку платформу для IaC-процесів, але залишає за собою платний бар'єр, складну архітектуру, і його складно вбудувати у простіші сценарії типу "два сервери і один агент".

На цьому тлі проєкт вирізняється тим, що створений саме для таких сценаріїв: легкий, автономний, з нативною підтримкою Ansible, без потреби у прямому доступі до серверів, і при цьому повністю слідує GitOps-філософії. Він не конкурує з великими платформами, а закриває нішу простої та безпечної автоматизації серверної конфігурації, де великі рішення виглядають як "гармати по горобцях".

Більше того, цей підхід може стати новим способом взаємодії з серверами, що не потребує прямого SSH-доступу, складних ключів або ручної аутентифікації. Користувач просто працює з Git-репозиторієм, а агент у внутрішній мережі автоматично синхронізує зміни - це інтерфейс, який зрозумілий кожному розробнику.

У майбутньому проєкт може розширюватись:

- На інші інструменти керування конфігурацією (наприклад, Chef, Puppet, Terraform).
- Інтегруватися з популярними DevOps-інструментами (CI, секрет-менеджерами, моніторингом), додати підтримку різних хмарних провайдерів.
- І навіть стати основою для повноцінної GitOps-платформи в контексті Platform Engineering, коли команда має єдину точку входу для керування інфраструктурою, зрозумілу, безпечну і зручну.

Таким чином, проєкт не лише вирішує реальну проблему, а й має потенціал еволюціонувати у щось значно більше - GitOps-орієнтовану платформу нового покоління.

Висновок до розділу 1

У межах даного розділу було досліджено сучасні підходи до конфігурації інфраструктури, зокрема зосереджено увагу на GitOps як перспективному напрямку автоматизації керування серверами. Проаналізовано ключові принципи GitOps:

зберігання конфігурацій у Git, автоматизацію оновлень, прозорість змін, можливість розширення через API, а також підвищену безпеку завдяки роботі в закритому середовищі без прямого доступу до серверів.

Також проведено порівняльний аналіз популярних інструментів, таких як ArgoCD, FluxCD і Spacelift. Було визначено, що хоча всі вони мають потужні функції, жодне з них не є ідеальним рішенням саме для GitOps-підходу до конфігурації серверів з використанням Ansible. Більшість з них орієнтовані або на Kubernetes, або на більш комплексні інфраструктурні сценарії з високим порогом входу.

У такому контексті запропонований у межах роботи проєкт демонструє свою актуальність і потенціал: він вирішує конкретну задачу просто, безпечно та ефективно. Проєкт відкриває можливості нового способу взаємодії з серверам без необхідності прямого доступу, з інтуїтивною Git-логікою, і здатністю масштабуватися у повноцінну GitOps-платформу в контексті сучасного Platform Engineering.

Проведений аналіз підтвердив потребу у створенні нового інструменту, орієнтованого саме на GitOps-конфігурацію серверів, який буде зрозумілим розробникам, легко інтегруватиметься у наявні процеси та забезпечуватиме високу безпеку без ускладнень для команди.

РОЗДІЛ 2

Системний аналіз об’єкта дослідження

Системний аналіз - це метод, що використовується для дослідження складних систем з метою глибшого розуміння їхньої структури, функцій та взаємозв’язків між компонентами. Він широко застосовується в інженерії, бізнесі, управлінні, IT та інших сферах, де важливо враховувати численні чинники, що впливають на ефективність роботи системи.

Основна ідея системного аналізу полягає в поетапному підході: спочатку відбувається збір інформації про систему та її елементи, потім формується чітке визначення проблеми й цілей. Далі розглядаються можливі шляхи вирішення проблеми, порівнюються альтернативи, обирається найоптимальніший варіант і формується стратегія його реалізації.

Один з ключових інструментів у системному аналізі - моделювання. Воно дозволяє створити спрощене уявлення про систему (у вигляді математичних чи графічних моделей), яке допомагає зрозуміти її логіку, виявити слабкі місця та підготувати її до реальних умов експлуатації. Графічні схеми, блок-діаграми та інші візуалізації полегшують сприйняття та аналіз системних процесів.

У контексті мого проєкту системний аналіз дозволив глибше оцінити проблематику конфігурації серверів через Git, виявити обмеження наявних рішень, а також сформувати бачення майбутнього інструменту, який вирішує цю проблему просто та безпечно. Аналіз системи охопив не лише технічні компоненти, а й способи взаємодії між учасниками процесу, а також потенціал інтеграції з іншими DevOps-інструментами в межах єдиної GitOps-платформи.

2.1. Дерево цілей

Дерево цілей є одним із базових методів системного аналізу, що використовується для структурування цілей дослідження або проєкту. Це інструмент, який дозволяє чітко візуалізувати ієрархію цілей, від загальної мети до конкретних

задач і дій. Завдяки йому аналітики можуть краще зрозуміти логіку побудови системи, взаємозв’язки між її компонентами та визначити найбільш ефективні шляхи досягнення бажаних результатів.

У класичному вигляді дерево цілей - це графічна структура, що нагадує дерево у прямому значенні: верхівка (корінь дерева) - це головна ціль або стратегічне завдання, яке проект або система повинні реалізувати. Від неї відходять гілки, підцілі (тактичні завдання), які у свою чергу деталізуються до конкретних задач, дій або функцій. У результаті формується чітка багаторівнева структура, що демонструє логіку досягнення основної мети через реалізацію сукупності проміжних результатів.

Дерево цілей виконує кілька важливих функцій:

- Візуалізація логіки проєкту, змогу побачити, як окремі дії та етапи сприяють досягненню загальної мети;
- Планування, яке дозволяє послідовно визначити етапи реалізації, розподілити ресурси та визначити пріоритетність завдань;
- Засіб комунікації, який сприяє узгодженню бачення проєкту між різними учасниками, від технічних спеціалістів до менеджменту.

Конструювання дерева цілей зазвичай починається з формулювання загальної мети, яка повинна бути максимально узагальненою, але конкретною, вимірюваною та досяжною. Далі ця мета розкладається на тактичні підцілі, кожна з яких відповідає за окремий аспект реалізації головної мети. Наступний рівень деталізації - це операційні цілі, які вже включають в себе конкретні задачі, дії або функції системи, що безпосередньо виконуються на практиці.

Дерево цілей тісно пов’язане з побудовою дерева проблем, яке застосовується на попередньому етапі системного аналізу. Спочатку ідентифікуються ключові проблеми системи, їх причинно-наслідкові зв’язки та впливи, після чого ці проблеми трансформуються в цілі, які вже структуруються у вигляді дерева [17].

Такий підхід широко використовується у проєктному менеджменті, інженерії, державному управлінні, IT-сфері, розробці програмного забезпечення та інших

галузях, де важливо забезпечити цілісне бачення розвитку складних систем. Він допомагає не лише сформулювати стратегію, але й перевести її у конкретний план дій з чіткими орієнтирами та критеріями успіху.

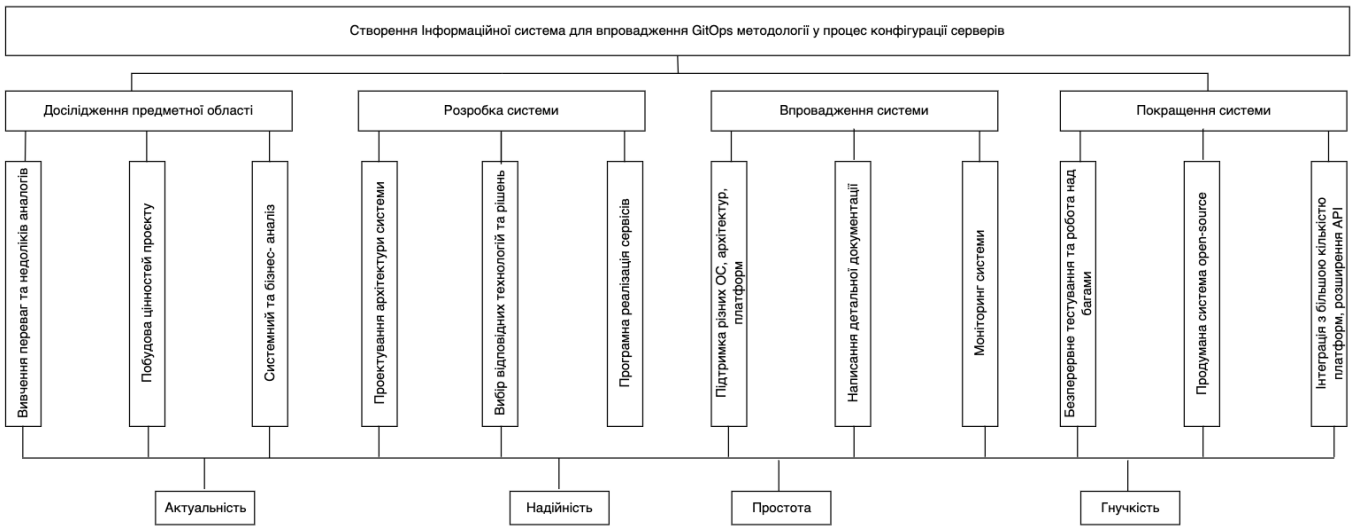

Рис. 2.1 Дерево цілей

Цілі:

1. Дослідження предметної області

Мета цього етапу - сформувати глибоке розуміння проблемного простору, користувацьких потреб та існуючих рішень.

Вивчення переваг та недоліків аналогів:

Аналіз існуючих інструментів, які вирішують подібні завдання. Визначення сильних і слабких сторін конкурентів.

Побудова цінностей проєкту:

Визначення ключових цінностей і переваг, які має надавати проєкт цільовій аудиторії.

Системний та бізнес-аналіз:

Оцінка потреб, вимог до функціональності та архітектури системи. Побудова бізнес-моделі.

2. Розробка системи

Ця ціль охоплює технічну сторону створення продукту - від архітектури до програмної реалізації.

Проєктування архітектури системи:

Побудова гнучкої та масштабованої структури програмного забезпечення.

Вибір відповідних технологій та рішень:

Застосування принципу "best-of-breed" - обираються найбільш підходящі та сучасні технології для реалізації кожної підсистеми.

Програмна реалізація сервісів:

Написання коду, розгортання та первинне тестування основних компонентів системи.

3. Впровадження системи

Мета - зробити продукт готовим до використання в реальному середовищі з підтримкою різних платформ.

Підтримка різних ОС, архітектур, платформ:

Забезпечення кросплатформенності та адаптивності програмного забезпечення.

Написання детальної документації:

Створення зрозумілих інструкцій для користувачів та розробників.

Моніторинг системи:

Впровадження інструментів для контролю стабільності, продуктивності та швидкого виявлення збоїв.

4. Покращення системи

Фокус цього етапу - забезпечення довготривалої актуальності та адаптації системи до нових вимог.

Безперервне тестування та робота над багами:

Організація CI/CD-процесів, усунення помилок, оптимізація продуктивності.

Продумана система open-source:

Відкритість до участі спільноти, структурована розробка з урахуванням зовнішніх вкладень.

Інтеграція з більшою кількістю платформ, розширення API:

Збільшення сумісності з іншими системами, підтримка нових сценаріїв використання.

Критерії:

- Актуальність - відповідність сучасним вимогам.
- Надійність - стабільність роботи системи.
- Простота - зручність у використанні.
- Гнучкість - легкість адаптації до змін.

2.2. Метод аналітичної ієрархії

Вибір архітектури системи:

Під час системного аналізу постає важливе завдання - визначити, на якій архітектурі будуватиметься система. Це рішення є критичним, оскільки саме архітектура впливає на подальший розвиток проєкту: наскільки легко буде масштабувати систему, підтримувати її, оновлювати та адаптувати до змін. У моїй роботі було розглянуто три сучасні варіанти: монолітну, мікросервісну та serverless-архітектуру.

Монолітна архітектура - це коли вся система реалізована як одне ціле: зручно запускати і деплоїти, але важко масштабувати або змінювати окремі частини без впливу на все інше. Мікросервісна - це підхід, де система розділена на незалежні частини (сервіси), кожна з яких виконує свою функцію. Це гнучко, але складніше в реалізації й підтримці. Serverless підхід ще далі відходить від традицій: розробник просто пише функції, а все інше (сервери, масштабування, хостинг) бере на себе хмарна платформа. Це дуже зручно, але іноді не вистачає контролю і може бути дорого залежно від навантаження [18, 19].

Кожна з архітектур має свої плюси та мінуси, і зробити вибір "на око" - це не той підхід, який би відповідав цілям системного підходу. Щоб ухвалити обґрунтоване рішення, вирішено скористатися методом аналітичної ієрархії (MAI). Це один із

класичних системних методів, який допомагає порівнювати альтернативи між собою за певними критеріями.

Чотири основні критерії: актуальність, надійність, простота реалізації та гнучкість. Саме ці характеристики найкраще відображають очікування від майбутньої системи, а також відповідають сформованому дереву цілей. Далі наведено сам процес аналізу: від побудови матриць парних порівнянь до підрахунку ваг і вибору найбільш відповідної архітектури.

Побудова MAI:

1. Альтернативи:
   - Моноліт.
   - Мікросервіси.
   - Serverless.
2. Критерії:
   - Актуальність.
   - Надійність.
   - Простота.
   - Гнучкість.

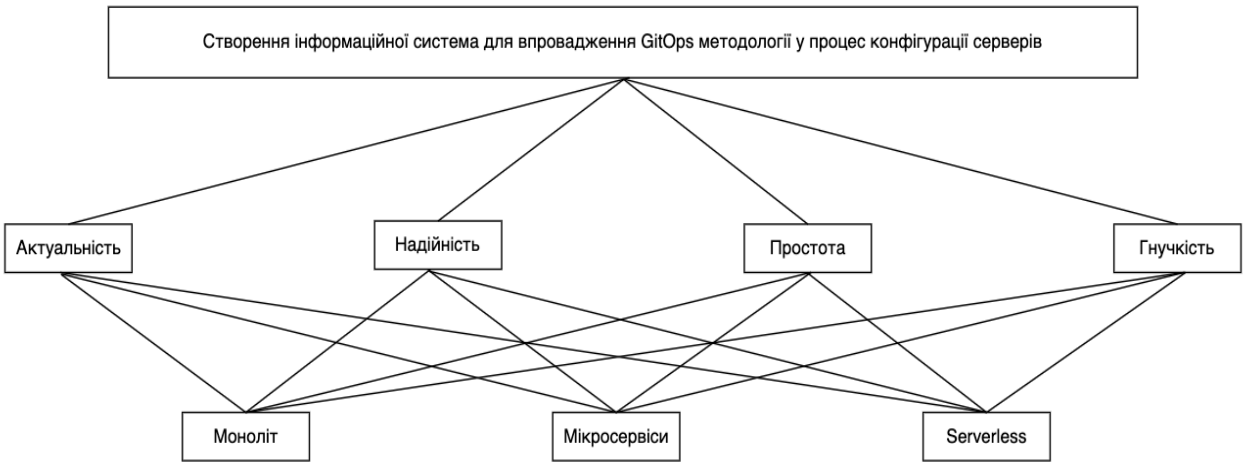

Рис. 2.2 Ієрархія MAI з вибору архітектури

3. Матриця парних порівнянь критеріїв:

Оцінки за шкалою Сааті є наступними: 1 - однаково важливо, 3 - трохи важливіше, 5 - набагато важливіше, 0.5 - менш важливо і т.д. Кртиерії були парно оцінені наступним чином:

Таблиця 2.1

Матриця парних порівнянь критеріїв

	Актуальність	Надійність	Простота	Гнучкість
Актуальність	1	2	3	2
Надійність	0.5	1	2	1
Простота	1/3	0.5	1	0.5
Гнучкість	0.5	1	2	1

Обчислюємо локальні ваги наступним чином (за приклад опису дій візьмемо критерій “Актуальність”):

1. Знаходимо суми всіх стовпців окремо (“Актуальність” = 2.33).
2. Нормалізуємо матрицю способом ділення значення у кожному рядку стовпчика (“Актуальність - Актуальність” = 1/2.33 = 0.43).
3. Обчислюємо локальні ваги (пріоритети) шляхом обчислення середніх арифметичних кожного рядка після нормалізації (Актуальність: (0.429 + 0.444 + 0.375 + 0.444) / 4 = 0.423).

Таблиця 2.2

Вага критеріїв

Критерій	Вага
Актуальність	0.423
Надійність	0.227
Простота	0.122
Гнучкість	0.227

Так само оцінюємо всі альтернативи за критеріями та знаходимо їх вагу. Отримуємо наступні оцінки та результати:

Таблиця 2.3

Матриця парних порівнянь альтернатив за критерієм “Актуальність”

	Моноліт	Мікросервіси	Serverless
Моноліт	1	1/3	2
Мікросервіси	3	1	4
Serverless	0.5	0.25	1

Таблиця 2.4

Нормалізація та вага альтернатив за критерієм “Актуальність”

	Моноліт	Мікросервіси	Serverless	Середнє (вага)
Моноліт	0.222	0.209	0.286	0.239
Мікросервіси	0.667	0.633	0.571	0.624
Serverless	0.111	0.158	0.143	0.137

Таблиця 2.5

Матриця парних порівнянь альтернатив за критерієм “Надійність”

	Моноліт	Мікросервіси	Serverless
Моноліт	1	0.5	3
Мікросервіси	2	1	4
Serverless	1/3	0.25	1

Таблиця 2.6

Нормалізація та вага альтернатив за критерієм “Надійність”

	Моноліт	Мікросервіси	Serverless	Середнє (вага)
Моноліт	0.3	0.286	0.375	0.320
Мікросервіси	0.6	0.571	0.5	0.557
Serverless	0.1	0.143	0.125	0.123

Таблиця 2.7

Матриця парних порівнянь альтернатив за критерієм “Простота”

	Моноліт	Мікросервіси	Serverless
Моноліт	1	3	2
Мікросервіси	1/3	1	0.5
Serverless	0.5	2	1

Таблиця 2.8

Нормалізація та вага альтернатив за критерієм “Простота”

	Моноліт	Мікросервіси	Serverless	Середнє (вага)
Моноліт	0.546	0.5	0.571	0.539
Мікросервіси	0.182	0.167	0.143	0.164
Serverless	0.273	0.333	0.286	0.297

Таблиця 2.9

Матриця парних порівнянь альтернатив за критерієм “Гнучкість”

	Моноліт	Мікросервіси	Serverless
Моноліт	1	0.5	1/3
Мікросервіси	2	1	2
Serverless	3	0.5	1

Таблиця 2.10

Нормалізація та вага альтернатив за критерієм “Гнучкість”

	Моноліт	Мікросервіси	Serverless	Середнє (вага)
Моноліт	0.167	0.25	0.1	0.172
Мікросервіси	0.333	0.5	0.6	0.478
Serverless	0.5	0.25	0.3	0.350

І останнім кроком обчислюємо загальні ваги для остаточного визначення вибору архітектури:

Таблиця 2.11

Загальні ваги

Архітектура	Актуальність (0.422)	Надійність (0.225)	Простота (0.121)	Гнучкість (0.225)	Загальна вага
Моноліт	0.239 × 0.422 = 0.101	0.320 × 0.225 = 0.072	0.539 × 0.121 = 0.065	0.172 × 0.225 = 0.039	0.277
Мікросервіси	0.624 × 0.422 = 0.263	0.557 × 0.225 = 0.125	0.164 × 0.121 = 0.020	0.478 × 0.225 = 0.108	0.516
Serverless	0.137 × 0.422 = 0.058	0.123 × 0.225 = 0.028	0.297 × 0.121 = 0.036	0.350 × 0.225 = 0.079	0.201

Отже, за результатами методу аналітичної ієрархії було обрано архітектуру проєкту, де безсумнівне перше місце зайняла мікросервісна архітектура.

2.3. Конкретизація функціонування системи

Методологія IDEF0 є однією з найпоширеніших структурних методик для моделювання функціональних аспектів складних систем. Вона використовується для опису, аналізу та документації функцій системи, а також для виявлення зв’язків між цими функціями та елементами зовнішнього середовища. Основна ідея IDEF0 полягає в тому, що будь-яку систему можна уявити як набір взаємопов’язаних функцій, кожна з яких має певні входи, виходи, механізми виконання та правила управління.

Особливістю IDEF0 є її здатність представляти систему у вигляді ієрархії діаграм, починаючи від найзагальнішого рівня (контекстної діаграми), що демонструє загальну мету функціонування, і до дедалі глибшого рівня деталізації, де кожна функція розкривається на підфункції. Такий підхід дозволяє поступово переходити від загального уявлення до конкретних процесів і механізмів без втрати цілісності сприйняття.

У графічному представленні IDEF0 кожна функція зображується у вигляді прямокутного блоку. Цей блок зв’язується зі стрілками, які мають чітко визначену

семантику: вхідні дані, вихідні результати, керуючі впливи та механізми виконання. Завдяки цьому забезпечується структурований, логічний і водночас інтуїтивно зрозумілий опис функціонування будь-якої складної системи.

Застосування IDEF0 є особливо доцільним на етапах аналізу, проєктування та вдосконалення інформаційних систем, оскільки дозволяє виявити не лише функціональні зв’язки всередині системи, а й залежності від зовнішніх факторів. Це дає змогу створити прозору і зрозумілу модель, яка буде корисною як для розробників, так і для стейкхолдерів, що беруть участь у створенні чи експлуатації системи [20].

Для інформаційної системи використано програму “AllFusion Process Modeler”, що має чітко викладений та доступний функціонал для того, щоб зручно та швидко побудувати IDEF0 діаграми. Спочатку контекстна діаграми.

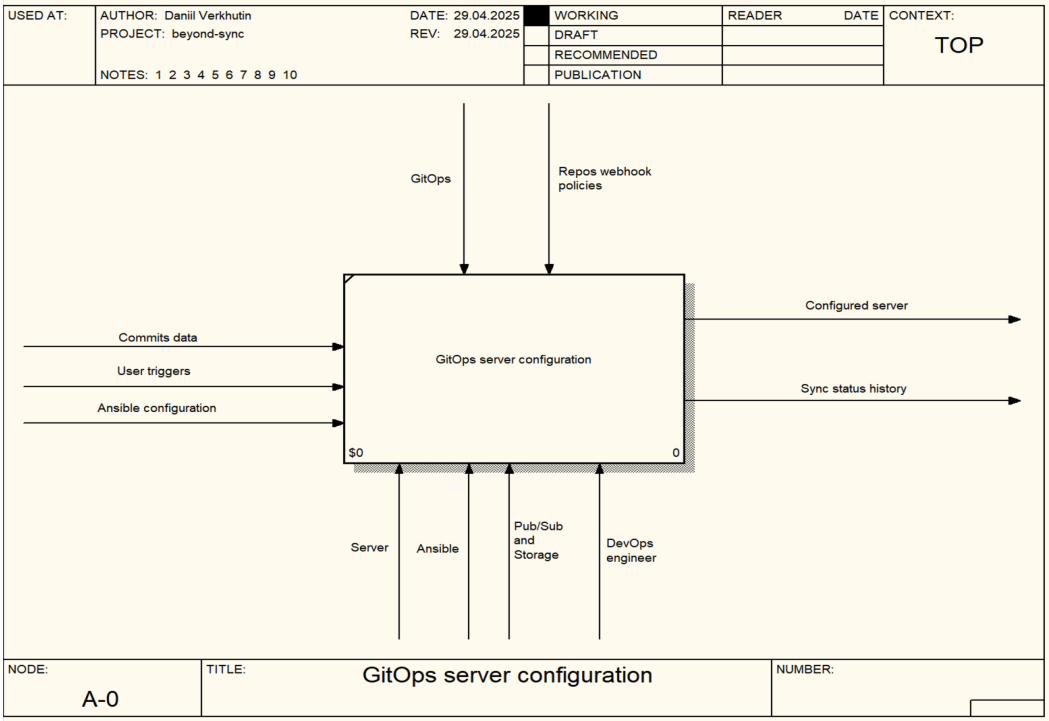

Рис. 2.3 Контекстна діаграма IDEF0

На цій діаграмі можна побачити наступне:

- Вхідні дані (зліва від основної функції системи):

- Дані змін у коді: інформація про зміни з репозиторію.
- Тригери користувача: ручні тригери для запуску синхронізації.
- Конфігурація Ansible: декларативна конфігурація серверу, яку ми відповідне імплементуємо.

- Вихідні результати (праворуч):
  - Сконфігурований сервер/сервери.
  - Збереження синхронізації в історію.
- Керуючі впливи (згори):
  - GitOps методологія.
  - Політики запитів у репозиторіях.
- Механізми виконання (знизу):
  - Сервер/сервери, які конфігуруються системою.
  - Ansible, що використовується для декларативної конфігурації.
  - Pub/Sub модель для спілкування між мікросервісами та сховище синхронізацій у Redis.
  - DevOps інженер - людина, що користується системою.

Після побудови контекстної діаграми, яка відображає систему в найзагальнішому вигляді, наступним важливим етапом моделювання за методологією IDEF0 є декомпозиція. Вона полягає у послідовному розкритті кожної функції в деталях, шляхом створення нових діаграм, що показують, з яких підфункцій складається кожна основна функція. Такий підхід дозволяє поступово занурюватися в структуру системи та аналізувати її логіку більш точно й повно.

Декомпозиція - це своєрідне “розшарування” загальної картини на простіші, більш зрозумілі елементи. Кожен блок на верхньому рівні може мати дочірню діаграму, де він подається як окрема система з внутрішніми процесами, що виконують одну загальну функцію. Ці дочірні діаграми, у свою чергу, можуть бути ще більше деталізовані, якщо це необхідно для досягнення чіткого розуміння функціонування всієї системи.

У моделюванні за IDEF0 прийнято говорити про рівні декомпозиції:

- Контекстний рівень (рівень A-0) - найвищий рівень абстракції, який описує систему загалом як одну функцію.
- Перший рівень декомпозиції (A0) - деталізує головну функцію на кілька основних підфункцій, що разом забезпечують роботу всієї системи.
- Наступні рівні (A1, A2, A3 тощо) - з підфункцій, у свою чергу, може бути деталізована ще більше, до рівня, на якому кожен процес буде чітко визначеним, зрозумілим і керованим.

Кількість рівнів декомпозиції залежить від складності системи та мети аналізу. У простих системах достатньо одного-двох рівнів, у складних можуть бути потрібні кілька ступенів розкладання. Головна мета - це дійти до рівня, на якому кожна функція є елементарною і не потребує подальшого пояснення.

Завдяки декомпозиції можна легко визначити, які саме функції виконує система, як між собою взаємодіють її окремі компоненти, які дані передаються між ними, які механізми залучені до реалізації кожної задачі та які правила впливають на виконання функцій. Це сприяє глибшому розумінню не лише логіки роботи системи, але й можливих слабких місць, що особливо важливо при оптимізації, тестуванні або впровадженні нових рішень.

Отже, в “AllFusion Process Modeler” пройдено наступний етапу, де обрано декомпозицію контекстної діаграми та програма зробила відповідний шаблон, з якого побудовано діаграму першого рівня декомпозиції.

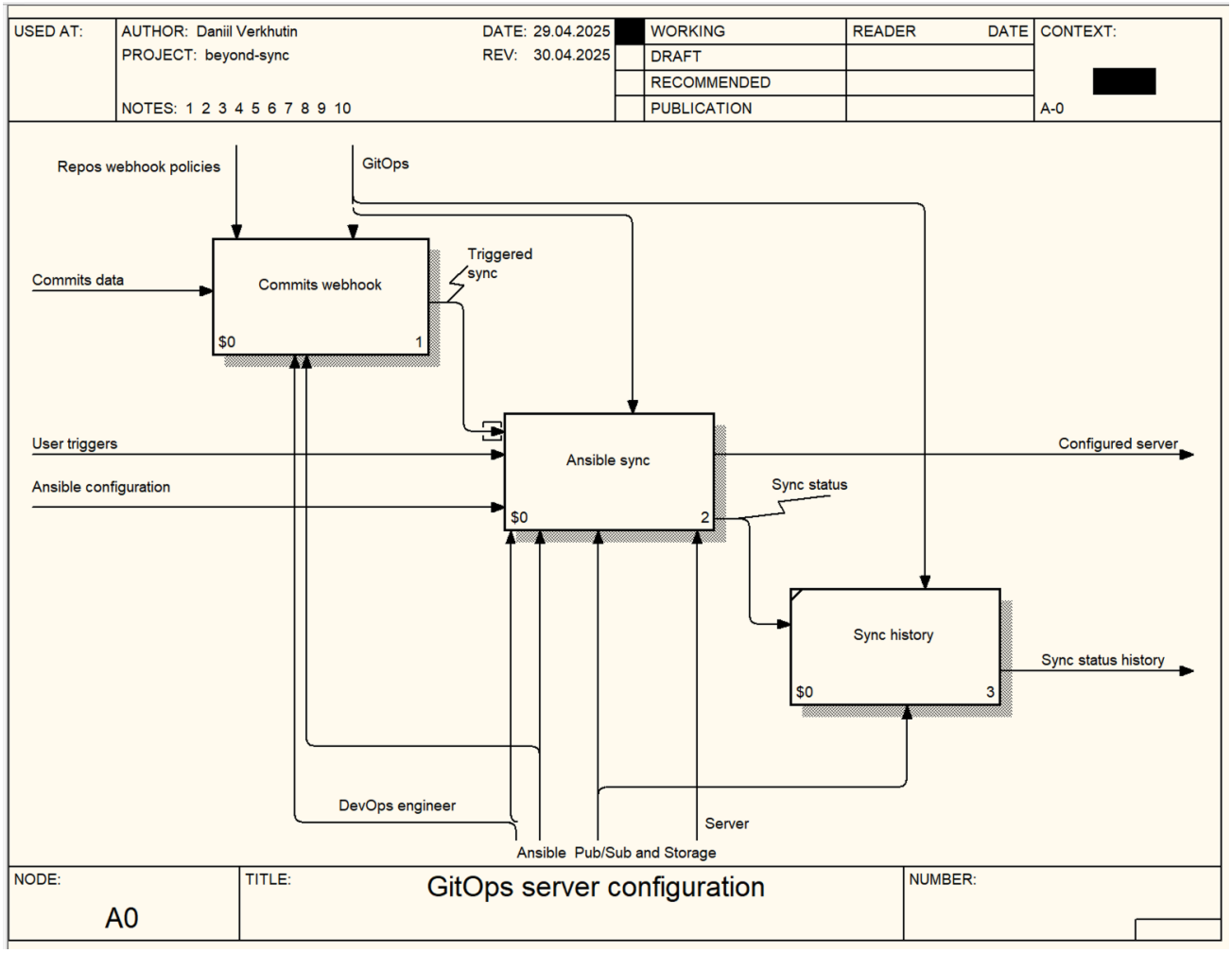

Рис. 2.4 Декомпозиція першого рівня контекстної діаграми IDEF0

Тут основний процес поділяється на три підпроцеси, з яких він і складається. Ці процеси пов’язуються з усіма складовими контекстної діаграми, але також додаються елементи що поєднують ці процеси між собою.

Далі кожен з процесів першого рівня декомпозиції розкладаємо на другий рівень декомпозиції, деталізуючи процеси у них.

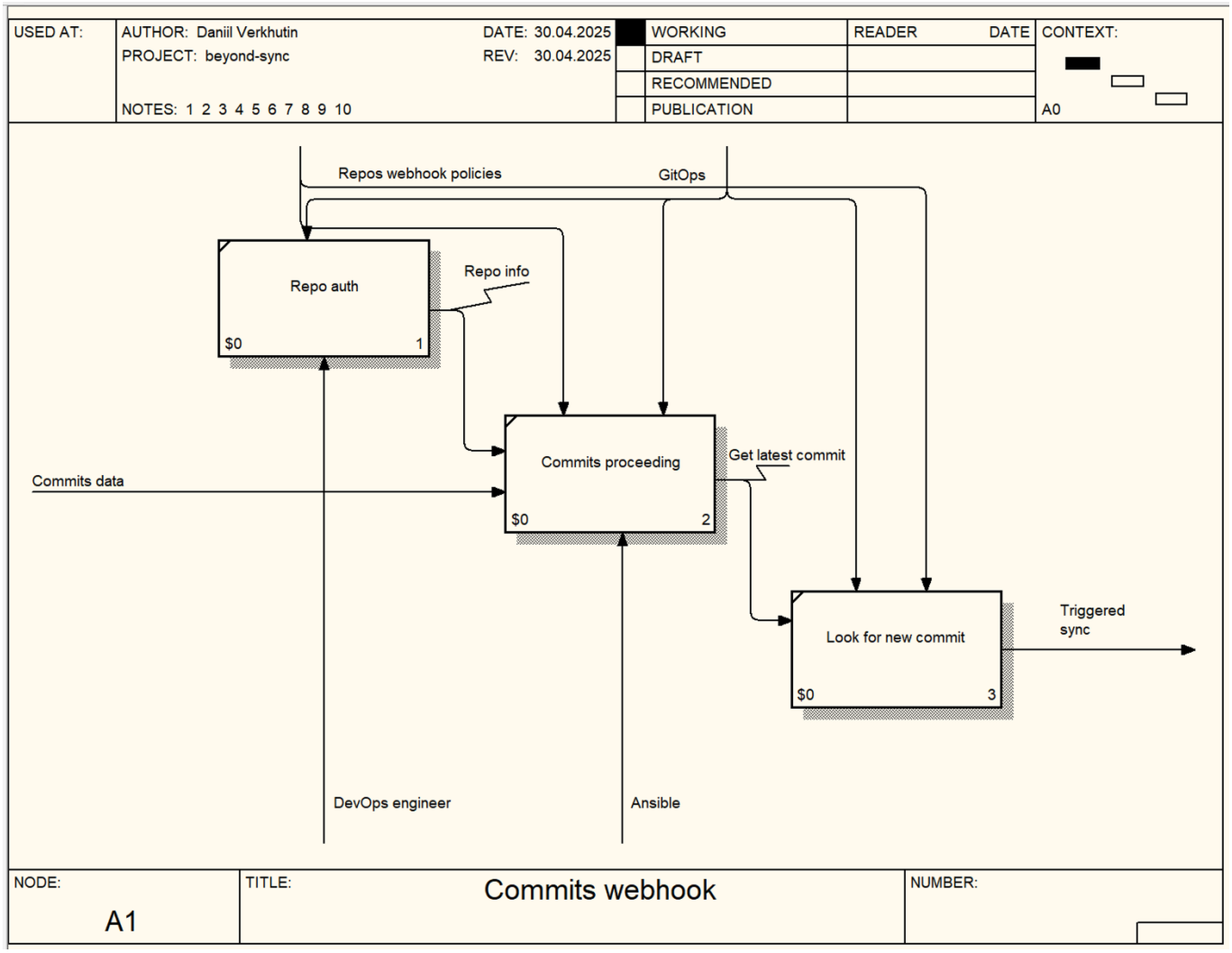

Рис. 2.5 Декомпозиція другого рівня процесу "Commits webhook"

На рисунку 2.5 можна побачити як процес запитів до змін виглядає у деталях з іншими процесами. Спочатку відбувається аутентифікація з репозиторієм, де ми в результаті отримуємо всю потрібну нам інформацію. Далі з неї отримуємо дані про зміни та аналізуємо їх, а конкретно останній та перевіряємо, чи оновився він. Коли це так, запускається тригер синхронізації.

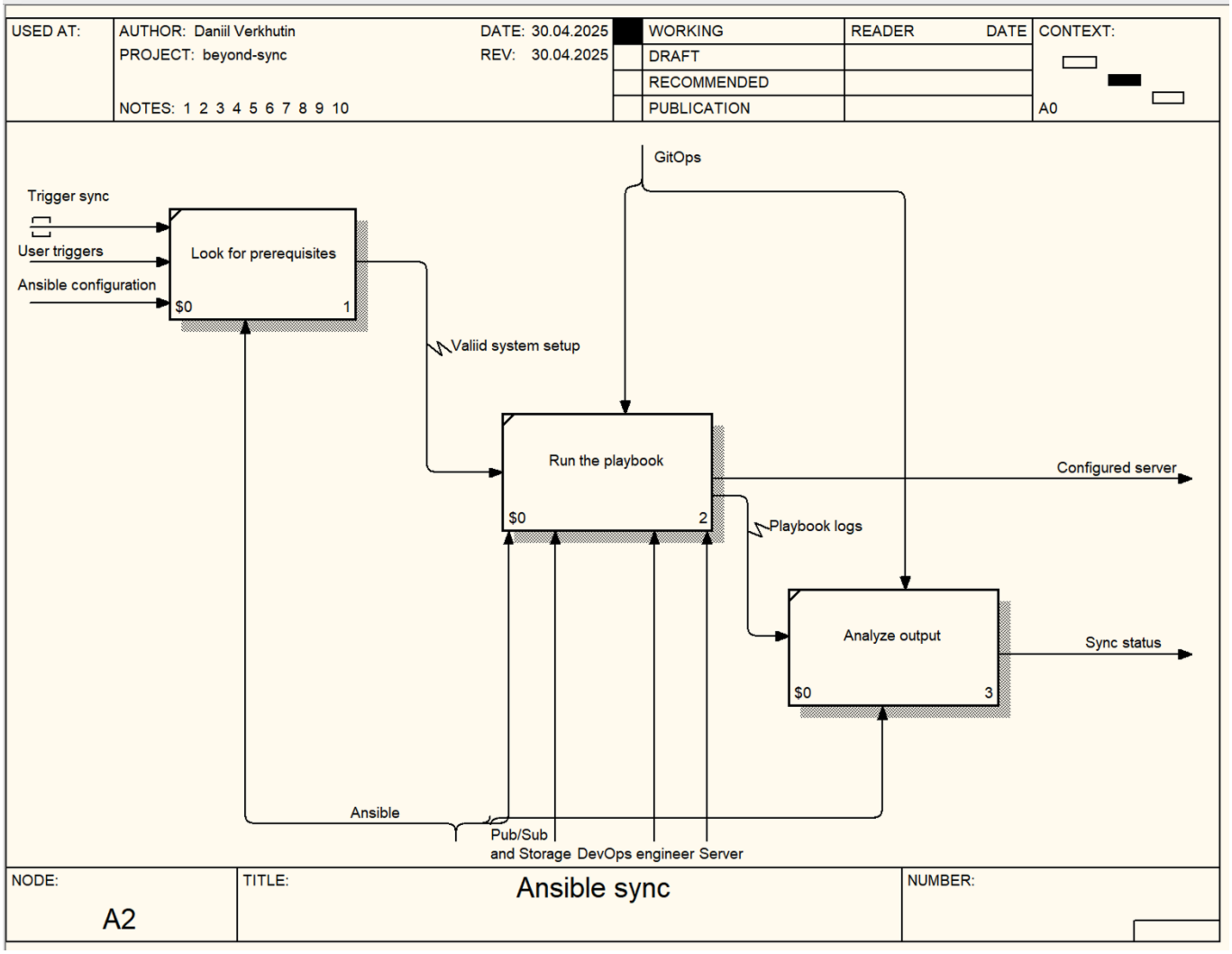

Рис. 2.6 Декомпозиція другого рівня процесу "Ansible sync"

На рисунку 2.6 показано декомпозицію функції Ansible синхронізації, яка складається з наступних процесів:

1. Перевірка, чи виконані всі вимоги для синхронізації (наприклад, чи вдалось склонувати репозиторій або встановлений Ansible).
2. Це дозволяє перейти на наступний етап та запустити конфігурацію.
3. Проаналізувати вивід результату конфігурації та передати статус синхронізації у наступний процес.

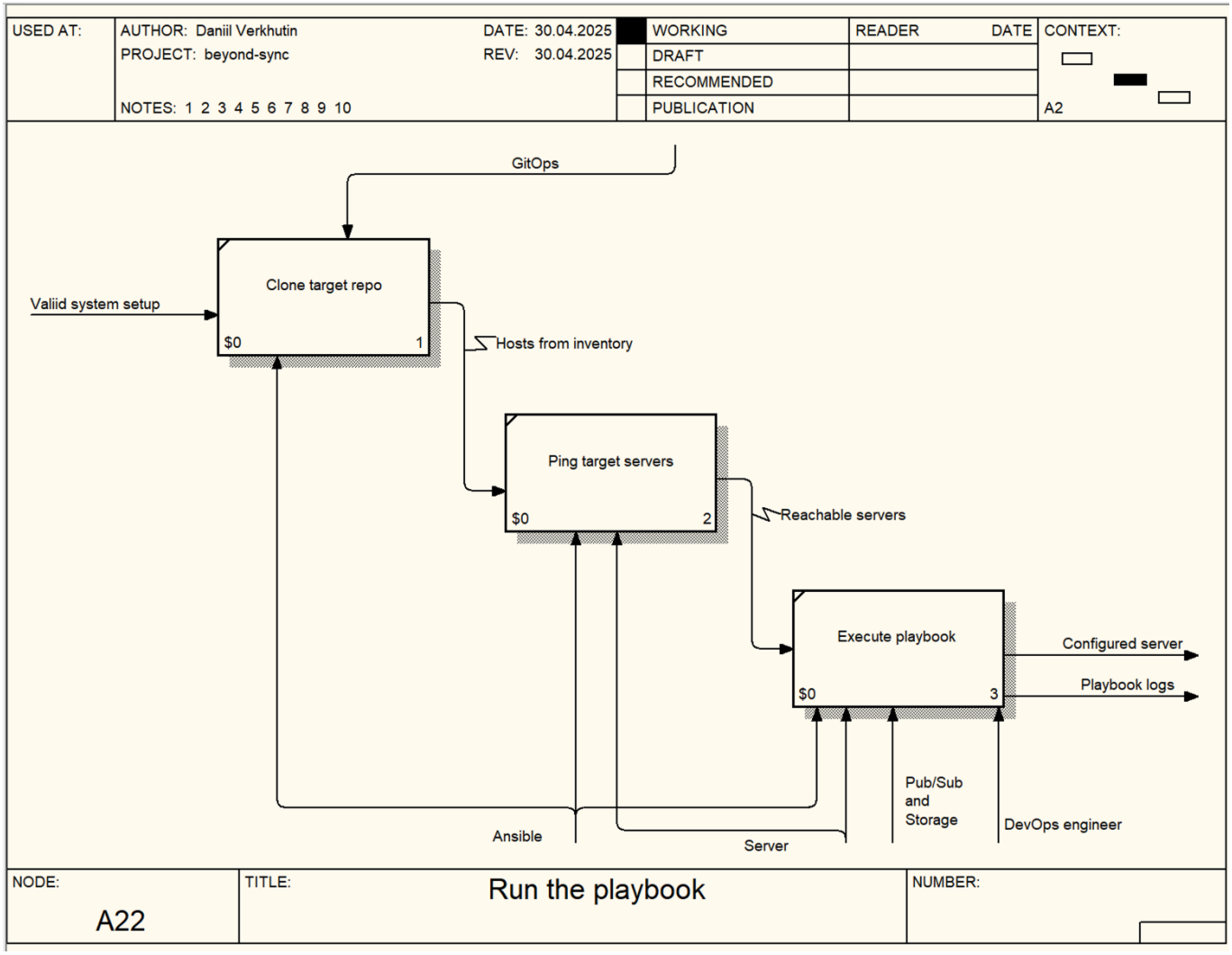

Рис. 2.7 Декомпозиція третього рівня процесу "Run the playbook"

На рисунку 2.7 наведено ще глибшу деталізацію системи, відображаючи декомпозицію процесу другого рівня декомпозиції запуску конфігурації, де відбувається стягнення репозиторію, пінг цільових серверів та відповідно виконання конфігурації.

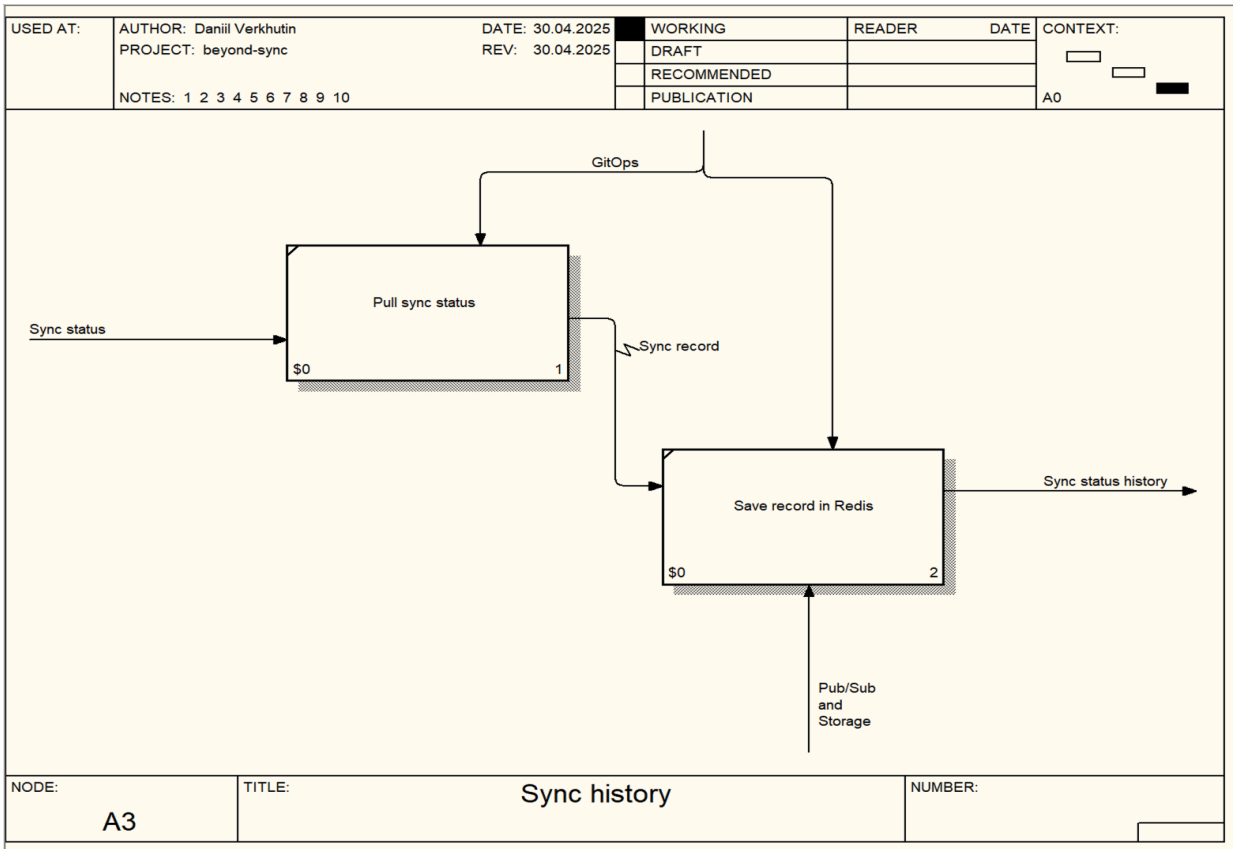

Рис. 2.8 Декомпозиція другого рівня процесу "Sync history"

Декомпозиція останнього процесу історії синхронізації декомпозиції першого рівня відображена рисунку 2.8. Вона є відносно простішою за попередні процесу, бо вона відповідає за збереження інформації про синхронізації у нереляційній базі даних. Ми просто отримуємо статус синхронізації та формуємо коректно відформатований запис, який записуємо у БД. Через API надалі можна буде переглянути історичну інформацію синхронізацій.

Останнім етапом важливо отримати загальний вигляд процесів. Опис функціонування будь-якої складної системи неможливий без чіткого розуміння її внутрішньої логіки та послідовності виконання процесів. З цією метою застосовується діаграма ієрархії процесів, або FEO-діаграма, яка дозволяє візуально представити структуру функцій системи та логіку їх взаємозв'язку.

FEO-діаграма відображає функціональні складові системи у вигляді ієрархічної структури, де верхній рівень містить загальні процеси, що поступово розкриваються

в підпроцеси нижчих рівнів. Це дозволяє поетапно деталізувати діяльність системи: від загальних цілей до конкретних задач і операцій. Подібна структуризація є важливою для аналізу функціональності, оптимізації ресурсів, визначення відповідальностей та виявлення вузьких місць у роботі системи.

FEO-діаграма виконує важливу роль у процесі системного аналізу, проєктування або реорганізації систем. Вона формує логічну основу для подальшого моделювання інформаційних потоків, розробки програмних або організаційних рішень, а також служить зручним засобом комунікації між розробниками, аналітиками та іншими зацікавленими сторонами.

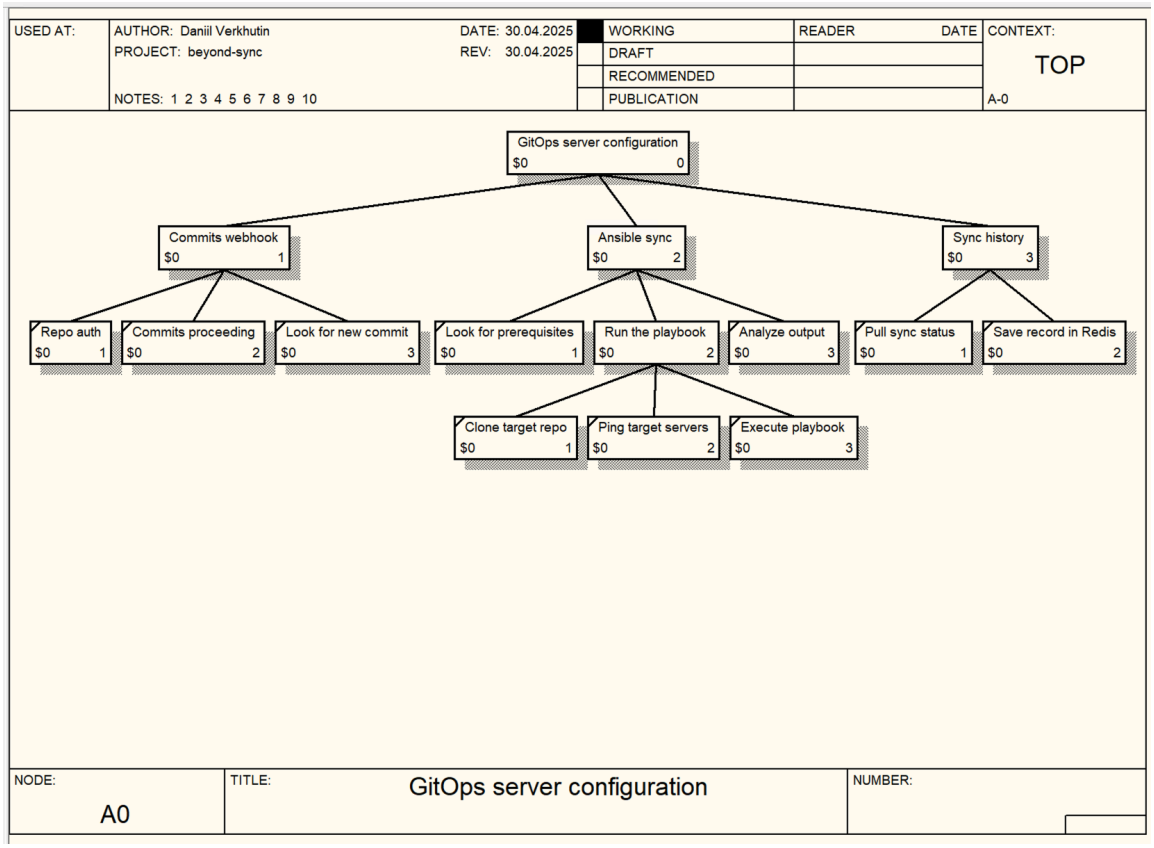

Рис. 2.9 Ієрархія процесів системи інформаційної системи для впровадження GitOps методології у процес конфігурації серверів

Висновок до розділу 2

У результаті проведеного системного аналізу було здійснено комплексне дослідження інформаційної системи для впровадження GitOps методології у процес

конфігурації серверів, що дало змогу краще зрозуміти його структуру, цілі та принципи функціонування. Побудова дерева цілей дозволила чітко сформулювати головну мету системи та визначити підцілі, що сприяють її досягненню.

Завдяки застосуванню методу аналізу ієрархії (MAI) було проведено порівняння альтернатив і визначено архітектуру, яка стане в основі технічної реалізації проєкту.

Конкретизація функціонування системи за допомогою методології IDEF0 забезпечила можливість наочно описати основні процеси, їх взаємодію, вхідні та вихідні дані, механізми та обмеження. Такий рівень деталізації дозволив краще уявити роботу системи в цілому та заклав основу для подальшої декомпозиції. Завершальним етапом стало побудування FEO-діаграми, яка представила ієрархію процесів системи та дозволила описати функціональність у вигляді логічної структури, що демонструє послідовність дій і взаємозв’язок між окремими елементами.

Таким чином, виконані дії в межах другого розділу дозволили не лише глибше проаналізувати об’єкт дослідження, а й сформувати чітке бачення архітектури майбутньої інформаційної системи. Це створює надійну методологічну та практичну базу для переходу до етапу проєктування.

РОЗДІЛ 3

Програмні засоби розв’язання задачі

У сучасній розробці інформаційних систем вибір відповідних програмних засобів є критичним етапом, який значною мірою визначає ефективність, масштабованість та надійність майбутнього рішення. Саме програмні інструменти забезпечують реалізацію закладеної логіки, підтримують взаємодію між компонентами системи та гарантують стабільну роботу програмного продукту в умовах реального навантаження. Крім того, правильне поєднання технологій дозволяє значно знизити витрати часу на розробку, підвищити рівень автоматизації процесів та забезпечити гнучкість при подальшій модифікації системи.

Розгляд програмних засобів, які застосовуються для розв’язання поставленої задачі, є необхідною умовою для забезпечення обґрунтованого вибору технологічного стеку. Це дозволяє не лише оцінити функціональні можливості кожного інструмента, але й врахувати їхню сумісність, популярність серед розробників, наявність документації, спільноти підтримки та перспективи подальшого розвитку. У цьому розділі буде здійснено аналіз обраних технологій та обґрунтовано їх доцільність у контексті реалізації проєкту.

3.1. Вибір та обґрунтування засобів розв’язання задачі

3.1.1. Мова програмування

У процесі розробки розподіленої системи на базі мікросервісної архітектури, яка включає окремі сервіси синхронізації, API-компоненти та агентське програмне забезпечення, що взаємодіє з інструментами на рівні операційної системи (зокрема Ansible та Git), критично важливим є обґрунтований вибір мови програмування. Такий вибір повинен враховувати низку факторів: швидкодію, простоту розгортання, підтримку паралелізму, екосистему, можливості інтеграції з іншими інструментами та бібліотеками, а також широту підтримки з боку спільноти розробників.

Apr 2025	Apr 2024	Change	Programming Language	Ratings	Change
1	1		Python	23.08%	+6.67%
2	3	▲	C++	10.33%	+0.56%
3	2	▼	C	9.94%	-0.27%
4	4		Java	9.63%	+0.69%
5	5		C#	4.39%	-2.37%
6	6		JavaScript	3.71%	+0.82%
7	7		Go	3.02%	+1.17%
8	8		Visual Basic	2.94%	+1.24%
9	11	▲	Delphi/Object Pascal	2.53%	+1.06%
10	9	▼	SQL	2.19%	+0.57%

Рис. 3.1 TIOBE індекс популярності мов програмування [21]

Одним із найпоширеніших варіантів для таких задач є Python. Завдяки великій кількості бібліотек, простоті синтаксису та активній спільноті, Python активно використовується в автоматизації, DevOps і побудові REST API. Його інтеграція з такими інструментами, як Ansible (який сам по собі базується на Python), виглядає максимально органічною. Проте, Python не завжди є оптимальним вибором для задач, що потребують високої продуктивності або багатопотокового виконання, з огляду на наявність GIL (Global Interpreter Lock) та інтерпретовану природу [22].

Іншою сильною опцією є Node.js, що базується на JavaScript. Його асинхронна модель та ефективність у обробці запитів роблять його популярним вибором для побудови веб-сервісів та RESTful API. Node.js має велику кількість доступних npm-пакетів, гнучкий підхід до розробки, і добре підходить для швидкого створення прототипів. Водночас, для низькорівневої взаємодії з системними утилітами та стабільної роботи в умовах високих навантажень, цей варіант може поступатися більш системно-орієнтованим мовам [23].

Також варто розглянути Rust, який, попри свою відносну молодість, зарекомендував себе як потужний інструмент для розробки високопродуктивних, безпечних до помилок систем. Rust пропонує відмінну підтримку багатопотоковості,

високий рівень контролю над пам’яттю, а також строгу систему типів, що запобігає багатьом класам помилок на етапі компіляції. Водночас крута крива навчання та менша кількість готових бібліотек у деяких сферах порівняно з Python або Go можуть бути стримуючими чинниками [24].

Нарешті, Go (Golang) також розглядається як один з провідних варіантів. Його простий синтаксис, швидка компіляція, відмінна підтримка паралелізму через goroutines, а також зручне управління залежностями роблять його ефективним вибором для побудови як агентів, що взаємодіють із системними утилітами, так і серверних API. Однак до його недоліків можна віднести обмежену кількість вбудованих можливостей у порівнянні з Python або Java, а також менш виразну об’єктно-орієнтовану модель [25].

3.1.2. Веб-фреймворк

Python: FastAPI - це сучасний асинхронний фреймворк для створення RESTful сервісів, побудований на базі Starlette та Pydantic. Він має вбудовану підтримку OpenAPI (Swagger), типізацію, автоматичну генерацію документації та високу продуктивність завдяки async/await. Перевагою є також дуже хороша інтеграція з Python-екосистемою та зручність для новачків, але він може поступатись у швидкодії Go-альтернативам [26].

Node.js: Express є класичним та найпоширенішим фреймворком для Node.js. Його гнучкість, величезна кількість плагінів та велика спільнота роблять його першочерговим вибором для багатьох REST API проєктів. Express простий у використанні, добре масштабується, проте не є найшвидшим варіантом і може вимагати додаткових рішень для асинхронної обробки та безпеки [27].

Rust: Actix Web - високопродуктивний та безпечний web-фреймворк для мови Rust. Його основна перевага - це швидкодія, яка часто перевищує інші фреймворки в незалежних тестах. Завдяки типобезпечності Rust, розробник має високу гарантію

стабільності й безпеки. Проте, вивчення Rust і його фреймворків потребує більше часу, а екосистема ще не така зріла, як у Python чи Node.js [28].

Go: Gin - один з найпоширеніших web-фреймворків для мови Go. Він надає простий інтерфейс для побудови REST API з мінімальними накладними витратами. Основні переваги: висока швидкість, низьке споживання ресурсів, легкість у деплої та зрілість інструментарію. Крім того, Go забезпечує зручну паралельну обробку завдяки вбудованим горутинам. Недоліком Gin може бути обмеженість у деяких аспектах розширення порівняно з більш повнофункціональними фреймворками [29].

3.1.3. База даних

Одним із поширених рішень для високошвидкісного доступу до даних є Redis - in-memory сховище ключ-значення, яке може використовуватись як кеш, тимчасова база даних або механізм взаємодії між сервісами за допомогою вбудованого механізму pub/sub. Redis демонструє високу продуктивність, надзвичайно простий у розгортанні й не потребує складного адміністрування. Однак Redis зберігає дані в оперативній пам’яті, тому його використання менш доречне для критичних або великих обсягів постійних даних, якщо не налаштовано збереження на диск чи реплікацію [30].

Іншим варіантом є PostgreSQL - потужна реляційна СУБД з відкритим кодом, яка підтримує складні запити, транзакції, розширення та гнучку схему даних. PostgreSQL часто застосовується у випадках, коли дані мають чітко визначену структуру та потребують гарантованої цілісності. Крім того, вона чудово працює у зв'язку з мікросервісами, які мають потребу у збереженні історичних записів, зв’язків між сутностями чи складної логіки на рівні бази даних [31]. Тут також може бути будь-який аналог реляційної SQL бази даних. Проте налаштування і масштабування є дещо складнішими, ніж у NoSQL-рішень.

У випадках, коли структура даних є динамічною або погано формалізованою, доцільним варіантом може бути MongoDB - документоорієнтована NoSQL-база. Вона

забезпечує зберігання JSON-подібних документів, масштабування за рахунок шардингу, а також вбудовані механізми реплікації. MongoDB підходить для швидкої розробки та зміни моделей даних без жорстких схем, однак у порівнянні з реляційними базами має обмеження щодо складних транзакцій та узгодженості в умовах високої конкуренції [32].

3.1.4. Pub/sub системи

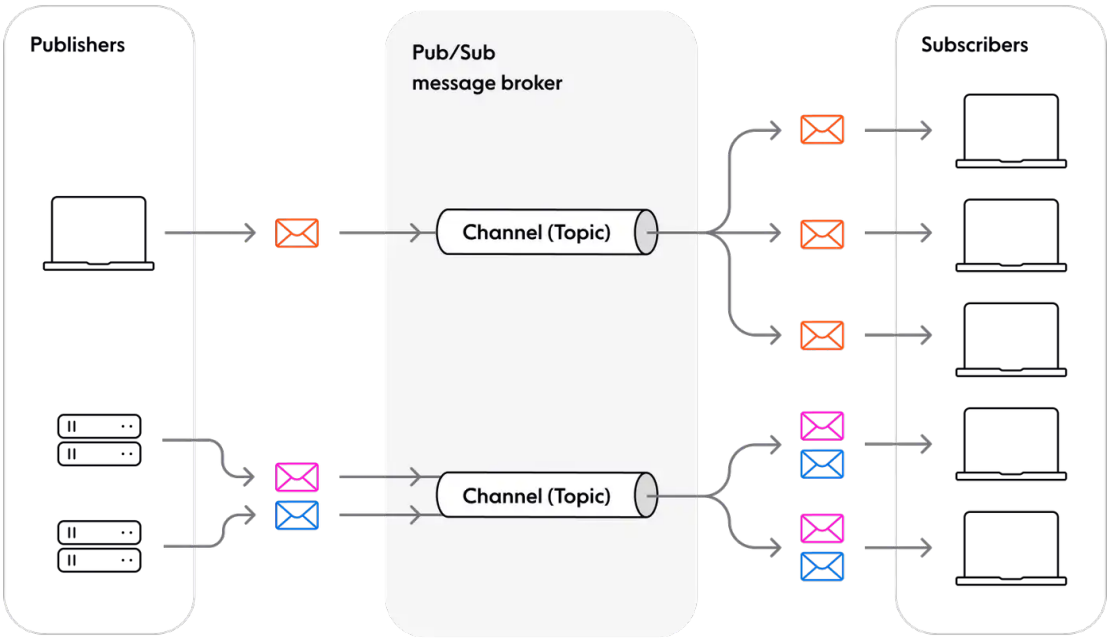

Рис. 3.2 Візуалізації роботи pub/sub систем [33]

Apache Kafka орієнтований на високопродуктивну обробку великих потоків даних у реальному часі. Його архітектура забезпечує зберігання повідомлень на тривалий час, можливість їх повторного зчитування, стійкість до збоїв і масштабованість, що робить цю технологію ідеальною для аналітичних або логуючих систем. Однак за все це доводиться платити складністю розгортання та суттєвими вимогами до інфраструктури. Kafka найчастіше обирають там, де важливо зберігати події, навіть якщо споживач тимчасово недоступний [34].

RabbitMQ, у свою чергу, пропонує більш гнучку систему обміну повідомленнями, що підтримує різні шаблони маршрутизації завдяки AMQP-протоколу. Його сильна сторона - це гнучкість у побудові черг і стратегій доставки, а також хороша підтримка інтеграцій. Це робить RabbitMQ доречним вибором для більшості типових бізнес-завдань, хоча при великому навантаженні система може демонструвати вищу затримку порівняно з Kafka, а її горизонтальне масштабування є складнішим [35].

Redis у режимі pub/sub - це найлегший і найшвидший інструмент серед перелічених. Його модель обміну побудована на передачі повідомлень через канали, що дозволяє досягти наднизької затримки. Такий підхід особливо зручний у випадках, коли критичною є швидкість, а не надійність збереження даних. Redis не зберігає повідомлення: якщо отримувач у момент публікації був недоступний, повідомлення безповоротно втрачається. Саме тому він підходить для простих систем реального часу, де не є критичним факт доставки кожного повідомлення [36].

3.1.5. Інструмент менеджменту конфігурацій

Ansible - це інструмент без використання агенту, що використовує SSH для доступу до цільових машин. Його основна перевага - простота: він не потребує попередньої установки агентів на сервер, а конфігурації пишуться у YAML-файлах (playbooks), що робить їх читабельними навіть для людей без глибокої технічної підготовки. Ansible добре підходить для сценаріїв, де важлива швидкість розгортання, простота підтримки та інтеграція з існуючими CI/CD-інструментами. Його обирають за низький поріг входу, модульність і активну спільноту [37].

Chef побудований на Ruby і використовує pull-модель: агенти на вузлах періодично звертаються до центрального сервера за інструкціями. Його архітектура більше підходить до великих інфраструктур, де потрібна складна логіка конфігурацій, яка може бути винесена в окремі скрипти. Chef добре масштабується, але має вищий

поріг входу через використання Ruby та складніший синтаксис. У деяких випадках він може виявитися надлишковим для простих автоматизаційних задач [38].

Puppet також працює за моделлю client-server та підтримує декларативний підхід до опису конфігурацій. Як і Chef, Puppet вимагає попередньої установки агентів і деякого часу на налаштування інфраструктури. Його DSL дозволяє створювати складні конфігурації з гнучкою логікою, що підходить для великих підприємств. Puppet має довгу історію, велику кількість модулів, але його використання іноді є надмірним для невеликих чи середніх проєктів через складність розгортання [39].

3.2. Технічні характеристики обраних програмних засобів розроблення

Обраний стек технологій - це поєднання Go (Golang), фреймворку Gin, Redis (з pub/sub), а також інструмента конфігураційного управління Ansible. Він формує ефективну, легку й масштабовану основу для побудови мікросервісної системи, орієнтованої на швидкодію та простоту супроводу. Go як мова програмування забезпечує високу продуктивність, низьке споживання ресурсів і хорошу підтримку паралелізму, що критично для сервісів реального часу та агентів, які встановлюються безпосередньо на сервери. Фреймворк Gin, завдяки своїй мінімалістичності та високій швидкості, дозволяє швидко створювати REST API з чіткою маршрутизацією й розширюваною структурою. Для зберігання даних та організації обміну повідомленнями між мікросервісами обрано Redis - це легкий у розгортанні й адмініструванні in-memory сховище, яке не вимагає складної конфігурації, а його вбудована підтримка pub/sub-механізму дозволяє налаштувати ефективну взаємодію між сервісами без додаткових брокерів повідомлень. Нарешті, Нарешті, Ansible, потужний CM-інструмент, який дозволяє автоматизувати налаштування та оновлення серверного середовища через SSH без встановлення агентів, що спрощує первинну інтеграцію й розгортання. У рамках архітектури проєкту ключовою особливістю є єдиний агент, що встановлюється безпосередньо у внутрішню інфраструктуру: він поєднує функціональність конфігуратора на базі Ansible та GitOps-інструмента,

відповідаючи за синхронізацію, виконання завдань та контроль змін. Такий підхід дозволяє зменшити кількість точок входу, забезпечити компактність рішення, підвищити рівень безпеки завдяки єдиному контролю доступу та спростити адміністрування навіть у розподіленому середовищі.

Технічні характеристики обраних інструментів:

Go (Golang):

Go є компільованою мовою, що генерує статичні бінарні файли без необхідності в інтерпретаторі або сторонніх бібліотеках. Серед особливостей - підтримка паралелізму через goroutines, що дозволяє ефективно обробляти велику кількість одночасних подій або запитів. Збірка проєкту виконується за допомогою вбудованого go build, а управління залежностями - через go mod. Go має власний garbage collector (concurrent, non-generational), який гарантує низькі затримки при зборі сміття. За допомогою net/http та context стандартної бібліотеки забезпечується контроль життєвого циклу запитів у мережевих додатках.

Gin Framework:

Gin побудований на основі net/http і використовує механізм "router tree", що базується на патерні radix tree - це забезпечує дуже швидку маршрутизацію HTTP-запитів. Підтримуються middleware з можливістю впливу на контекст обробки (*gin.Context), вбудована логіка обробки JSON, форм, multipart-даних. Для тестування надається об'єкт httptest, який дозволяє емулювати повноцінні HTTP-виклики. Gin легко інтегрується з Swagger (через swaggo/gin-swagger), OpenTelemetry, CORS, JWT і rate-limiting рішеннями. У продуктивних системах Gin може обробляти тисячі запитів на секунду з мінімальним overhead [40].

Redis (з Pub/Sub):

Redis працює в оперативній пам’яті, що забезпечує швидкість читання/запису до 1 мільйона операцій на секунду. Його внутрішня структура побудована на single-threaded event loop з неблокуючим I/O. Для публікації та підписки Redis використовує патерн "fire-and-forget": підписник миттєво отримує повідомлення, коли видається

подія. Ключові команди: PUBLISH, SUBSCRIBE, UNSUBSCRIBE, PSUBSCRIBE. Redis також дозволяє зберігати volatile-дані (наприклад, за TTL), що актуально для тимчасових сесій або конфігурацій. Кластеризація Redis забезпечує автоматичне поділу даних та копію через Sentinel або Redis Cluster [41].

Ansible:

Ansible використовує ssh (або WinRM на Windows) як транспортний механізм, працює без агента, а виконання команд здійснюється через YAML-файли (playbooks). Модулі Ansible написані переважно на Python і охоплюють тисячі типових сценаріїв (керування файлами, пакетами, службами, користувачами, git, docker, systemd тощо). Архітектура pull/push реалізується через рольову структуру (roles), можливість підключення змінних (vars), шаблонів (Jinja2) та inventory-файлів (динамічних і статичних). Ansible може працювати з git через ansible-galaxy, git модуль або зовнішні plugins, що забезпечує синхронізацію інфраструктурного стану з VCS.

Висновок до розділу 3

У цьому розділі було проведено комплексний аналіз програмних засобів, необхідних для реалізації інформаційної системи з мікросервісною архітектурою. Розглянуто альтернативні варіанти мов програмування, фреймворків, систем керування конфігураціями, баз даних та механізмів обміну повідомленнями, що дозволило зіставити переваги та недоліки кожного з них у контексті поставлених вимог.

На основі технічних характеристик та архітектурних потреб було сформовано оптимальний стек технологій, який включає мову програмування Go, вебфреймворк Gin, нереляційну базу даних Redis з підтримкою Pub/Sub, а також Ansible як інструмент для автоматизації розгортання та конфігурації. Вибрані засоби забезпечують високу продуктивність, масштабованість, простоту інтеграції та безпечне функціонування системи. Особливу увагу приділено єдиному агенту, який

поєднує в собі функції GitOps та конфігурування, що значно спрощує адміністрування інфраструктури.

Загалом, прийняті технологічні рішення дозволяють побудувати надійну, гнучку та легко підтримувану систему, що повністю відповідає вимогам до першого етапу її впровадження.

РОЗДІЛ 4

Практична реалізація

4.1. Опис створеного програмного засобу

Документація інформаційної система для впровадження GitOps методології у процес конфігурації серверів.

Відповідно до ISO/IEC 26514:2015.

Github репозирій: https://github.com/gitops-beyond/beyond-sync

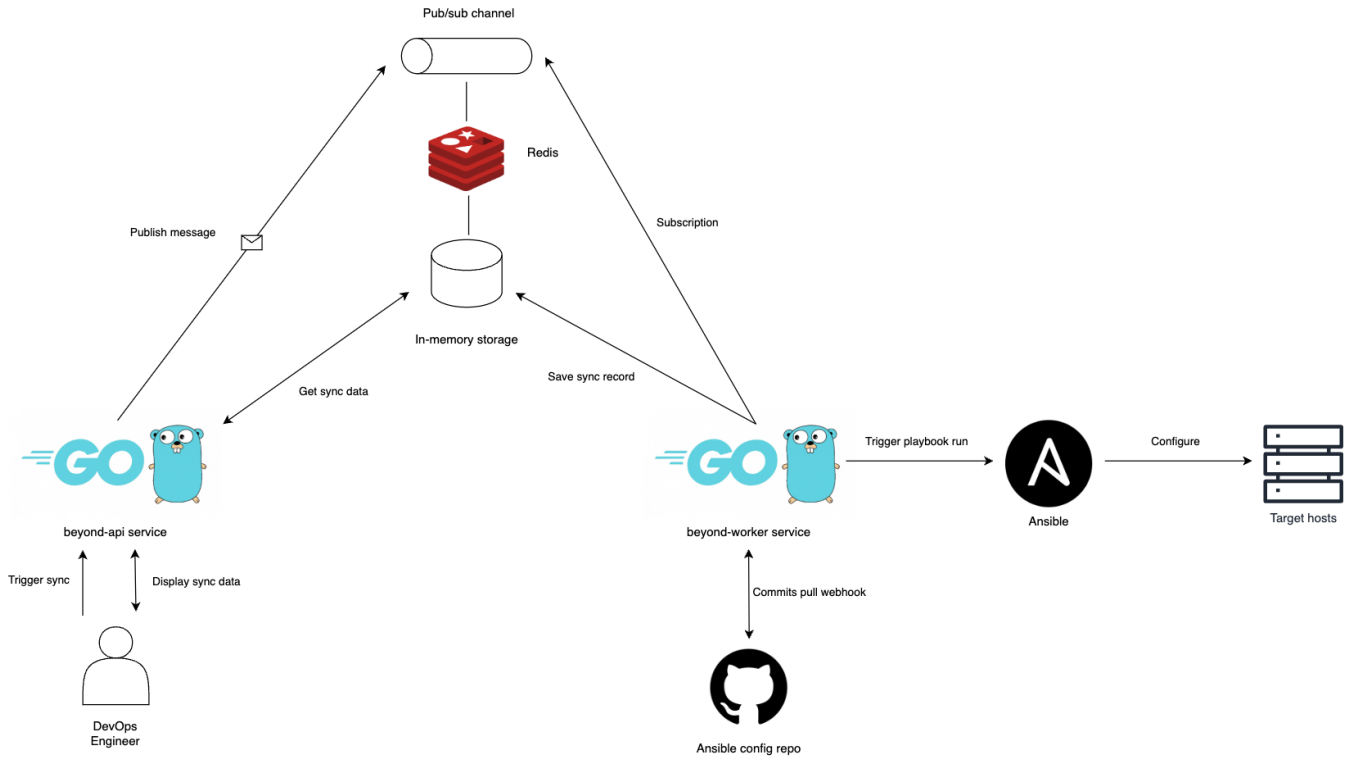

Рис. 4.1 Довільно зображена архітектура роботи системи

4.1.1. Ідентифікація продукту та стан

Назва: Beyond Sync

Версія: 1.0

Класифікація: Система синхронізації та керування конфігураціями корпоративного рівня

Стан розробки: Виробнича (MVP)

Технології: Go, Redis, Ansible, GitHub

4.1.2. Огляд системи

Призначення:

Beyond Sync - це автоматизована система управління синхронізацією та конфігурацією, розроблена для роботи в розподіленому середовищі. Вона поєднує переваги GitOps-підходу та можливості Ansible для автоматичного застосування змін конфігурації. Система надає API для запуску, моніторингу та аудиту синхронізацій.

Основні функції:

GitOps-інтеграція:

- Відстеження змін у GitHub-репозиторії
- Автоматичне застосування змін через агента
- Обробка подій за допомогою Redis Pub/Sub

Синхронізація та керування конфігурацією:

- Ручне та автоматичне ініціювання синхронізацій через API або webhook
- Реальне застосування Ansible-конфігурацій
- Збереження історії змін та логів у Redis
- Моніторинг статусу синхронізацій

4.1.3. Архітектура

Система має мікросервісну архітектуру та складається з:

1. Агент/Worker-сервіс (beyond-sync-worker)

Тип: фоновий сервіс.

Призначення: застосування конфігурацій, обробка подій з GitHub, виконання Ansible.

Бібліотеки:

- go-git/go-git/v5
- Redis-клієнт

Повторний запуск: кожні 5 сек. при збої

2. API-сервіс (beyond-sync-api)

Фреймворк: Gin.

Документація: Swagger/OpenAPI.

Порт: 8080

Бібліотеки:

- gin-gonic/gin
- go-redis/redis/v9
- swaggo/gin-swagger

3. Redis

Використовується для:

- Зберігання SyncRecord
- Механізм обміну повідомленнями (Pub/Sub)
- Статус синхронізацій

4. Ansible

- Виконує сценарії конфігурації (playbooks)
- Підключений до worker-сервісу

4.1.4. Моделі даних

Агент-сервіс:

```
type SyncData struct {
    Sha string `json:"sha"`
    Status string `json:"status"`
    Message string `json:"message"`
}
```

SyncData використовується для позначення інформації запису про синхронізацію, містячи у собі всю базову необхідної для її опису: хеш змін з Github, на основі отримання якого робиться синхронізація, статус - чи успішна була виконана Ansible-конфігураціґ, повідомлення - логи Ansible з виконання конфігурації.

```
type Webhook struct {
 RepoName string
```

```
 Username string
 Token string
}
```

Webhook використовується для опису об’єкта, що робить постійні запити у репозиторію, перевіряючи його оновлення. У нього входять поля: ім’я цільового репозиторію, ім’я користувача у Github в даному випадку та токен PAT для авторизації у репозиторій.

API-сервіс:

```
type SyncRecord struct {
   Timestamp string          `json:"timestamp"`
   Data      redis.SyncData `json:"data"`
}
```

SyncRecord - модель, що позначає один запис об’єкту синхронізації в Redis та містить у собі часову позначку та дані синхронізації типу об’єкта попередньої моделі SyncData.

4.1.5. API інтерфейс

Отримання всіх записів:

- Метод: GET /sync
- Призначення: Повертає масив SyncRecord
- Формат: JSON

Отримання запису за часом:

- Метод: GET /sync/{timestamp}
- Призначення: Повертає конкретний SyncRecord
- Формат: JSON

Запуск синхронізації:

- Метод: POST /sync/trigger
- Призначення: Запускає синхронізацію вручну
- Формат: JSON

- Результат: 201 Created

Також з автоматично згенерованою Swagger документацією API можна ознайомитись за посиланням `<beyond-sync host>:8080/swagger/index.html#/`, де `beyond-sync host` - це IP-адреса/доменне ім’я серверу, де розгорнута система.

4.1.6. Інсталяція та сервісна структура

Вимоги

Сервер агенту:

- Архітектура: ARM
- ОС: Linux Debian-based (Debian, Ubuntu і т.п.)
- Вільні порти: 8080 та 6379
- SSH ключ до серверів з inventory
- Може дістатись по SSH цільових серверів

Ansible проєкт:

- Репозиторій: Github
- Згенерований Github PAT
- Локація: ./ansible/
- Playbook: ./ansible/playbook.yml
- Inventory: ./ansible/playbook.yml

Встновлення системи відбувається ща допомогою `install.sh` скрипта, що скачує на сервер Python, Ansible та Redis, створює сервісного користувача та розгортає сервіси systemd.

4.1.7. Інсталяція та сервісна структура

- Обмежений shell-доступ (/sbin/nologin)
- Ізоляція сервісів у /opt/beyond-sync
- Файл середовища окремо
- Мінімальні привілеї (least privilege)

- PAT авторизація у Github

4.1.8. Моніторинг і відновлення

- Перевірка статусу через systemctl status beyond-sync-*
- Перевірка підключення до Redis
- Перевірка підключення до цільових серверів
- Журнали: systemd journal, окремі файли логів
- Автоматичне відновлення при збої (5 сек.)

4.1.9 Плани на майбутнє:

- UI та CLI інтерфейс
- Авторизація
- Інтеграції з CM провайдерами та платформами репозиторіїв
- Бекапи та відновлення
- Підтримка багатьох агентів

4.2. Інструкція користувача

Відповідно до ISO/IEC 26514:2015.

Github репозирій: https://github.com/gitops-beyond/beyond-sync

4.2.1. Вступ

Інформаційна система призначена для автоматизованого управління конфігурацією серверної інфраструктури з використанням підходу GitOps та інструментів Ansible. Її основна функція полягає в забезпеченні синхронізації конфігурацій за змінами у віддаленому git репозиторії, що дозволяє автоматично розгортати та підтримувати необхідний стан інфраструктури. Для цього використовується мікросервісний агент, розроблений на Go з використанням Gin,

який взаємодіє з Redis, Ansible та GitHub, а також надає API для ручного запуску синхронізацій і моніторингу їхнього стану.

Документ містить опис призначення системи, умов її застосування, логічної структури, а також формату вхідних і вихідних даних. Наводиться архітектура компонентів, описано налаштування, спосіб запуску процесів синхронізації, логіку реагування на події у репозиторії, зберігання стану виконання та результати Ansible-конфігурацій у Redis.

4.2.2. Загальні відомості про програму

Позначення і найменування програми:

“Beyond” - інформаційна система для впровадження GitOps методології у процес конфігурації серверів.

Мови програмування, на яких написана програма:

Go

Призначення програми:

Програма призначена для впровадження GitOps-підходу у процес конфігурації серверів, забезпечуючи автоматичне застосування змін з віддаленого Git-репозиторію. Вона може використовуватись у сферах DevOps, адміністрування інфраструктури, тестування та впровадження декларативної конфігурації у середовищах з Linux-серверами.

Можливості програми:

Система надає можливість автоматичного й ручного запуску процесів синхронізації, моніторингу їхнього стану через API, а також інтегрується з Ansible для застосування конфігурацій.

Обмін даними між компонентами забезпечується за допомогою Redis, де також зберігаються логи виконання та поточний стан.

4.2.3. Класи вирішуваних завдань

Опис завдань:

Система дозволяє вирішувати завдання автоматизованого розгортання конфігурацій серверів, моніторингу їхнього стану, централізованого керування змінами в конфігурації. Вона підходить для побудови стабільного й відтворюваного середовища з використанням декларативної конфігурації.

Методи вирішення завдань:

Для розв'язання зазначених завдань застосовується методика GitOps, де всі конфігурації зберігаються у версійному контролі Git. Агент системи відстежує зміни в репозиторії, а при їх виявленні ініціює застосування через Ansible. Комунікація компонентів реалізована через Redis, що виконує функції сховища станів і каналу обміну повідомленнями

Функції, що виконуються програмою:

1. Управління синхронізаціями:

- `TriggerSync()` (`api/handlers/sync.go`) - запускає нову синхронізацію, публікуючи повідомлення в Redis.
- `GetAllSyncs()` (`api/handlers/sync.go`) - отримує історію всіх записів синхронізацій.

2. Інтеграція з Redis (`internal/redis/redis.go`):

- `PublishMessage()` - публікує повідомлення для запуску синхронізації в Redis.
- `Subscribe()` - підписується на Redis-канали для отримання повідомлень про синхронізацію.
- `GetSyncRecords()` - отримує історію записів синхронізацій з Redis.
- `AddSyncRecord()` - зберігає результати виконання синхронізацій у Redis із позначкою часу.

3. Управління встановленням (install.sh):

- Створює системного користувача та необхідні каталоги.

- Встановлює системні залежності (Redis, Ansible, Git).
- Налаштовує служби systemd (beyond-sync-api і beyond-sync-worker).

Відповідно до README.md, застосунок є інструментом GitOps, який:

- Моніторить Git-репозиторії на предмет змін у конфігураціях Ansible.
- Автоматично застосовує оновлення конфігурацій Ansible.
- Забезпечує моніторинг статусу в реальному часі.
- Надає REST API для інтеграції.
- Використовує Redis для моніторингу станів.

Для повної документації API застосунок надає Swagger-документацію, яка доступна за адресою:

```
<beyond-sync host>:8080/swagger/index.html#/
```

4.2.4. Опис основних характеристик і особливостей програми

Часові характеристики:

Впровадження інформаційної системи для автоматизованого управління конфігурацією серверів за допомогою GitOps значно скорочує час виконання завдань, порівняно з традиційними методами. Завдяки автоматичному застосуванню змін через Git та Ansible, час на розгортання нових конфігурацій зменшується на 40-50% . Використання єдиного інтерфейсу Git для управління конфігураціями спрощує процес навчання нових співробітників, оскільки всі зміни документуються в Git, що забезпечує прозорість і зручність у відстеженні змін . Інтеграція GitOps дозволяє об'єднати процеси контролю версій, тестування та розгортання в єдину автоматизовану лінію, що усуває необхідність у додаткових інструментах CD та зменшує складність інфраструктури . Завдяки централізованому зберіганню конфігурацій у Git, розробники мають чітке уявлення про поточний стан серверів, що полегшує процеси розробки та тестування . У разі виникнення проблем, GitOps дозволяє швидко відкотити зміни до попереднього стабільного стану, що значно скорочує час відновлення системи.

Режим роботи:

Цілодобовий.

Засоби контролю правильності виконання і самовідновлення програми (з прикладами):

1. Перевірки стану підключення до Ansible та Redis

```
// internal/redis/redis.go
// Connection health check before operations
if err := rdb.Ping(ctx).Err(); err != nil {
    return fmt.Errorf("failed connecting to Redis: %v", err)
}

// internal/ansible/ansible.go
 // Verify connectivity to all hosts
 err = pingAllHosts()
 if err != nil {
      log.Printf("ERROR: %s", err.Error())
      redis.AddSyncRecord(sha, "Failed", err.Error())
      return
 }
```

Усі операції Redis перевіряють з’єднання перед виконанням команд.

2. Блокування під час паралельних процесів

```
  // First goroutine
  // Automatic sync
  go func() {
      // End goroutine when function ends
      defer wg.Done()
     …

          // Compare sha value from current loop with previous one
          if sha != newSha {
              // Lock the process to not create a conflict between
syncs
              lock.Lock()
```

```
                …
                 ansible.RunAnsibleSync(sha)
                 // Unlock the process to use concurrency
                 lock.Unlock()
             }
         }
    }()
```

Для запобігання конфліктам у конфігурації система використовує механізми блокування під час паралельних процесів, що забезпечує коректність виконання та автоматичне відновлення у разі помилок.

3. Обробка помилок у API ендпоінтах:

```
// api/handlers/sync.go
func GetAllSyncs(c *gin.Context) {
    redisRecords, err := redis.GetSyncRecords("*")
    if err != nil && err.Error() == "key not found"{
        c.JSON(404, gin.H{"error": err.Error()})
        return
    } else if err != nil {
        c.JSON(500, gin.H{"error": err.Error()})
        return
    }
    // ...
}
```

API належним чином обробляє та повідомляє про різні типи помилок із відповідними кодами стану HTTP.

4. Виявлення помилок під час встановлення

```
# install.sh
error() {
    echo -e "${RED}[ERROR]${NC} $1"
    exit 1
}
```

```
# Distribution compatibility check
if [ -f /etc/debian_version ]; then
    # Debian/Ubuntu handling
elif [ -f /etc/redhat-release ]; then
    # RHEL/CentOS handling
else
    error "Unsupported distribution"
fi
```

Сценарій встановлення включає виявлення помилок і звітування для сумісності системи.

5. Валідація даних

```
// internal/redis/redis.go
func AddSyncRecord(sha string, status string, message string) {
    value, err := json.Marshal(SyncData{sha, status, message})
    if err != nil {
        fmt.Printf("Error encoding json value: %v\n", err)
        return
    }
}
```

Перевірка вхідних даних через маршалінг/демаршалінг JSON.

Обмеження області застосування програми:

Включають її сумісність лише з віддаленими Git-репозиторіями (зокрема, GitHub), підтримку лише Linux-операційних систем, а також відсутність авторизації в системі та API, UI та CLI. Крім того, програма підтримує базову інтеграцію з Ansible для конфігураційних змін та не містить функцій для автоматичного масштабування чи резервного копіювання, що обмежує її використання у великих та складних інфраструктурах.

4.2.5. Відомості про функціональні обмеження на застосування

Умови, необхідні для виконання програми:

Системні вимоги:

- Bash
- Операційна система: Linux на основі ARM (Debian/Ubuntu)

Потрібні сервіси:

- Redis сервер
- Ansible
- Git
- curl

Мережеві вимоги:

- Порти:
  - 6379 (Redis)
  - 8080 (API)
- Мережевий доступ до цільових серверів Ansible

Вимоги до доступу (з README.md):

- SSH ключі для доступу до Ansible проекту
- GitHub Personal Access Token (PAT)

Вимоги до структури проєкту: Ansible проект має бути розташований у такій структурі:

- Проект у директорії 'ansible/'
- Файл інвентарю в 'ansible/inventory'
- Файл конфігурації в 'ansible/playbook.yml'

Змінні середовища:

- `REPONAME` - назва репозиторію з Ansible проєктом
- `USERNAME` - ім’я користувача у Github
- `TOKEN` - Github PAT
- `REDIS_HOST` - хост Redis

Користувачі та дозволи:

- Обліковий запис сервісного користувача (створюється під час установки)
- Правильні дозволи для директорій

Кращі практики:

- Регулярні перевірки стану системи
- Моніторинг операцій синхронізації
- Правильне оброблення помилок
- Регулярні оновлення та виправлення безпеки

Обмеження:

- Підтримка тільки архітектури ARM64
- Обмежено до конкретних дистрибутивів Linux
- Потрібна специфічна структура проєкту
- Залежність від Redis

Відомості про технічні та програмні засоби, що забезпечують виконання програми:

Програма працює на ARM64-сумісних Linux-системах із встановленими Redis, Ansible, Git та curl; використовує systemd-сервіси для запуску, а також SSH та мережевий доступ для взаємодії з цільовими серверами.

Вимоги до складу і параметрів периферійних пристроїв:

Спеціальні вимоги до периферійних пристроїв відсутні - достатньо стандартного мережевого інтерфейсу для доступу до серверів.

Вимоги до програмного забезпечення:

Системні вимоги включають операційну систему Linux з підтримкою ARM64-архітектури, зокрема дистрибутиви Debian/Ubuntu або RHEL/CentOS. До спеціального програмного забезпечення належать: Redis Server для зберігання стану, Ansible для керування конфігураціями, Git для роботи з репозиторіями, а також curl для мережевих запитів. Спеціальні драйвери або додаткове апаратне забезпечення, як-от принтери чи сканери, не вимагаються.

Вимоги та умови організаційного, технічного і технологічного характеру:

Для повноцінного функціонування системи необхідна стабільна локальна мережа з доступом до Інтернету, щоб забезпечити доступ до Git-репозиторіїв та

цільових серверів для Ansible. Також потрібно налаштувати міжмережевий екран для безпечної роботи сервісів, відкрити порти 6379 (Redis) і 8080 (REST API), а також забезпечити безперебійне живлення та моніторинг роботи серверного обладнання.

4.3. Аналіз контрольного прикладу

Слідуючи інструкції користувач, встановлюємо програму. Для цього додаємо SSH-ключ цільового серверу, який зазначений у Ansible inventory, клонуємо репозиторій на хост, куди хочемо встановити агент.

Також підготуємо на Github Ansible проєкт, який ми хочемо відстежувати інформаційною системою для впровадження GitOps методології у процес конфігурації серверів. Це буде простий приклад конфігурації, у якому є одна задача - змінити назву хосту цільового сервера. Для початку у ньому навмисно була допущена помилка у inventory для того, щоб продемонструвати обробку помилок та правильне відстеження оновлення коммітів.

Коли ці вимоги виконанні, запускаємо скрипт для встановлення системи:

```
daniil@beyond:~/beyond-sync$ sudo ./install.sh
[BEYOND-SYNC] Starting Beyond Sync installation...
[BEYOND-SYNC] Creating service user and group...
[BEYOND-SYNC] Creating installation directory...
[BEYOND-SYNC] Installing system dependencies...
Hit:1 http://ports.ubuntu.com/ubuntu-ports noble InRelease
```

Рис. 4.2 Запуск скрипту встановлення beyond-sync

Вводимо дані про відстежуванний репозиторій з Ansible конфігурацією, що зображений вище - “gitops-test”:

```
[BEYOND-SYNC] Setting up environment file...
Enter GitHub username: DanVerh
Enter GitHub token (output is hidden):
Enter repository name: gitops-test
```

Рис. 4.3 Встановлення змінних середовища для авторизації з репозиторієм

Проєкт було успішно завантажено:

```
[BEYOND-SYNC] Installing systemd services...
Created symlink /etc/systemd/system/multi-user.target.wants/beyond-sync-api.service → /etc/systemd/system/beyond-sync-api.service.
Created symlink /etc/systemd/system/multi-user.target.wants/beyond-sync-worker.service → /etc/systemd/system/beyond-sync-worker.service.
[BEYOND-SYNC] Installation completed successfully!
```

Рис. 4.4 Логи інсталяційного скрипту, що свідчать про успішне встановлення

Перевіряємо статус beyond-sync-worker:

```
daniil@beyond:~/beyond-sync$ sudo systemctl status beyond-sync-worker
● beyond-sync-worker.service - Beyond Sync Worker Service
     Loaded: loaded (/etc/systemd/system/beyond-sync-worker.service; enabled; preset: enabled)
     Active: active (running) since Sun 2025-05-04 20:38:31 UTC; 8s ago
   Main PID: 2092 (beyond-sync-wor)
      Tasks: 7 (limit: 1610)
     Memory: 18.8M (peak: 64.3M)
        CPU: 513ms
     CGroup: /system.slice/beyond-sync-worker.service
             └─2092 /opt/beyond-sync/beyond-sync-worker

May 04 20:38:31 beyond systemd[1]: Started beyond-sync-worker.service - Beyond Sync Worker Service.
May 04 20:38:32 beyond beyond-sync-worker[2092]: 2025/05/04 20:38:32 Sync is triggered with new commit hash value of ac3af402f249a94cc80a349fa4718c46e651b2ce
May 04 20:38:32 beyond beyond-sync-worker[2092]: 2025/05/04 20:38:32 Cloning repo https://github.com/DanVerh/gitops-test
May 04 20:38:36 beyond beyond-sync-worker[2092]: 2025/05/04 20:38:36 ERROR: 192.168.0.110 | UNREACHABLE! => {
May 04 20:38:36 beyond beyond-sync-worker[2092]:     "changed": false,
May 04 20:38:36 beyond beyond-sync-worker[2092]:     "msg": "Failed to connect to the host via ssh: ssh: connect to host 192.168.0.110 port 22: No route to host",
May 04 20:38:36 beyond beyond-sync-worker[2092]:     "unreachable": true
May 04 20:38:36 beyond beyond-sync-worker[2092]: }
```

Рис. 4.5 Статус systemd сервісу beyond-sync-worker

Бачимо наступне:

1. Так як інструмент запущений перший раз, даних про попередні зміни немає, відповідно конфігурація була запущена з останньої зміни.
2. Як зазначено попередньо, конфігурація від початку мала неправильний inventory файл, тому можна побачити, що синхронізація пройшла неуспішно.
3. Як висновок, worker-сервіс відпрацював коректно.

Одразу перевіримо статус API-сервісу:

```
daniil@beyond:~/beyond-sync$ sudo systemctl status beyond-sync-api
● beyond-sync-api.service - Beyond Sync API Service
     Loaded: loaded (/etc/systemd/system/beyond-sync-api.service; enabled; preset: enabled)
     Active: active (running) since Sun 2025-05-04 20:39:27 UTC; 4s ago
   Main PID: 2118 (beyond-sync-api)
      Tasks: 6 (limit: 1610)
     Memory: 11.2M (peak: 11.6M)
        CPU: 47ms
     CGroup: /system.slice/beyond-sync-api.service
             └─2118 /opt/beyond-sync/beyond-sync-api

May 04 20:39:27 beyond beyond-sync-api[2118]:  - using code:        gin.SetMode(gin.ReleaseMode)
May 04 20:39:27 beyond beyond-sync-api[2118]: [GIN-debug] GET    /swagger/*any             --> github.com/swaggo/gin-swagger.CustomWrapHandler.func1 (3 handlers)
May 04 20:39:27 beyond beyond-sync-api[2118]: [GIN-debug] GET    /docs/*filepath           --> github.com/gin-gonic/gin.(*RouterGroup).createStaticHandler.func1 (3 handlers)
May 04 20:39:27 beyond beyond-sync-api[2118]: [GIN-debug] HEAD   /docs/*filepath           --> github.com/gin-gonic/gin.(*RouterGroup).createStaticHandler.func1 (3 handlers)
May 04 20:39:27 beyond beyond-sync-api[2118]: [GIN-debug] GET    /sync                     --> github.com/gitops-beyond/beyond-sync/api/handlers.GetAllSyncs (3 handlers)
May 04 20:39:27 beyond beyond-sync-api[2118]: [GIN-debug] GET    /sync/:timestamp          --> github.com/gitops-beyond/beyond-sync/api/handlers.GetSyncByDate (3 handlers)
May 04 20:39:27 beyond beyond-sync-api[2118]: [GIN-debug] POST   /sync/trigger            --> github.com/gitops-beyond/beyond-sync/api/handlers.TriggerSync (3 handlers)
May 04 20:39:27 beyond beyond-sync-api[2118]: [GIN-debug] [WARNING] You trusted all proxies, this is NOT safe. We recommend you to set a value.
May 04 20:39:27 beyond beyond-sync-api[2118]: Please check https://pkg.go.dev/github.com/gin-gonic/gin#readme-don-t-trust-all-proxies for details.
May 04 20:39:27 beyond beyond-sync-api[2118]: [GIN-debug] Listening and serving HTTP on :8080
```

Рис. 4.6 Статус systemd сервісу beyond-sync-api

Він успішно запустився, але перевіримо його пізніше разом з успішною конфігурацією.

Перед тим, як виправити inventory файл, перевіримо, яка назва була попередньо на цільовому сервері:

```
daniil@target:~$ hostname
target
```

Рис. 4.7 Попередня назва цільового хосту

Тепер виправляємо помилки у репозиторії конфігурацій новою зміною та даємо назву серверу, наприклад, “newhostname”:

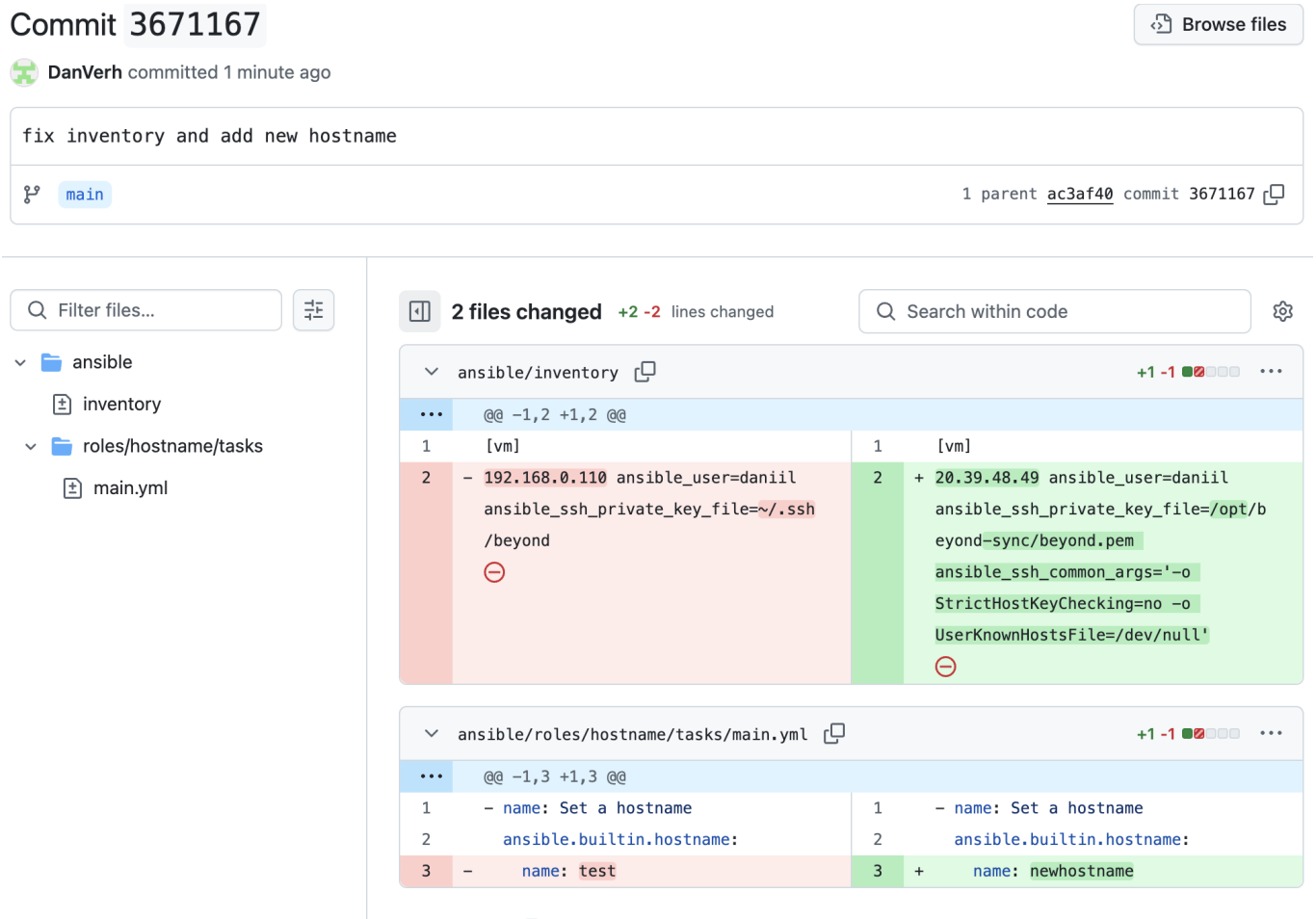

Рис. 4.8 Інформація про зміни, що ініціював автоматичну GitOps-синхронізацію

Перевіряємо логи worker-сервісу, який мав би синхронізувати зміни на сервері відповідно до змін у репозиторії:

```
daniil@beyond:/opt/beyond-sync$ sudo systemctl status beyond-sync-worker
● beyond-sync-worker.service - Beyond Sync Worker Service
     Loaded: loaded (/etc/systemd/system/beyond-sync-worker.service; enabled; preset: enabled)
     Active: active (running) since Sun 2025-05-04 20:38:31 UTC; 29min ago
   Main PID: 2092 (beyond-sync-wor)
      Tasks: 8 (limit: 1610)
     Memory: 23.5M (peak: 89.0M)
        CPU: 7.219s
     CGroup: /system.slice/beyond-sync-worker.service
             ├─2092 /opt/beyond-sync/beyond-sync-worker
             └─2547 "ssh: /opt/beyond-sync/.ansible/cp/cc81fb1bd1 [mux]"

May 04 21:07:18 beyond beyond-sync-worker[2092]: 2025/05/04 21:07:18 Sync is triggered with new commit hash value of 367116781d9e9b67086eda22c9bc57445bd8fd7b
May 04 21:07:18 beyond beyond-sync-worker[2092]: 2025/05/04 21:07:18 Cloning repo https://github.com/DanVerh/gitops-test
May 04 21:07:34 beyond beyond-sync-worker[2092]: 2025/05/04 21:07:34
May 04 21:07:34 beyond beyond-sync-worker[2092]: PLAY [Change hostname] ********************************************************
May 04 21:07:34 beyond beyond-sync-worker[2092]: TASK [Gathering Facts] ********************************************************
May 04 21:07:34 beyond beyond-sync-worker[2092]: ok: [20.39.48.49]
May 04 21:07:34 beyond beyond-sync-worker[2092]: TASK [hostname : Set a hostname] ***********************************************
May 04 21:07:34 beyond beyond-sync-worker[2092]: changed: [20.39.48.49]
May 04 21:07:34 beyond beyond-sync-worker[2092]: PLAY RECAP *********************************************************************
May 04 21:07:34 beyond beyond-sync-worker[2092]: 20.39.48.49               : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Рис. 4.9 Логи beyond-sync-worker сервісу з успішним виконанням автоматичної синхронізації

Автоматична синхронізація пройшла успішно.

Тепер перевіримо працездатність API. Почнемо з документації у Swagger. На рисунку 4.10 зображено те, що сервіс по-перше працює та доступний та документація у Swagger коректно завантажена та згенерована.

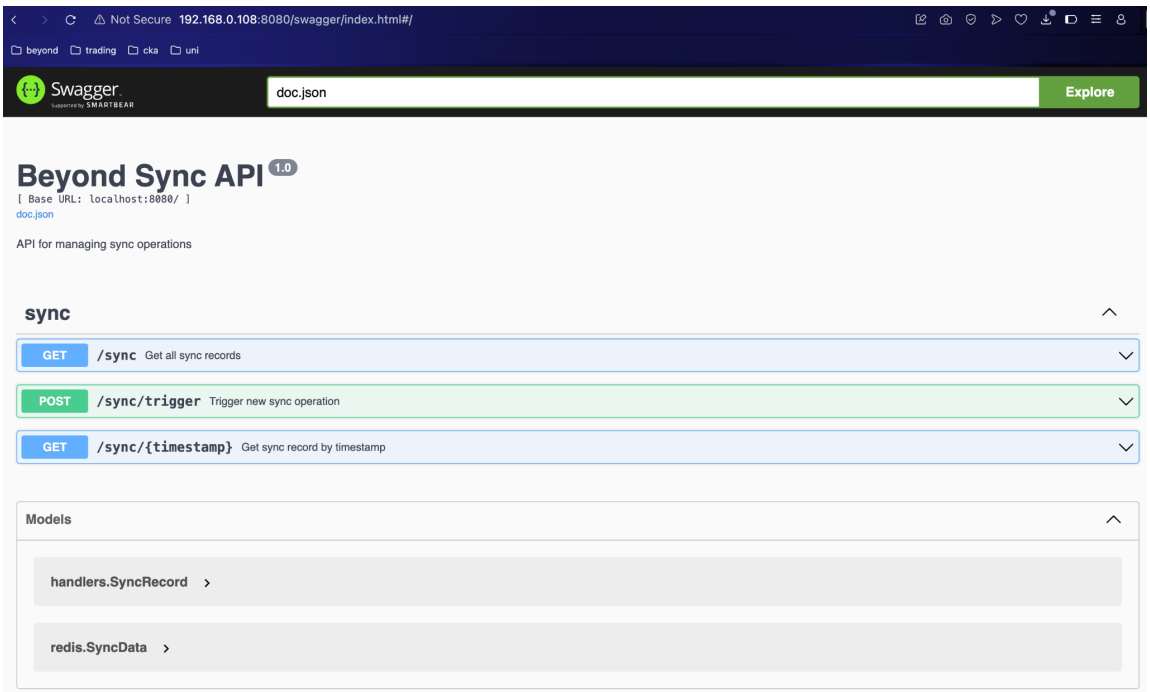

Рис. 4.10 Swagger beyond-sync-api

Тепер за допомогою Postman для зручності протестуємо API:

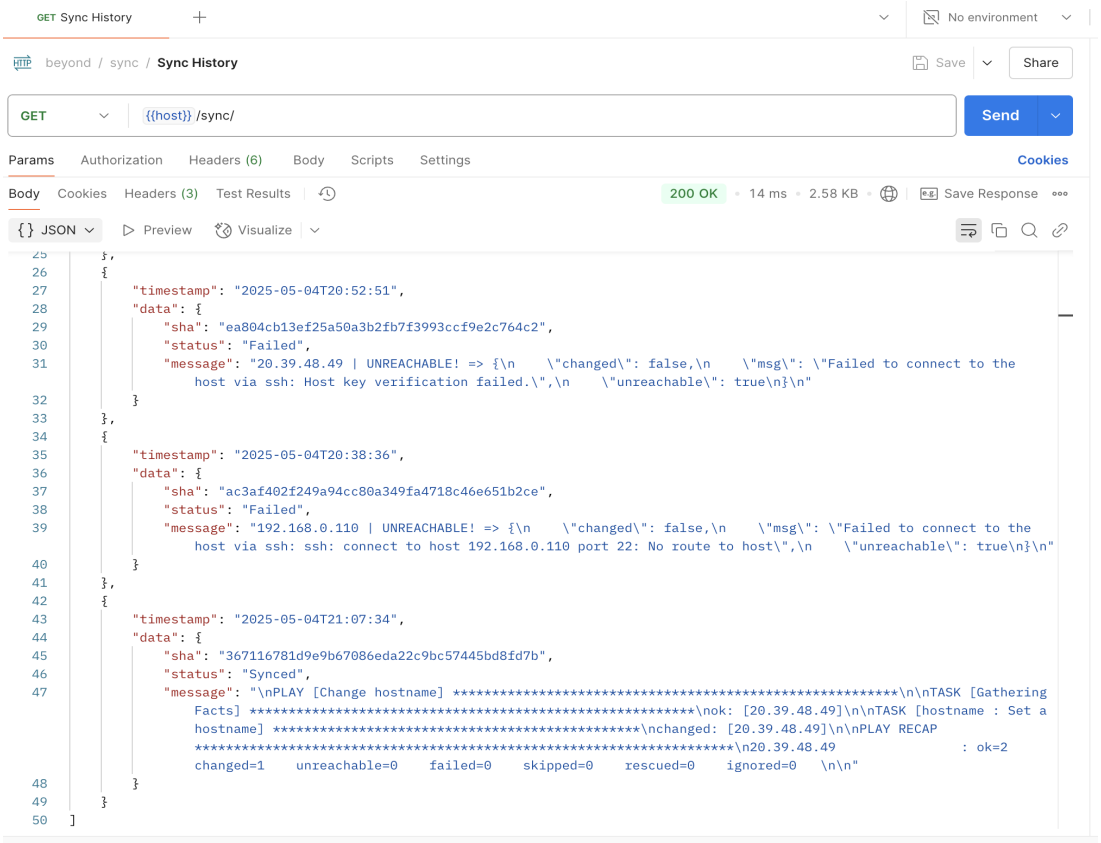

Рис. 4.11 Скріншот з Postman результату GET запиту на /sync beyond-sync-api сервісу

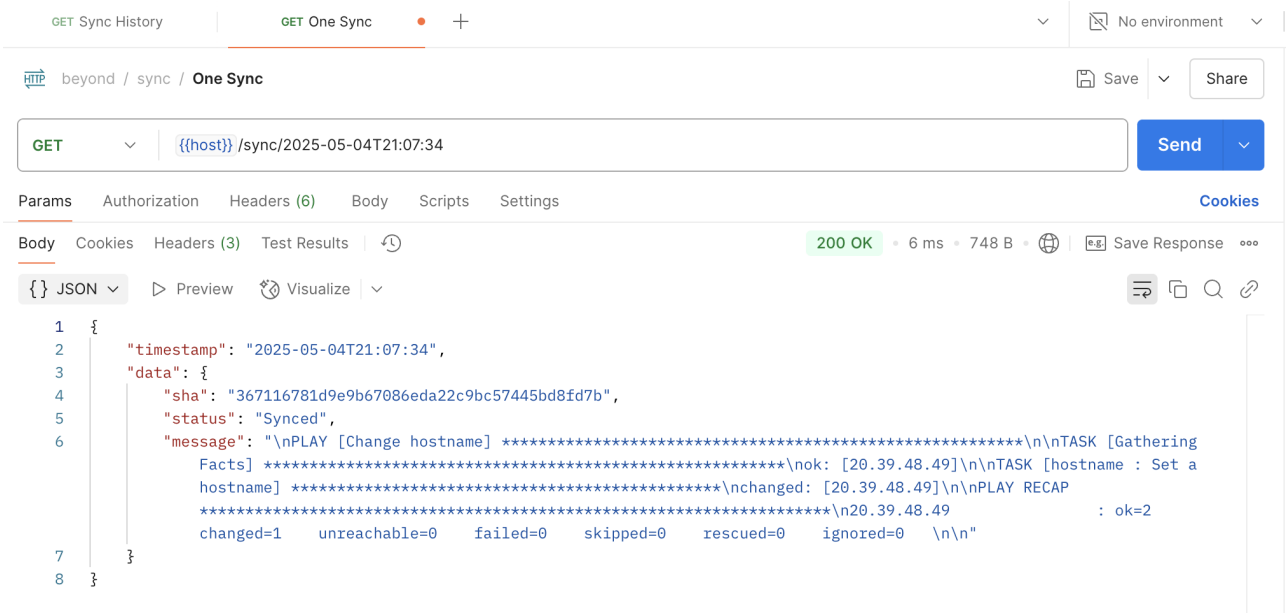

Рис. 4.12 Скріншот з Postman результату GET запиту на /sync/{timestamp} beyond-sync-api сервісу

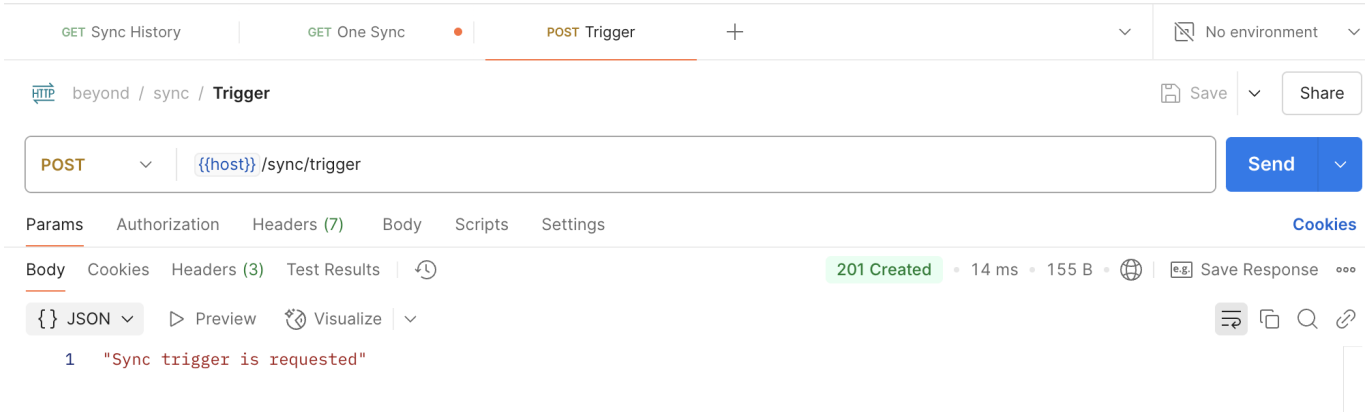

Рис. 4.13 Скріншот з Postman результату POST запиту на /sync/trigger beyond-sync-api сервісу

Після запуску ручного тригеру з API перевіряємо логи beyond-sync-api сервісу, які покажуть, що конфігурація була успішно застосована без змін. Також у цих логах можна побачити, що синхронізація була ініційована вручну:

```
daniil@beyond:/opt/beyond-sync$ sudo systemctl status beyond-sync-worker
● beyond-sync-worker.service - Beyond Sync Worker Service
     Loaded: loaded (/etc/systemd/system/beyond-sync-worker.service; enabled; preset: enabled)
     Active: active (running) since Sun 2025-05-04 20:38:31 UTC; 40min ago
   Main PID: 2092 (beyond-sync-wor)
      Tasks: 7 (limit: 1610)
     Memory: 22.5M (peak: 89.3M)
        CPU: 9.825s
     CGroup: /system.slice/beyond-sync-worker.service
             └─2092 /opt/beyond-sync/beyond-sync-worker

May 04 21:16:51 beyond beyond-sync-worker[2092]: 2025/05/04 21:16:51 Sync is triggered with manual trigger and commit hash value of 367116781d9e9b67086eda22c9bc57445bd8fd7b
May 04 21:16:51 beyond beyond-sync-worker[2092]: 2025/05/04 21:16:51 Cloning repo https://github.com/DanVerh/gitops-test
May 04 21:17:08 beyond beyond-sync-worker[2092]: 2025/05/04 21:17:08
May 04 21:17:08 beyond beyond-sync-worker[2092]: PLAY [Change hostname] ********************************************************
May 04 21:17:08 beyond beyond-sync-worker[2092]: TASK [Gathering Facts] *********************************************************
May 04 21:17:08 beyond beyond-sync-worker[2092]: ok: [20.39.48.49]
May 04 21:17:08 beyond beyond-sync-worker[2092]: TASK [hostname : Set a hostname] ************************************************
May 04 21:17:08 beyond beyond-sync-worker[2092]: ok: [20.39.48.49]
May 04 21:17:08 beyond beyond-sync-worker[2092]: PLAY RECAP *********************************************************************
May 04 21:17:08 beyond beyond-sync-worker[2092]: 20.39.48.49               : ok=2    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Рис. 4.14 Логи beyond-sync-worker сервісу з успішним виконанням вручну ініційованої синхронізації

І наостанок, звісно треба переконатись, чи була імплементована зміна у конфігурації на цільовому сервері:

```
daniil@target:~$ hostname
newhostname
```

Рис. 4.15 Змінена назва цільового хосту

Бачимо задовільний результат, який демонструє успішне вровадження конфігурації визначеної у Git Ansible проєкті за допомогої створеної GitOps інформаційної системи.

Висновок до розділу 4

У цьому розділі наведено повний перелік технічних, програмних і організаційних вимог, необхідних для ефективної роботи інформаційної системи для впровадження GitOps методології у процес конфігурації серверів. Зокрема, було визначено підтримувані операційні системи (Debian/Ubuntu на ARM64-архітектурі), необхідні компоненти (Redis, Ansible, Git, curl), мережеві порти (6379 та 8080), структуру проєкту Ansible та змінні середовища. Також окреслено обмеження: зокрема залежність від Redis і сувору вимогу до структури Ansible-проєкту. Описано потребу в стабільному мережевому з’єднанні, наявності доступу до цільових серверів по SSH. У якості завершального етапу було здійснено демонстрацію роботи сервісу, яка підтвердила відповідність системи заявленим вимогам і її готовність до практичного використання.

ВИСНОВКИ

У процесі виконання бакалаврської роботи була реалізована інформаційна система, що автоматизує процес конфігурації серверного середовища за допомогою методології GitOps. Система розгортається у вигляді сервісу, який дозволяє централізовано керувати інфраструктурою за допомогою Ansible-проєкту, зберігаючи зміни у системі контролю версій. Основне призначення - синхронізація конфігурацій у реальному часі та забезпечення контрольованого, безпечного і відтворюваного розгортання. Розроблений сервіс враховує важливі аспекти безпеки, підтримує цілодобову роботу, передбачає блокування при паралельних конфліктах і забезпечує базові засоби самовідновлення.

Перший розділ роботи був присвячений актуальності теми автоматизованого конфігурування, особливо в умовах зростаючої складності інфраструктур у DevOps-практиках. Було досліджено основні принципи GitOps, переваги такого підходу у порівнянні з традиційними методами та причини зростання його популярності серед компаній, що прагнуть швидкості, надійності та масштабованості. Також було розглянуто приклади аналогічних рішень, що дозволило виявити сильні та слабкі сторони існуючих інструментів.

Наступним етапом стало проектування цільової архітектури: було складено дерево цілей, сформульовано функціональні та нефункціональні вимоги, побудовано контекстні діаграми і визначено взаємодію між компонентами. Аналіз ієрархій допоміг обґрунтовано обрати оптимальні технічні рішення та сформувати логіку роботи системи.

Після завершення етапу проектування було здійснено вибір відповідних технологій, які найкраще відповідають поставленим цілям та вимогам до продуктивності, надійності й масштабованості системи. Основною мовою програмування для розробки бекенду обрано Go, завдяки її високій швидкодії, простоті у розгортанні та зручній роботі з багатопоточністю. Для забезпечення швидкого доступу до тимчасових даних і фіксації поточного стану процесів

використано Redis, як легку й ефективну систему in-memory зберігання. GitHub виступає у ролі джерела істини для конфігурацій, а для автоматизації процесів управління конфігураціями використовується Ansible, що дозволяє централізовано виконувати завдання на віддалених вузлах. Обрані технології були протестовані на сумісність, простоту інтеграції та стабільність роботи в умовах цільового середовища.

У технічній частині роботи описано реалізацію сервісу за допомогою мови програмування Go для серверної логіки, а також використання Redis для зберігання стану. Для автоматизації конфігурацій застосовується Ansible, а управління синхронізацією відбувається через періодичні запити до GitHub. Усі компоненти інтегруються у систему з урахуванням безперервного моніторингу, логування та резервного копіювання. Збірка і розгортання тестувались у середовищі ARM64 з використанням стандартних засобів Linux.

Завершальний етап включав демонстрацію функціональної системи, де було перевірено її роботу на практиці. Під час демонстрації було показано типові сценарії синхронізації змін у репозиторії з фактичним станом серверів, а також механізми блокування, обробки помилок і логування. Система виявила себе як стабільний, адаптивний і розширюваний інструмент, готовий до використання у реальному середовищі.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Alexis Richardson. GitOps goes mainstream - Flux CD boasts largest ecosystem. URL: https://www.cncf.io/blog/2023/12/01/gitops-goes-mainstream-flux-cd-boasts-largest-ecosystem/?utm_source=chatgpt.com (01.12.2023).

2. Anja Kammer. Implementing GitOps without Kubernetes. URL: https://www.innoq.com/en/articles/2025/01/gitops-kubernetes/?utm_source=chatgpt.com (08.01.2025).

3. Michael Levan, Kubernetes Consultor. Is GitOps Just for Kubernetes? URL: https://dev.to/thenjdevopsguy/is-gitops-just-for-kubernetes-4p8c (22.02.2022).

4. 6sense. Ansible market share. URL: https://6sense.com/tech/build-and-deployment-automation/ansible-market-share?utm_source=chatgpt.com (2025).

5. CNCF blog. CNCF GitOps microsurvey: learning on the job as GitOps goes mainstream. URL: https://www.cncf.io/blog/2023/11/07/cncf-gitops-microsurvey-learning-on-the-job-as-gitops-goes-mainstream/?utm_source=chatgpt.com (07.11.2023).

6. Mike Tyson of the Cloud (MToC). The Origins of Infrastructure as Code: A Brief History of DevOps. URL: https://medium.com/%40mike_tyson_cloud/the-origins-of-infrastructure-as-code-a-brief-history-of-devops-a883d8877f19 (23.04.2023).

7. Google Cloud GKE Guides. GitOps best practices. URL: https://cloud.google.com/kubernetes-engine/enterprise/config-sync/docs/concepts/gitops-best-practices?utm_source=chatgpt.com (17.04.2025).

8. Datanyze. ArgoCD market share. URL: https://www.datanyze.com/market-share/deployment--320/argocd-market-share?utm_source=chatgpt.com

9. ArgoCD Operation Manual. Declarative Setup. URL: https://argo-cd.readthedocs.io/en/stable/operator-manual/declarative-setup/

10. Github argo-cd/ui project. Argo CD UI. URL: https://github.com/argoproj/argo-cd/blob/master/ui/README.md

11. Bhushan Nemade. What is FluxCD? A Quick Guide to GitOps with FluxCD. URL: https://devtron.ai/blog/what-is-fluxcd/ (07.11.2024).

12. Weave GitOps. The Flux expansion pack from the founders of Flux. URL: https://docs.gitops.weaveworks.org

13. Flux Docs. Automate image updates to Git. URL: https://fluxcd.io/flux/guides/image-update/ (11.10.2024).

14. Selvam Raju. FluxCD: Introduction and Installation Demo. URL: https://medium.com/@selvamraju007/fluxcd-introduction-and-installation-demo-fb9fe0cb7555 (12.07.2023).

15. Flavius Dinu. 16 Most Useful Infrastructure as Code (IaC) Tools for 2025. URL: https://spacelift.io/blog/infrastructure-as-code-tools#1-spacelift (23.04.2025).

16. Spacelift Docs. Getting Started. URL: https://docs.spacelift.io/vendors/ansible/getting-started

17. Шершньова З. Є. Стратегічне управління. Підручник. - 2-ге вид., перероб. і доп. К.: КНЕУ, 2004. - 699 с.

18. Олександр Кононенко. Що краще моноліт чи мікросервіси? Як обрати архітектуру проєкту? URL: https://iampm.club/ua/blog/shho-krashhe-monolit-chi-mikroservisi-yak-obrati-arhitekturu-projektu/?utm_source=chatgpt.com (17.11.2023).

19. Олександр Хотемський. Serverless архітектура та її застосування в автоматизації тестування. URL: https://www.slideshare.net/slideshow/serverless-115187012/115187012?utm_source=chatgpt.com (18.09.2018).

20. EdrawMax. The Complete Guide To Understand IDEF Diagram. URL: https://www.edrawmax.com/article/the-complete-guide-to-understand-idef-diagram.html?utm_source=chatgpt.com

21. Paul Jansen. TIOBE Index for April 2025. URL: https://www.tiobe.com/tiobe-index/ (04.2025).

22. Sophia Iroegbu. Python in DevOps: Automation, Efficiency, and Scalability. URL: https://dev.to/sophyia/python-in-devops-automation-efficiency-and-scalability-2h10 (18.01.2025).

23. Gabriel Silva. Building REST APIs with Node.js: Advantages and Disadvantages. URL: https://medium.com/@socialgabrielgomes/building-rest-apis-with-node-js-advantages-and-disadvantages-1698a9e3e982 (18.02.2023).

24. Mohammed Tawfik. Rust Language: Pros, Cons, and Learning Guide. URL: https://medium.com/@apicraft/rust-language-pros-cons-and-learning-guide-594e8c9e2b7c (19.02.2024).

25. Bipin Mishra. GoLang - Pros and Cons of Go language. URL: https://www.mindinventory.com/blog/pros-and-cons-programming-in-golang/ (26.10.2023).

26. Sebastián Ramírez. Using FastAPI to Build Python Web APIs. URL: https://realpython.com/fastapi-python-web-apis/#why-fastapi-is-the-fastest-python-web-framework

27. Express.js - Official Documentation. URL: https://expressjs.com

28. Actix Web - Official Documentation. URL: https://actix.rs

29. Gin Web Framework - Official Documentation. URL: https://gin-gonic.com

30. Rajat Singh. Redis Cache and its use cases for Modern Application. URL: https://www.einfochips.com/blog/redis-cache-and-its-use-cases-for-modern-application/ (06.03.2024).

31. Arthur C. Codex. PostgreSQL in the Microservices Architecture. URL: https://reintech.io/blog/postgresql-microservices-architecture (06.05.2024).

32. VIBIDSOFT PVT LTD. The Complete Guide to Microservices with MongoDB, Node.js, and Express. URL: https://www.linkedin.com/pulse/complete-guide-microservices-mongodb-nodejs-express-vibidsoft/ (21.08.2023).

33. Ably blog. Pub/Sub use cases: When to use the Pub/Sub pattern. URL: https://ably.com/blog/pub-sub-pattern-examples (10.05.2023).

34. AltexSoft blog. The Good and the Bad of Apache Kafka Streaming Platform. URL: https://www.altexsoft.com/blog/apache-kafka-pros-cons/ (21.10.2022).

35. Eran Levy. Kafka vs. RabbitMQ: Architecture, Performance & Use Cases. URL: https://www.upsolver.com/blog/kafka-versus-rabbitmq-architecture-performance-use-case (01.02.2022).

36. Abhirup Acharya. Redis Pub-Sub or Kafka: Choosing the Right Tool for Your Use Case. URL: https://medium.com/@abhirup.acharya009/redis-pub-sub-or-kafka-choosing-the-right-tool-for-your-use-case-7241bfa87690 (10.01.2024).

37. Praveen Dandu. A Comprehensive Ansible Tutorial for Beginners - Part 1. URL: https://praveendandu24.medium.com/a-comprehensive-ansible-tutorial-for-beginners-part-1-ab97a3c2df1e (28.07.2023).

38. Elle Krout. A Beginner's Guide to Chef. URL: https://www.linode.com/docs/guides/beginners-guide-chef/ (04.03.2021).

39. Ezeelive Technologies. What is Puppet IT Automation - Puppet Pros and Cons. URL: https://ezeelive.com/puppet-pros-cons/ (09.04.2023).

40. Go Tutorials. Tutorial: Developing a RESTful API with Go and Gin. URL: https://go.dev/doc/tutorial/web-service-gin

41. Francisco Mendes. Using Redis Pub/Sub with Golang. URL: https://dev.to/franciscomendes10866/using-redis-pub-sub-with-golang-mf9 (03.09.2021).

АНОТАЦІЯ

Верхутін Д.Є., Василюк А.С., (керівник). Інформаційна система для впровадження GitOps методології у процес конфігурації серверів. Бакалаврська кваліфікаційна робота. - Національний університет «Львівська політехніка», Львів, 2025.

Розширена анотація.

Інформаційна система для впровадження GitOps методології у процес конфігурації серверів - сервіс, що автоматизує керування інфраструктурою шляхом інтеграції з системами контролю версій, зокрема Git. GitOps - це підхід до інфраструктурної автоматизації, який передбачає, що стан системи зберігається у вигляді декларативних конфігурацій у Git-репозиторії. Усі зміни в інфраструктурі відслідковуються через коміти, що дозволяє забезпечити контроль, відтворюваність і безперервність у процесах розгортання та супроводу серверних середовищ. Завдяки цьому досягається висока прозорість і аудитність змін, зменшується ймовірність людських помилок та підвищується надійність систем [1].

Система дозволяє зчитувати зміни у Git-репозиторії та автоматично застосовувати їх до цільового середовища за допомогою інструментів конфігураційного менеджменту, таких як Ansible. API-інтерфейс надає адміністраторам можливість переглядати статус синхронізації, аналізувати лог змін, та вручну ініціювати процеси розгортання. Для обміну повідомлення між сервісами і зберігання актуального стану використовувалась система Redis, що дозволяє зменшити навантаження на бекенд і пришвидшити доступ до даних. Серверна логіка реалізована мовою Go, що забезпечує високу продуктивність та ефективну обробку запитів.

Об’єкт дослідження - процес впровадження GitOps методології у процес конфігурації серверів.

Предмет дослідження - методи і засоби, які необхідні для провадження GitOps методології у процес конфігурації серверів.

Мета дослідження: розроблення інформаційної системи для впровадження GitOps методології у процес конфігурації серверів

Результати дослідження:

У результаті виконання бакалаврської роботи було розроблено й впроваджено інформаційну систему для автоматизації конфігурації серверного середовища за методологією GitOps, яка забезпечує безпечне, контрольоване та відтворюване розгортання інфраструктури. Реалізація на базі Go, Redis, GitHub та Ansible дозволила досягти високої продуктивності, надійності й масштабованості. Проведене проектування, обґрунтований вибір технологій і успішне тестування у практичному середовищі підтвердили ефективність системи та її готовність до використання в реальних умовах.

Ключові слова - GitOps, DevOps, Ansible, мікросервіси, конфігурація, сервер.

Перелік використаних літературних джерел.

1. RedHat Topics: DevOps. What is GitOps. URL: https://www.redhat.com/en/topics/devops/what-is-gitops#:~:text=GitOps%20is%20a%20set%20of%20principles%20that%20guide%20your%20workflow,to%20a%20previously%20manual%20process. (27.03.2025).

ANNOTATION

Verkhutin D.Ye., Vasilyuk A.S., (supervisor). Information system for implementing GitOps methodology in the server configuration process. Bachelor's thesis - Lviv Polytechnic National University, Lviv, 2025.

Extended annotation.

The information system for implementing GitOps methodology in the server configuration process is a service that automates infrastructure management through integration with version control systems, particularly Git. GitOps is an approach to infrastructure automation based on the principle that the system's state is stored as declarative configurations in a Git repository. All infrastructure changes are tracked via commits, which ensures control, reproducibility, and continuity in deployment and maintenance processes of server environments. This results in high transparency and auditability of changes reduces the likelihood of human errors and increases system reliability [1].

The system allows monitoring changes in the Git repository and automatically applying them to the target environment using configuration management tools such as Ansible. The API interface provides administrators with the ability to view synchronization status, analyze change logs and manually initiate deployment processes. Redis is used for inter-service communication and storing the current state, which reduces backend load and speeds up data access. The server logic is implemented in Go, ensuring high performance and efficient request handling.

Object of research: the process of implementing the GitOps methodology in server configuration.

Subject of research: the methods and tools necessary for implementing the GitOps methodology in the server configuration process.

Purpose of the research: to develop an information system for implementing the GitOps methodology in the server configuration process.

Research results:

As a result of the bachelor's work, an information system was developed and implemented to automate the configuration of the server environment using the GitOps methodology, ensuring secure, controlled, and reproducible infrastructure deployment. The implementation based on Go, Redis, GitHub, and Ansible enabled high performance, reliability, and scalability. The system design, justified technology selection, and successful testing in a practical environment confirmed the effectiveness of the system and its readiness for real-world use.

Keywords: GitOps, DevOps, Ansible, microservices, configuration, server.

List of references:

RedHat Topics: DevOps. What is GitOps. URL: https://www.redhat.com/en/topics/devops/what-is-gitops#:~:text=GitOps%20is%20a%20set%20of%20principles%20that%20guide%20your%20workflow,to%20a%20previously%20manual%20process. (27.03.2025).

ДОДАТКИ

Додаток А

README.md https://github.com/gitops-beyond/beyond-sync

Beyond Sync - GitOps for Ansible Configuration Management

Beyond Sync is a groundbreaking GitOps tool designed to bring GitOps methodology beyond Kubernetes and into the world of Ansible configuration management. It's possibly the first GitOps tool specifically designed for configuration management outside of Kubernetes environments.

What is Beyond Sync?

Beyond Sync enables continuous deployment and automation of Ansible configurations using GitOps principles. It monitors your Git repository and automatically applies changes to your infrastructure when modifications are detected in your Ansible playbooks and configurations.

Key Features:

- Automated sync of Ansible configurations
- GitOps workflow for infrastructure management
- Real-time status monitoring
- REST API for integration
- Redis-backed state management

Installation

Prerequisites

Installation server:

- Linux **ARM** server (Debian or CentOS/RHEL)
- Ports 6379, 8080 opened
- Can access Ansible target servers from inventory
- Contains SSH keys required for Ansible project

Ansible tracked project:

- Hosted in Github
- Github PAT
- Ansible project in `ansible/` dir of the repo
- Inventory in `ansible/inventory`
- Playbook in `ansible/playbook.yml`

Installation Steps

1. Clone the repo:

```
https://github.com/gitops-beyond/beyond-sync.git
```

2. Run the installation script as root:

```
sudo ./install.sh
```

Input the information about the GitHub repository with Ansible project you want to track
3. Check that installed services are running:

```
sudo systemctl status beyond-sync-api
bsudo systemctl status beyond-sync-worker
```

Release

Release Steps

1. Run the packaging script:

```
./package.sh
```

2. In `beyond-sync` repo create release and archive generated tar.gz archive; in `install.sh` change the release download path

Swagger

To check API documentation one can access swagger with following link:

```
<beyond-sync host>:8080/swagger/index.html#/
```

To generate/update swagger docs one can run:

```
go get -u github.com/swaggo/gin-swagger
swag init -g cmd/api/main.go -o docs
```

Додаток Б

Елементи коду системи.

internal/sync/sync.go:

```
package sync

import (
 "context"
 "log"
 "sync"
 "time"

 "github.com/gitops-beyond/beyond-sync/internal/ansible"
 "github.com/gitops-beyond/beyond-sync/internal/redis"
)


func Sync() {
 // Create Webhook object
 w := Webhook{}
 // Define a variable to store sha value
 sha := ""
 // Define a variable to use a lock of routine
 var lock sync.Mutex

 // Define a collection of goroutines
 ctx := context.Background()
 var wg sync.WaitGroup
 wg.Add(2)

 // First goroutine
 // Automatic sync
 go func() {
       // End goroutine when function ends
       defer wg.Done()
       // Inifinite loop
       for {
             // Save last sha value
             newSha, err := w.GetLastCommit()
             if err != nil {
                   log.Fatal(err)
             }

             // Compare sha value from current loop with previous one
             if sha != newSha {
                   // Lock the process to not create a conflict between syncs
                   lock.Lock()
                   // Update sha value with current loop's one
                   sha = newSha
```

```
                        log.Printf("Sync is triggered with new commit hash value of
%s", sha)
                        // Run Ansible playbook
                        ansible.RunAnsibleSync(sha)
                        // Unlock the process to use concurrency
                        lock.Unlock()
                  // Make 30 second pause, if repo has not been updated
                  } else {
                        time.Sleep(30 * time.Second)
                  }
            }
      }()

      // Second goroutine
      // Manul sync
      go func() {
            // End goroutine when function ends
            defer wg.Done()
            // Subscribe for Redis Pub/Sub channel
            sub, err := redis.Subscribe()
            if err != nil {
                  log.Printf("Redis connection error: %v", err)
                  return
            }
            // Close the connection after function ends
            defer sub.Close()

            // Infinite loop of reading messages from Redis channel
            for {
                  // Get the message from channel
                  msg, err := sub.ReceiveMessage(ctx)
                  if err != nil {
                        log.Printf("Redis subscription error: %v", err)
                        return
                  // If the message is there run the sync
                  } else if len(msg.Payload) > 0 {
                        // Lock the process to not create a conflict between syncs
                        lock.Lock()
                        log.Printf("Sync is triggered with manual trigger and commit
hash value of %s", sha)
                        // Run Ansible playbook
                        ansible.RunAnsibleSync(sha)
                        // Unlock the process to use concurrency
                        lock.Unlock()
                  }
            }
      }()

      // Block the main goroutine until all goroutines have completed their execution
      wg.Wait()
     }
```

internal/redis/redis.go:

```
package redis

import (
   "context"
   "encoding/json"
   "errors"
   "fmt"
   "os"
   "time"

   "github.com/redis/go-redis/v9"
)

// Struct definition to set sync parameters to save in Redis
type SyncData struct {
   Sha string `json:"sha"`
   Status string `json:"status"`
   Message string `json:"message"`
}

// Save sync record to Redis
func AddSyncRecord(sha string, status string, message string) {
   // Create client connection with Redis
   rdb := redis.NewClient(&redis.Options{
       Addr:     fmt.Sprintf("%s:6379", os.Getenv("REDIS_HOST")),
       Password: "",
       DB:       0,
   })

   ctx := context.Background()
   // Close the connection after function execution
   defer rdb.Close()

   // Check connection with Redis
   if err := rdb.Ping(ctx).Err(); err != nil {
       fmt.Printf("Error connecting to Redis: %v\n", err)
       return
   }

   // Set sync time as key
   key := time.Now().Format("2006-01-02T15:04:05")
   // Put sync parameters to save into variable
   value, err := json.Marshal(SyncData{sha, status, message})
   if err != nil {
       fmt.Printf("Error encoding json value: %v\n", err)
       return
   }
```

```
    // Set the value using the context
    if err := rdb.Set(ctx, string(key), string(value), 0).Err(); err != nil {
        fmt.Printf("Error setting value in Redis: %v\n", err)
        return
    }
}

// Get sync record/records
func GetSyncRecords(query string) (map[string]SyncData, error) {
    // Create client connection with Redis
    rdb := redis.NewClient(&redis.Options{
        Addr:     fmt.Sprintf("%s:6379", os.Getenv("REDIS_HOST")),
        Password: "",
        DB:       0,
    })

    ctx := context.Background()
    // Close the connection after function execution
    defer rdb.Close()

    // Check connection first
    if err := rdb.Ping(ctx).Err(); err != nil {
        return nil, fmt.Errorf("failed connecting to Redis: %v", err)
    }

    // Get all keys from Redis
    keys, err := rdb.Keys(ctx, query).Result()
    if err != nil {
        return nil, fmt.Errorf("failed getting keys from Redis: %v", err)
    } else if len(keys) == 0 {
        return nil, errors.New("key not found")
    }

    // Map to store data pulled from Redis
    result := make(map[string]SyncData)

    // Iterate through keys
    for _, key := range keys {
        // Get value from Redis of each key
        value, err := rdb.Get(ctx, key).Result()
        if err != nil {
            continue
        }

        // Put value into SyncData struct
        var syncData SyncData
        if err := json.Unmarshal([]byte(value), &syncData); err != nil {
            continue
        }

        // Put the value into map of Redis key-values
```

```
        result[key] = syncData
    }

    return result, nil
}

// Redis message publishing
func PublishMessage() error {
    // Create client connection with Redis
    rdb := redis.NewClient(&redis.Options{
        Addr:     fmt.Sprintf("%s:6379", os.Getenv("REDIS_HOST")),
        Password: "",
        DB:       0,
    })

    ctx := context.Background()
    // Close the connection after function execution
    defer rdb.Close()

    // Check connection first
    if err := rdb.Ping(ctx).Err(); err != nil {
        return fmt.Errorf("failed connecting to Redis: %v", err)
    }

    // Publish message to "triggers channel"
    err := rdb.Publish(ctx, "triggers", `sync trigger`).Err()
    if err != nil {
        return fmt.Errorf("failed publishing message in Redis: %v", err)
    }

    return nil
}

// Redis channel subscription
func Subscribe() (*redis.PubSub, error) {
    // Create client connection with Redis
    rdb := redis.NewClient(&redis.Options{
        Addr:     fmt.Sprintf("%s:6379", os.Getenv("REDIS_HOST")),
        Password: "",
        DB:       0,
    })

    ctx := context.Background()
    // We do not close connection here to be left subscribed to channel

    // Check connection first
    if err := rdb.Ping(ctx).Err(); err != nil {
        return nil, fmt.Errorf("failed connecting to Redis: %v", err)
    }

    // Subscribe to channel
    sub := rdb.Subscribe(ctx, "triggers")
```

```
   return sub, nil
}
```

api/handlers/sync.go:

```
package handlers

import (
   "github.com/gin-gonic/gin"
   "github.com/gitops-beyond/beyond-sync/internal/redis"
)

// SyncRecord represents a single sync operation result
type SyncRecord struct {
   Timestamp string          `json:"timestamp"`
   Data      redis.SyncData `json:"data"`
}

// SyncRecords is a collection of sync responses
type SyncRecords []SyncRecord

// GetAllSyncs godoc
// @Summary      Get all sync records
// @Description  Retrieves all sync records from Redis
// @Tags         sync
// @Accept       json
// @Produce      json
// @Success      200  {array}   SyncRecord
// @Router       /sync [get]
func GetAllSyncs(c *gin.Context) {
   redisRecords, err := redis.GetSyncRecords("*")
   if err != nil && err.Error() == "key not found"{
       c.JSON(404, gin.H{"error": err.Error()})
       return
   } else if err != nil {
       c.JSON(500, gin.H{"error": err.Error()})
       return
   }

   // Convert Redis records to response format
   response := make(SyncRecords, 0)
   for timestamp, value := range redisRecords {
       sync := SyncRecord{
           Timestamp: timestamp,
           Data:     value,
       }
       response = append(response, sync)
   }

   c.JSON(200, response)
```

```
    }

    // GetSyncByDate godoc
    // @Summary      Get sync record by timestamp
    // @Description  Retrieves a specific sync record by its timestamp
    // @Tags         sync
    // @Accept       json
    // @Produce      json
    // @Param        timestamp   path      string  true  "Timestamp of the sync
record"
    // @Success      200  {object}  SyncRecord
    // @Router       /sync/{timestamp} [get]
    func GetSyncByDate(c *gin.Context) {
       redisKey := c.Param("timestamp")
       redisValue, err := redis.GetSyncRecords(redisKey)
       if err != nil && err.Error() == "key not found" {
           c.JSON(404, gin.H{"error": err.Error()})
           return
       } else if err != nil {
           c.JSON(500, gin.H{"error": err.Error()})
           return
       }

       response := SyncRecord{
           Timestamp: redisKey,
           Data: redisValue[redisKey],
       }

       c.JSON(200, response)
    }

    // TriggerSync godoc
    // @Summary      Trigger new sync operation
    // @Description  Triggers a new synchronization operation
    // @Tags         sync
    // @Accept       json
    // @Produce      json
    // @Success      201  {string}  string    "Sync trigger is requested"
    // @Router       /sync/trigger [post]
    func TriggerSync(c *gin.Context) {
       err := redis.PublishMessage()
       if err != nil {
           c.JSON(500, gin.H{"error": err.Error()})
           return
       }

       c.JSON(201, "Sync trigger is requested")
    }
```