Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

СИСТЕМА КОНТРОЛЮ ДОСТУПУ В KUBERNETES НА ОСНОВІ SMART CONTRACTS

ACCESS CONTROL SYSTEM IN KUBERNETES BASED ON SMART CONTRACTS

спеціальність 122 Комп'ютерні науки

освітня програма «Комп'ютерні науки»

Виконав: здобувач вищої освіти
групи КН-41
Віннічук Богдан Петрович
_____
(підпис)
Керівник: к.т.н., доцент
Кошелюк Віктор Андрійович
_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«___» _____ 2026 р.
Гарант освітньої програми:
д.пед.н., професор
Тулашвілі Юрій Йосипович
_____
(підпис)

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерних наук*
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *122 Комп'ютерні науки*
Освітня програма: *«Комп'ютерні науки»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Валерій ЛІЩИНА
«16» січня 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА
ПЕРШОГО (БАКАЛАВРСЬКОГО) РІВНЯ ВИЩОЇ ОСВІТИ

Віннічук Богдан Петрович
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Система контролю доступу в Kubernetes на основі Smart Contracts»
Керівник к.т.н., доцент Кошелюк Віктор Андрійович
затверджені наказом закладу вищої освіти від «*16*» *січня 2026* р. № 32/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи *«02» червня 2026 р.*
3. Вихідні дані до роботи: *аналітичне дослідження систем контролю доступу та технології smart contracts*
4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити):
Вступ
Розділ 1. Аналіз предметної області
Розділ 2. Специфікація вимог до розроблюваної системи
Розділ 3. Розробка системи контролю доступу
Розділ 4. SEO інформаційно-комп'ютерної системи
Висновки
Додатки
5. Перелік графічного матеріалу:
10 рисунків;
лістинг коду;
презентація.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми за темою роботи та постановка завдань дослідження	*Кошелюк В. А.*		
Теоретичне дослідження	*Кошелюк В. А.*		
Практична реалізація	*Кошелюк В. А.*		
Спеціальні розрахунки	*Кошелюк В. А.*		
Показник запозичень тексту	_____%		
Інструментальна перевірка	*Кошелюк В. А.*		
Нормоконтроль	*Сачук В. О.*		
Гарант ОПП	*Тулашвілі Ю. Й.*		

7. Дата видачі завдання *«16» січня 2026 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	*Провести огляд літературних джерел по темі кваліфікаційної роботи*	*до 17.02.2026 р*	
2	*Провести аналіз загальної проблеми і вибір напрямків дослідження*	*до 28.02.2025 р.*	
3	*Розробити функціональну схему роботи програмного продукту*	*до 20.04.2026 р*	
4	*Описати засоби розробки об'єкта проектування*	*до 25.04.2026 р.*	
5	*Практична реалізація об'єкта проектування*	*до 10.05.2026 р.*	
6	*Провести спеціальні розрахунки*	*до 18.05.2026 р.*	
7	*Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру*	*до 21.05.2026 р.*	

Здобувач вищої освіти _____ Богдан ВІННІЧУК

Керівник роботи _____ Віктор КОШЕЛЮК

АНОТАЦІЯ

Віннічук Б. П. Система контролю доступу в Kubernetes на основі Smart Contracts. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерні науки». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків.

Перший розділ кваліфікаційної роботи присвячено дослідженню актуального стану досліджуваної проблематики, обґрунтуванню обраного напряму роботи, а також визначенню та відбору методів і засобів проектування. У другому розділі розглянуто та обґрунтовано підходи й технології для вирішення поставлених завдань. Також наведено функціонально-структурну схему об'єкта проектування та ER-діаграму бази даних. Третій розділ охоплює безпосередню розробку програмного забезпечення та практичне втілення проєктованого об'єкта. Четвертий розділ містить SEO-оптимізацію інформаційно-комп'ютерної системи. У висновках підсумовано результати, отримані в ході виконання всіх попередніх розділів роботи.

Ключові слова: безпека інфраструктури, блокчейн, контроль доступу, Kubernetes.

ANNOTATION

Bogdan Vinnichuk. Access control system in Kubernetes based on Smart Contracts. Manuscript.

Bachelor's qualifying thesis of the OP «Computer Sciences». Lutsk National Technical University. Lutsk, 2026.

The bachelor's qualification work consists of an introduction, four chapters, conclusions, a list of used sources and appendices.

The first chapter of this thesis is devoted to examining the current state of the research topic, justifying the chosen research direction, and identifying and selecting design methods and tools. The second chapter examines and justifies the approaches and technologies for solving the set tasks. It also presents a functional-structural diagram of the design object and an ER diagram of the database. The third chapter covers the actual software development and practical implementation of the designed system. The fourth chapter contains SEO optimization of the information and computer system. The conclusions summarize the results obtained during the completion of all previous chapters of the work.

Keywords: infrastructure security, blockchain, access control, Kubernetes.

ЗМІСТ

ВСТУП

Актуальність теми. Стрімкий розвиток хмарних обчислень та контейнерних технологій зумовив широке впровадження платформи Kubernetes як стандарту де-факто для оркестрації контейнеризованих застосунків. У середовищах, де розгортаються мікросервісні архітектури з десятками і сотнями взаємодіючих компонентів, забезпечення надійного контролю доступу набуває критичного значення для захисту конфіденційних даних та цілісності інфраструктури.

Традиційні механізми контролю доступу на основі RBAC (Role-Based Access Control) у Kubernetes мають ряд суттєвих обмежень: централізоване зберігання політик, відсутність прозорості в аудиті змін та вразливість до несанкціонованого втручання з боку привілейованих адміністраторів. Технологія блокчейн та смарт-контракти пропонують принципово новий підхід до вирішення цих проблем, забезпечуючи децентралізоване, незмінне та прозоре управління правилами доступу. Інтеграція смарт-контрактів із системою контролю доступу Kubernetes дозволяє усунути єдину точку відмови, автоматизувати виконання політик безпеки та забезпечити незаперечний журнал аудиту. Таким чином, розробка такої системи є актуальним завданням, що відповідає сучасним викликам у сфері кібербезпеки та хмарних технологій.

Метою кваліфікаційної роботи бакалавра є розробка та дослідження системи контролю доступу в середовищі Kubernetes на основі смарт-контрактів, що забезпечує децентралізоване, прозоре та захищене від несанкціонованих змін управління політиками безпеки.

Об’єктом дослідження є процес управління доступом до ресурсів у розподілених контейнерних середовищах на базі Kubernetes. Зокрема, досліджуються механізми автентифікації, авторизації та аудиту, що функціонують у взаємодії з децентралізованою блокчейн-інфраструктурою.

Предметом дослідження є методи та засоби інтеграції смарт-контрактів як механізму управління політиками доступу в Kubernetes-кластерах.

Розглядаються архітектурні підходи до побудови такої системи, алгоритми взаємодії між компонентами Kubernetes та смарт-контрактами, а також показники ефективності та безпеки запропонованого рішення.

Завдання дослідження:

– проаналізувати існуючі підходи до контролю доступу в Kubernetes та виявити їх ключові недоліки з точки зору безпеки та прозорості аудиту;

– розробити архітектуру системи контролю доступу в Kubernetes на основі смарт-контрактів, компоненти взаємодії між API та блокчейн-мережею;

– реалізувати прототип системи, що включає смарт-контракти для управління ролями та правами доступу, а також webhook-компонент для інтеграції з механізмом авторизації Kubernetes;

– провести тестування та оцінювання розробленої системи за показниками безпеки, затримки обробки запитів і надійності в порівнянні зі стандартним механізмом RBAC у Kubernetes.

Новизна роботи. Наукова новизна кваліфікаційної роботи полягає у розробці концепції та архітектури гібридної системи контролю доступу, яка поєднує нативні механізми Kubernetes з децентралізованим виконанням політик безпеки на основі смарт-контрактів. На відміну від існуючих рішень, запропонований підхід забезпечує незмінність журналу аудиту, усуває залежність від єдиного адміністратора та надає можливість криптографічно верифікованого підтвердження кожного рішення щодо доступу, що є принципово новим для Kubernetes-екосистеми.

Практична цінність. Результати роботи мають практичне значення для організацій, що використовують Kubernetes у критично важливих середовищах з підвищеними вимогами до безпеки (compliance). Розроблений прототип може бути застосований у фінансових установах, медичних системах та державних хмарних інфраструктурах, де необхідно забезпечити незаперечний аудит дій з привілейованим доступом. Практична цінність підтверджується можливістю безпосереднього розгортання розробленого webhook-компонента у реальних Kubernetes-кластерах без модифікації вихідного коду платформи.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз сучасного стану проблеми

Стрімкий розвиток контейнерних технологій та мікросервісної архітектури призвів до масового впровадження платформи Kubernetes як де-факто стандарту оркестрації контейнерів у корпоративних та хмарних середовищах. За даними Cloud Native Computing Foundation (CNCF) на 2024 рік, понад 0,96 організацій використовують або оцінюють Kubernetes для виробничих навантажень. Проте разом із поширенням платформи гостро постала проблема забезпечення надійного та гранульованого контролю доступу до ресурсів кластера.

Сучасні Kubernetes-середовища характеризуються надзвичайною динамічністю: тисячі подів, сервісів та неймспейсів можуть створюватися і знищуватися протягом секунд. Традиційні механізми управління доступом, розраховані на статичні інфраструктури, не справляються з такою мінливістю. Виникає потреба у принципово нових підходах, які поєднують гнучкість, автоматизацію та криптографічну верифікацію дозволів [1].

Вбудована система рольового контролю доступу (RBAC) Kubernetes, введена як стабільна у версії 1.8, на сьогодні є основним інструментом управління правами. Вона дозволяє визначати ролі (Role, ClusterRole) та прив’язувати їх до суб’єктів (ServiceAccount, User, Group) через об’єкти RoleBinding та ClusterRoleBinding. Незважаючи на широке застосування, RBAC має суттєві архітектурні обмеження.

По-перше, централізоване зберігання політик у etcd створює єдину точку відмови та довіри. Усі рішення про доступ залежать від цілісності API-сервера, і будь-яке його компрометування дає зловмиснику повний контроль над правами доступу в кластері. По-друге, відсутня нативна підтримка динамічних атрибутів: RBAC оперує статичними ролями і не враховує контекстуальні чинники – час доби, поточне навантаження, геолокацію запиту чи рівень ризику сесії.

Модель атрибутивного контролю доступу (ABAC), яка теоретично покриває зазначені недоліки, у Kubernetes реалізована обмежено і фактично витіснена RBAC. Зовнішні системи авторизації, підключені через Webhook Authorizer, дозволяють реалізувати ABAC-логіку, однак вносять затримки, ускладнюють налагодження та потребують підтримки окремої інфраструктури. Крім того, жоден із вбудованих механізмів не забезпечує незмінного аудиторського журналу, придатного для форензики чи відповідності регуляторним вимогам [2].

Смарт-контракти – самовиконувані програми, розгорнуті у розподіленому реєстрі (блокчейні) – пропонують якісно відмінну парадигму управління доступом. Ключові властивості, що роблять їх привабливими для систем авторизації: детермінованість виконання, криптографічна незмінність коду та стану, прозорість і верифікованість логіки, а також відсутність централізованого оператора. На рисунку 1.1 наведено архітектура функціональності управління з використанням смарт-контракту.

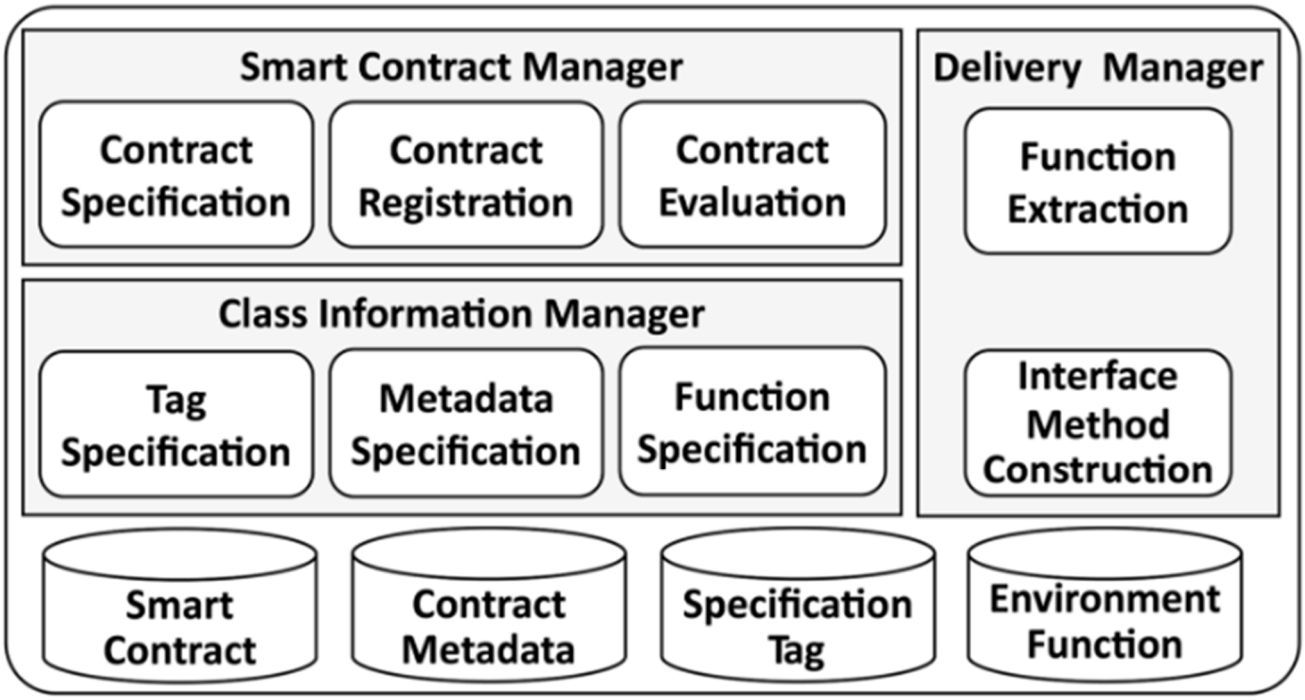

Рисунок 1.1 – Архітектура функціональності управління [3]

Платформи Ethereum та сумісні EVM-мережі (Polygon, Hyperledger Besu) дозволяють реалізовувати складні логіки управління доступом через контракти на Solidity. Зокрема, стандарт ERC-5114 (Soulbound Tokens) та концепція Decentralized Identifiers (DID) за специфікацією W3C відкривають шлях до самосуверенної ідентифікації сервісних акаунтів Kubernetes [4]. Це означає, що права доступу можуть бути прив'язані до криптографічних ідентичностей, верифікованих незалежно від стану API-сервера.

Дослідження останніх років демонструють зростаючий інтерес академічної спільноти до blockchain-based access control. Роботи сучасних дослідників показують, що використання смарт-контрактів для авторизації у розподілених системах скорочує ризик привілейованого доступу адміністраторів та забезпечує tamper-evident журналювання [5]. Водночас залишаються невирішеними питання затримок on-chain транзакцій та масштабованості для high-throughput середовищ.

На ринку існує кілька рішень, що частково перетинаються з досліджуваною проблематикою. OPA (Open Policy Agent) з Gatekeeper забезпечує декларативне управління політиками через мову Rego, проте залишається централізованим і не надає криптографічних гарантій незмінності рішень. Kyverno пропонує зручний YAML-синтаксис для політик, але також не вирішує проблему довіри до центрального сховища [6]. Системи на основі SPIFFE/SPIRE забезпечують сильну ідентифікацію робочих навантажень через X.509 SVID сертифікати, однак управління правами залишається поза їхньою областю відповідальності. Комерційні рішення, такі як HashiCorp Vault із динамічними секретами, вирішують суміжні задачі, але не інтегруються з блокчейн-інфраструктурою і не забезпечують децентралізованого аудиту [7].

Таким чином, аналіз стану проблеми виявляє чіткий дефіцит: відсутнє комплексне рішення, яке поєднує нативну інтеграцію з Kubernetes Admission Control, децентралізоване зберігання та виконання політик на смарт-контрактах, криптографічно верифікований аудиторський слід та прийнятну продуктивність для виробничих середовищ.

1.2 Аналіз аналогічних програм, баз даних, систем, обґрунтування вибраного напряму та вибір методів рішення

Контроль доступу в Kubernetes є критично важливою складовою безпеки хмарних інфраструктур. На сьогодні існує кілька усталених підходів і систем, що реалізують механізми авторизації та автентифікації в оркестраційних середовищах. Перш ніж обрати архітектурне рішення на основі смарт-контрактів, необхідно проаналізувати наявні аналоги та визначити їх переваги й обмеження.

Вбудована система RBAC (Role-Based Access Control) Kubernetes є найпоширенішим рішенням і надає декларативний механізм призначення дозволів через ресурси ClusterRole, Role, ClusterRoleBinding та RoleBinding (рис. 1.2). Адміністратор описує права доступу у YAML-маніфестах, які зберігаються у розподіленому сховищі etcd. Основний недолік полягає у централізованому збереженні політик безпеки – etcd є єдиною точкою відмови, не забезпечується незмінність журналу подій, а крос-організаційне управління потребує складних федеративних конфігурацій [8].

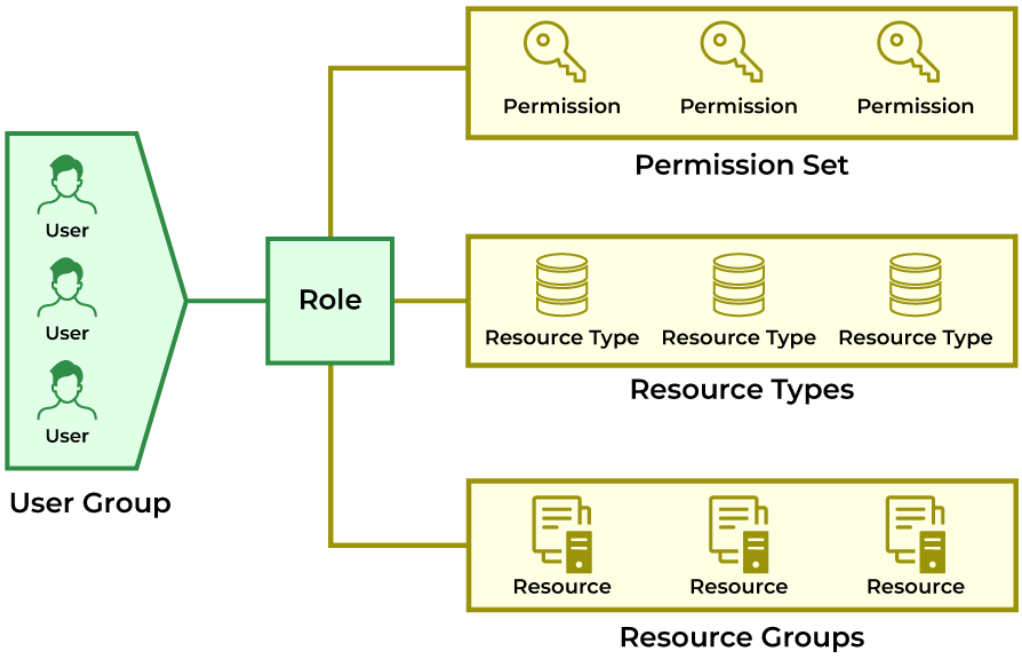

Рисунок 1.2 – Система RBAC (Role-Based Access Control) Kubernetes [9]

OPA/Gatekeeper (Open Policy Agent) – рішення CNCF, яке розширює RBAC через Admission Webhooks і дозволяє визначати складні політики мовою Rego. Система забезпечує гнучкість і підтримує мультиорганізаційні сценарії, проте зберігає правила централізовано, залежить від доступності webhook-сервера, а аудит-журнал не є криптографічно верифікованим. На рисунку 1.3 відображено механізм політик OPA/Gatekeeper.

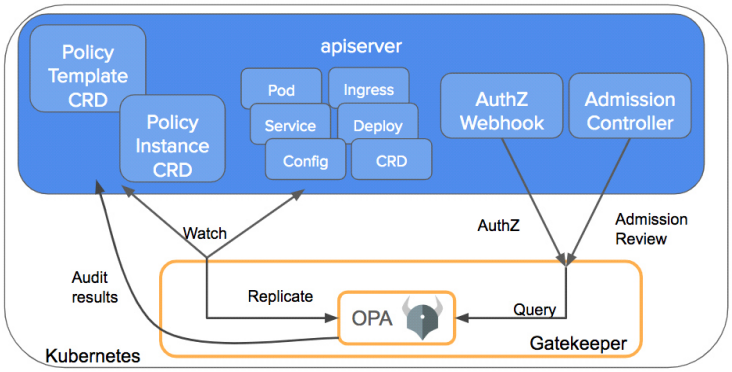

Рисунок 1.3 – Механізм політик OPA/Gatekeeper [10]

Kyverno – декларативний рушій політик, специфічний для Kubernetes, що підтримує генерацію, мутацію та валідацію ресурсів. Незважаючи на зручний YAML-інтерфейс, рішення зберігає стан централізовано й не вирішує проблему довіри у середовищах з кількома незалежними організаціями. На рисунку 1.4 проілюстровано типову діаграму Kyverno з усіма доступними контролерами.

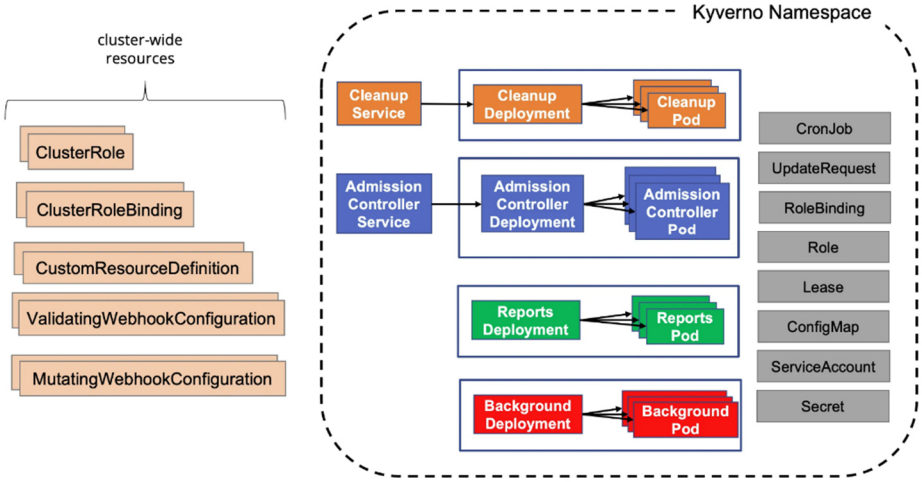

Рисунок 1.4 – Діаграма Kyverno з усіма доступними контролерами [11]

HashiCorp Vault надає централізоване управління секретами і може частково слугувати як PKI-інфраструктура для Kubernetes. Проте Vault є переважно сховищем секретів, а не повноцінним рушієм контролю доступу, і також схильний до централізованих ризиків [12].

Для систематизації результатів огляду в таблиці 1.1 наведено порівняльний аналіз ключових характеристик запропонованих рішень.

Таблиця 1.1 – Порівняльний аналіз систем контролю доступу

Система	Децентра-лізація	Незмінний аудит	Мульти-орг.	Крипто-підпис	Автомат.	Складність розгортання
Kubernetes RBAC	Ні	Ні	Обмежена	Ні	Часткова	Низька
OPA/Gatekeeper	Ні	Ні	Так	Ні	Висока	Середня
Kyverno	Ні	Ні	Обмежена	Ні	Висока	Середня
HashiCorp Vault	Часткова	Ні	Так	Ні	Висока	Висока
Пропонована система	Так	Так	Так	Так	Висока	Середня

У науковій літературі та прикладних розробках останніх років виокремлюється напрям застосування блокчейн-технологій для управління ідентичністю та доступом (Blockchain-based Identity and Access Management, BIAMd). Проекти на основі Hyperledger Fabric та Ethereum демонструють принципову можливість зберігання ACL-записів у розподіленому реєстрі з незмінною хронологією змін і верифікованим журналом операцій [13].

Hyperledger Fabric пропонує дозволений (permissioned) блокчейн із підтримкою чейнкоду (смарт-контрактів) і орієнтований на корпоративне середовище з відомим колом учасників. Проте значна складність розгортання мережі валідаторів і відсутність нативної інтеграції з Kubernetes API Server роблять цей варіант надмірним для цільового сценарію використання.

Ethereum (зокрема тестові мережі Sepolia та локальна мережа Hardhat Network) надає зрілу екосистему смарт-контрактів мовою Solidity, широку

підтримку бібліотек (OpenZeppelin, ethers.js, web3.py) та прозорий механізм детермінованого виконання. Легковагість розгортання локальної мережі через Hardhat дозволяє повністю відтворити середовище без залучення публічної інфраструктури, що є вагомою перевагою для розробки та тестування.

На основі проведеного аналізу аналогів встановлено такі ключові недоліки існуючих рішень: відсутність незмінного та криптографічно верифікованого журналу подій; централізоване зберігання політик з єдиною точкою відмови; неможливість прозорого аудиту у мультиорганізаційних сценаріях; ручне адміністрування без автоматизованого виконання правил доступу.

Запропонована система усуває зазначені недоліки шляхом інтеграції смарт-контрактів Ethereum з Kubernetes Admission Webhook. Смарт-контракти виконують роль децентралізованого реєстру політик доступу: всі зміни фіксуються у блокчейні, є публічно верифікованими та незмінними. Kubernetes-оператор, написаний на Go, взаємодіє з блокчейном через JSON-RPC і транслює блокчейн-стан у нативні RBAC-ресурси кластера в режимі реального часу.

Такий підхід забезпечує чотири ключові властивості: по-перше, децентралізацію управління доступом без єдиної точки відмови; по-друге, незмінний аудит-журнал на рівні блоків з криптографічним підтвердженням; по-третє, можливість крос-організаційного управління без взаємної довіри між адміністраторами; по-четверте, автоматичне виконання політик через детермінований механізм смарт-контрактів.

Архітектура системи базується на трьох методологічних рівнях. На рівні зберігання та виконання політик обрано смарт-контракти Solidity, розгорнуті в мережі Ethereum (локально – Hardhat Network, для production – Ethereum Mainnet або сумісний EVM-ланцюг). Контракти реалізують патерн AccessControl з бібліотеки OpenZeppelin, що забезпечує стандартизоване управління ролями та ієрархічну делегацію прав із перевіреною в аудитах кодовою базою.

На рівні інтеграції з Kubernetes застосовано патерн Kubernetes Operator з реалізацією ValidatingWebhookConfiguration. Оператор написаний на Go з використанням controller-runtime та client-go, підписується на події блокчейну

через ethclient і синхронізує стан RBAC-ресурсів кластера у відповідь на емітовані смарт-контрактом події (events). Така архітектура забезпечує слабку зв'язаність компонентів і можливість горизонтального масштабування.

На рівні безпеки комунікації між компонентами застосовано взаємну TLS-автентифікацію (mTLS) та підпис транзакцій приватним ключем вузла-оператора. Для верифікації ідентичності воркерів Kubernetes використано механізм ServiceAccount Token із прив'язкою до адреси гаманця Ethereum через процедуру on-chain реєстрації. На рисунку 1.5 продемонстровано процес рукостискання TLS.

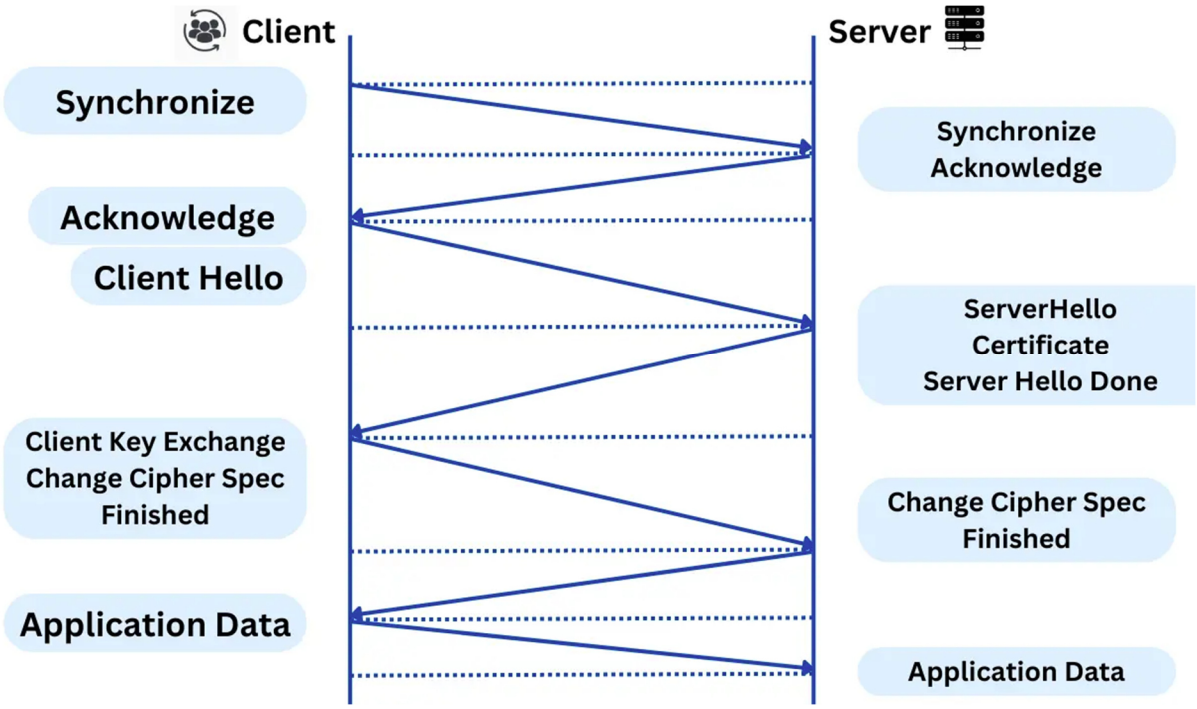

Рисунок 1.5 – Процес рукостискання TLS [14]

Методологічно проект спирається на принципи Zero Trust Architecture: кожен запит до API-сервера Kubernetes верифікується незалежно від мережевого положення ініціатора, привілеї видаються мінімально необхідні (Principle of Least Privilege), а журнал подій є верифікованим і незмінним. Поєднання блокчейн-технологій з нативними механізмами Kubernetes дозволяє досягти оптимального балансу між безпекою, децентралізацією та операційною сумісністю з наявними хмарними інфраструктурами.

1.3 Аналіз і вибір засобів проектування

Для реалізації системи контролю доступу на основі смарт-контрактів необхідно визначити базову платформу оркестрування контейнерів, яка забезпечить середовище виконання мікросервісів. Платформа повинна підтримувати розширювану модель авторизації, гнучке управління мережевими політиками та інтеграцію із зовнішніми системами автентифікації.

Kubernetes є де-факто стандартом оркестрування контейнерів у виробничих середовищах. Вбудована модель RBAC (Role-Based Access Control) забезпечує базовий рівень авторизації, однак вона є статичною і не підтримує динамічне управління дозволами на основі блокчейн-записів. Саме цей недолік і обумовлює необхідність розробки розширення на основі смарт-контрактів. В таблиці 1.2 продемонстровано порівняльний аналіз платформ оркестрування.

Таблиця 1.2 – Порівняльний аналіз платформ оркестрування

Платформа	RBAC	Webhook Auth	CRD	Оцінка
Kubernetes	✓	✓	✓	9/10
Docker Swarm	Обмежений	✗	✗	5/10
Nomad (HashiCorp)	✓	Частково	✗	6/10
OpenShift (Red Hat)	✓	✓	✓	8/10

Архітектура Kubernetes дозволяє інтегрувати зовнішні вебхуки авторизації через механізм Webhook Authorization, що робить її ідеальною платформою для впровадження кастомного контролера доступу. Також важливою перевагою є підтримка Custom Resource Definitions (CRD), що дозволяє декларативно описувати об’єкти смарт-контрактів у форматі, сумісному з Kubernetes API.

На підставі порівняльного аналізу для реалізації проєкту обрано платформу Kubernetes версії 1.28+. Ця версія підтримує структурові схеми CRD, покращений механізм вебхуків авторизації та стабільний API для Gateway API, що є критичним для побудови проксі-шару між блокчейн-вузлом і кластером.

Смарт-контракти є центральним компонентом системи, оскільки саме вони зберігають реєстр дозволів і виконують логіку авторизації в децентралізованому середовищі. До блокчейн-платформи висуваються такі вимоги: підтримка смарт-контрактів з мовою програмування загального призначення, детермінованість виконання, низька затримка підтвердження транзакцій і можливість розгортання у приватному мережевому режимі (permissioned blockchain).

Ethereum із приватною мережею на базі Besu або Ganache (для розробки) і фреймворком Hardhat є оптимальним вибором. Він поєднує зрілу екосистему Solidity, сумісність з OpenZeppelin (бібліотека перевірених смарт-контрактів) і можливість локального розгортання. Для виробничих середовищ рекомендується використання приватної мережі Ethereum (PoA консенсус), що забезпечує підтвердження транзакцій протягом 1-2 секунд.

Ethereum є найбільш зрілою платформою для смарт-контрактів з мовою програмування Solidity. Проте у контексті Kubernetes-інтеграції публічна мережа Ethereum має недоліки: висока вартість газу, затримки підтвердження до 15 секунд та публічна видимість транзакцій. Ці обмеження роблять публічний Ethereum неприйнятним для систем контролю доступу реального часу [15].

Hyperledger Fabric пропонує модель приватної блокчейн-мережі з підтримкою смарт-контрактів на Go, Java та JavaScript (chaincode). Fabric забезпечує суттєво нижчі затримки (100-200 мс) і конфіденційність даних у каналах. Однак складність інфраструктури (ordering service, MSP, peer-вузли) ускладнює інтеграцію з невеликими Kubernetes-кластерами. В таблиці 1.3 відображено порівняльний аналіз блокчейн-платформ.

Таблиця 1.3 – Порівняльний аналіз блокчейн-платформ

Платформа	Мова SC	Затримка	Приватна мережа	Оцінка
Ethereum (приватна)	Solidity	1-2 с	✓	9/10
Hyperledger Fabric	Go/Java/JS	0.1-0.2 с	✓	7/10
Ethereum (публічна)	Solidity	12-15 с	✗	4/10
Polkadot (Substrate)	Rust/ink!	6 с	✓	6/10

Для розробки та тестування смарт-контрактів обрано фреймворк Hardhat, оскільки він підтримує TypeScript, вбудований локальний вузол, покриття коду та інтеграцію з бібліотекою ethers.js. OpenZeppelin Contracts використовується як базова бібліотека для реалізації контролю доступу на основі ролей (AccessControl) та управління власністю (Ownable).

Контролер доступу є мостом між API-сервером Kubernetes і блокчейн-вузлом. Він реалізується у вигляді вебхука авторизації (Authorization Webhook) і повинен задовольняти суворі вимоги щодо продуктивності: час відповіді не повинен перевищувати 100 мс, оскільки він блокує виконання кожного запиту до Kubernetes API [16].

Мова програмування Go (Golang) є стандартом для розробки Kubernetes-розширень. Офіційний Kubernetes SDK (client-go), оператор-фреймворк (controller-runtime, kubebuilder) і більшість плагінів системи написані на Go. Golang забезпечує мінімальний час запуску, низьке споживання пам’яті і вбудовану підтримку паралелізму через горутини.

Для взаємодії з Ethereum-вузлом у середовищі Go використовується бібліотека go-ethereum (geth), яка надає ABI-кодування/декодування смарт-контрактів, підписання транзакцій і JSON-RPC клієнт. Разом з kubebuilder для генерації CRD і controller-runtime для управління циклом Reconcile цей стек утворює повноцінне технологічне рішення.

Для контейнеризації компонентів системи обрано Docker із багатоетапними збірками (multi-stage builds), що дозволяє мінімізувати розмір образів. Базовий образ distroless/static забезпечує зменшення поверхні атаки, що критично важливо для компонентів безпеки. Helm використовується для пакування і розгортання Kubernetes-маніфестів, а також для параметризації конфігурацій між середовищами (dev, staging, production).

Для автоматизації CI/CD обрано GitHub Actions з інтеграцією з інструментом kind (Kubernetes in Docker) для виконання інтеграційних тестів у середовищі, що максимально наближене до виробничого. Це дозволяє тестувати взаємодію між контролером доступу, Kubernetes API і Ethereum-вузлом у рамках

одного pipeline. На рисунку 1.6 відображено процес CI/CD з використанням GitHub Actions.

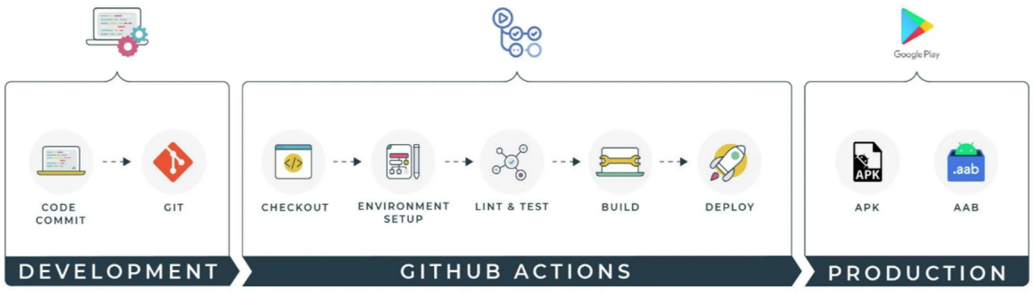

Рисунок 1.6 – Процес CI/CD з використанням GitHub Actions [17]

На підставі проведеного аналізу сформовано технологічний стек проєкту: Kubernetes 1.28+ як платформа оркестрування, Ethereum (приватна мережа, PoA) із смарт-контрактами на Solidity як децентралізований реєстр дозволів, Go із фреймворками kubebuilder і controller-runtime для реалізації контролера доступу, Hardhat і OpenZeppelin для розробки і тестування смарт-контрактів. Такий вибір забезпечує баланс між продуктивністю, зрілістю екосистеми та безпекою, що відповідає вимогам до систем контролю доступу корпоративного рівня.

1.4 Постановка завдання на кваліфікаційну роботу бакалавра

Спираючись на результати аналізу предметної області, огляду існуючих аналогів систем контролю доступу в контейнерних середовищах та обраного технологічного стеку, метою кваліфікаційної роботи бакалавра є проектування та програмна реалізація системи контролю доступу в Kubernetes на основі смарт-контрактів, що забезпечує децентралізований, прозорий та криптографічно захищений механізм авторизації ресурсів кластера без залежності від централізованого сховища політик. Актуальність обраної теми зумовлена стрімким поширенням контейнерних технологій у промисловій розробці програмного забезпечення та зростаючими вимогами до безпеки інфраструктури, адже компрометація централізованого сервера авторизації у

традиційних підходах здатна надати зловмиснику необмежений доступ до всіх ресурсів кластера.

Для досягнення поставленої мети необхідно розв’язати такі основні завдання. Насамперед – провести ґрунтовний аналіз існуючих механізмів контролю доступу в Kubernetes, зокрема RBAC, ABAC та Admission Webhook, визначити їхні переваги, архітектурні обмеження та потенційні вразливості з метою формування чітких вимог до розроблюваної системи. Паралельно необхідно дослідити технологію смарт-контрактів на платформі Ethereum, принципи роботи віртуальної машини EVM, особливості мови Solidity, а також існуючі підходи до управління ідентичністю та авторизацією на блокчейні, включаючи стандарти ERC та патерни контролю доступу на кшталт OpenZeppelin AccessControl.

Наступним кроком є проектування архітектури системи з урахуванням принципів модульності, масштабованості та чіткого розмежування відповідальності між компонентами. На цьому етапі розробляється детальна модель взаємодії Kubernetes Admission Webhook із смарт-контрактом, визначаються межі довіри між компонентами, описується схема обміну даними та стратегія обробки відмов у разі тимчасової недоступності блокчейн-вузла. Архітектурні рішення фіксуються у вигляді UML-діаграм компонентів, послідовностей та розгортання, що забезпечує відтворюваність і зрозумілість рішення для подальших дослідників.

Після завершення проектування виконується реалізація смарт-контракту мовою Solidity для зберігання та перевірки політик доступу – ролей, ресурсів і дозволів – з урахуванням вимог безпеки: захист від реентерабельності, обмеження прав адміністратора, коректна емісія подій для аудиту. Паралельно розробляється Webhook-сервіс, який перехоплює запити до Kubernetes API Server, звертається до смарт-контракту через бібліотеку взаємодії з блокчейном та повертає рішення Allow або Deny у форматі AdmissionReview. Сервіс реалізується із підтримкою TLS відповідно до вимог Kubernetes до захищених

Webhook-ендпоінтів, а також із механізмом кешування відповідей для мінімізації затримок.

Окремим завданням є розробка адміністративного інтерфейсу або CLI-утиліти, що дозволяє уповноваженим особам додавати, змінювати та відкликати права доступу суб’єктів через взаємодію зі смарт-контрактом, не вдаючись до прямого редагування конфігураційних файлів кластера. Такий підхід забезпечує незмінність журналу змін і унеможливлює непомічену модифікацію політик, оскільки кожна транзакція фіксується у блокчейні з підписом ініціатора та міткою часу. Завершальним етапом практичної частини є всебічне тестування системи: функціональне – для перевірки коректності рішень авторизації, інтеграційне – для підтвердження стабільності взаємодії компонентів, навантажувальне – для оцінки продуктивності в умовах інтенсивного трафіку запитів до Webhook, а також базовий аудит безпеки смарт-контракту з використанням інструментів статичного аналізу.

Очікуваним результатом виконання кваліфікаційної роботи є функціонально завершена система контролю доступу, реалізована на основі інтеграції Kubernetes Admission Webhook та смарт-контракту Ethereum, готова до розгортання в тестовому середовищі та подальшого промислового впровадження.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Обґрунтування вибору шляхів, технологій (алгоритмів) і засобів вирішення поставленого завдання

Для реалізації системи контролю доступу в Kubernetes на основі смарт-контрактів обрано децентралізовану архітектуру з використанням блокчейн-мережі. Такий підхід дозволяє усунути єдину точку відмови, притаманну традиційним централізованим системам авторизації (RBAC, ABAC), і забезпечити незмінність та прозорість політик доступу. Децентралізована модель гарантує, що жоден окремий вузол не може самостійно змінити або скасувати правила авторизації без досягнення консенсусу у мережі.

Порівняно з альтернативними підходами – такими як використання зовнішніх IAM-провайдерів (AWS IAM, Azure AD) або Policy Engines (Open Policy Agent) – запропонований підхід на базі смарт-контрактів забезпечує більш високий рівень аудиту та стійкості до маніпуляцій. Усі зміни правил доступу фіксуються в незмінному ланцюжку блоків, що спрощує проведення криміналістичного аналізу інцидентів безпеки.

Для реалізації смарт-контрактів обрано платформу Ethereum (або сумісну з EVM корпоративну мережу, наприклад Hyperledger Besu). Вибір обґрунтовується такими факторами:

– зрілість екосистеми та широка підтримка інструментів розробки (Hardhat, Truffle, Foundry);

– підтримка мови Solidity, яка надає потужні механізми для реалізації логіки контролю доступу (модифікатори, події, маппінги);

– наявність стандартизованих патернів управління доступом (OpenZeppelin AccessControl, Ownable);

– можливість розгортання у приватній мережі для забезпечення конфіденційності корпоративних даних і підвищення пропускної здатності.

Hyperledger Fabric як альтернатива розглядалася, проте була відхилена через складнішу інфраструктуру розгортання та обмежену сумісність з наявними EVM-інструментами моніторингу.

Інтеграцію смарт-контрактів із Kubernetes реалізовано через механізм Admission Controllers, а саме Validating Admission Webhook. Цей підхід обрано тому, що він дозволяє перехоплювати будь-які API-запити до Kubernetes до їх запису в etcd, не вносячи змін до ядра платформи. Webhook-сервер звертається до блокчейн-вузла для перевірки поточних політик доступу, записаних у смарт-контракті, і повертає дозвіл або відмову.

Альтернативний варіант з використанням Mutating Admission Webhook або Custom Resource Definitions (CRD) з окремим контролером було відхилено: перший не надає механізму блокування запитів, а другий вимагає значно більшого обсягу коду та підтримки власного циклу узгодження (reconciliation loop).

Для реалізації компонентів системи обрано такий технологічний стек:

– Go (Golang) – для розробки Webhook-сервера. Мова є стандартною для екосистеми Kubernetes, забезпечує високу продуктивність та надає офіційний клієнт kubernetes/client-go;

– Solidity – для написання смарт-контрактів. Мова спеціалізована для EVM, має розвинену документацію та бібліотеки безпечних патернів;

– ethclient (Go Ethereum) – для взаємодії Webhook-сервера з блокчейн-вузлом через JSON-RPC/WebSocket;

– Helm – для пакування та розгортання всіх компонентів системи у Kubernetes-кластері у відтворюваний спосіб;

– Hardhat – для компіляції, тестування та деплою смарт-контрактів у середовищах розробки та тестування.

Алгоритм контролю доступу базується на атрибутній моделі (ABAC) із збереженням політик у блокчейні. При надходженні запиту до Kubernetes API-сервера Webhook-сервер формує запит до смарт-контракту, передаючи атрибути суб’єкта (ServiceAccount, простір імен), об’єкта (тип ресурсу, ім’я) та операції

(дієслово: get, create, delete тощо). Смарт-контракт виконує перевірку збережених правил і повертає булеве значення: дозволено чи заборонено.

Для мінімізації затримок, що виникають унаслідок мережевого звернення до блокчейн-вузла, передбачено кешування результатів на рівні Webhook-сервера з TTL, що налаштовується. Це дозволяє досягти прийнятного часу відповіді (менше 50 мс для кешованих запитів) без компромісу щодо актуальності політик доступу.

Таким чином, обраний комплекс технологій та архітектурних рішень забезпечує баланс між безпекою, прозорістю, продуктивністю та сумісністю з існуючою інфраструктурою Kubernetes, що повністю відповідає поставленим вимогам проекту.

2.2 Функціонально-структурна схема роботи об’єкта проектування

Функціонально-структурна схема відображає взаємодію між основними підсистемами: Kubernetes Control Plane, шлюзом контролю доступу (Access Gateway), рівнем смарт-контрактів та зовнішніми клієнтами. Кожен компонент виконує чітко визначену роль у процесі авторизації.

Kubernetes API Server приймає запити від клієнтів (розробників, CI/CD-пайплайнів) через HTTPS або gRPC. Запит потрапляє до Admission Webhook, який проксіює його до Access Gateway Service. Сервіс формує виклик смарт-контракту через Web3-провайдера та отримує рішення ALLOW або DENY. Результат рішення фіксується в незмінному журналі аудиту на блокчейні.

Рівень смарт-контрактів є центральним елементом архітектури і забезпечує три ключові властивості системи:

– децентралізованість – жоден окремий вузол не може змінити або скасувати рішення;

– незмінність аудит-логу – всі рішення записуються в блокчейн і не можуть бути видалені;

– прозорість – будь-який авторизований суб’єкт може перевірити поточний стан політик.

Алгоритм обробки запиту на доступ описує повний життєвий цикл запиту від моменту надходження до Kubernetes API Server до запису результату в блокчейн-журнал. На рисунку 2.1 проілюстровано функціонально-структурна схема системи контролю доступу.

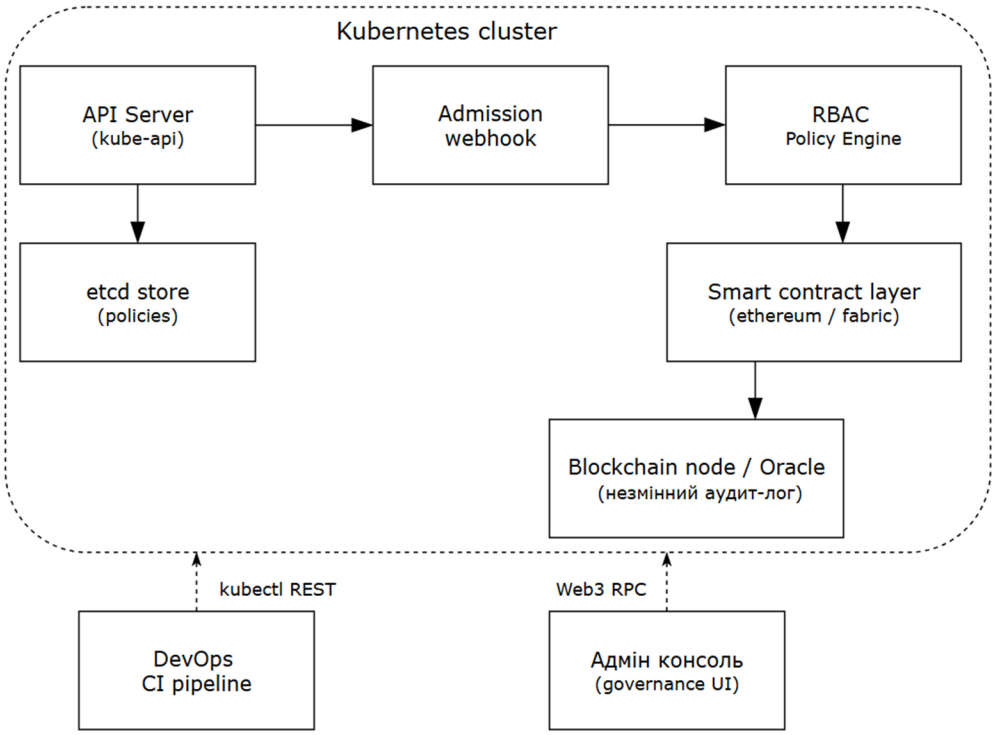

Рисунок 2.1 – Функціонально-структурна схема контролю доступу

На першому етапі виконується автентифікація за допомогою JWT-токена або взаємного TLS (mTLS). Якщо автентифікація не пройдена, клієнт отримує відповідь 401 Unauthorized без подальшої обробки. Після успішної автентифікації Admission Webhook формує структуру AccessRequest, що містить суб’єкт (subject), простір імен (namespace), ресурс (resource) та дію (verb: get, create, update, delete тощо).

На другому етапі Access Gateway викликає функцію verifyAccess смарт-контракту. Контракт перевіряє наявність суб’єкта у реєстрі ролей, завантажує відповідну Policy та порівнює запитувані права з дозволеними. Незалежно від

результату рішення записується в on-chain аудит-лог. На рисунку 2.2 наведено блок-схема алгоритму обробки запиту на доступ до ресурсів Kubernetes. Лістинг коду Webhook-сервісу на Go відображено в додатку А.

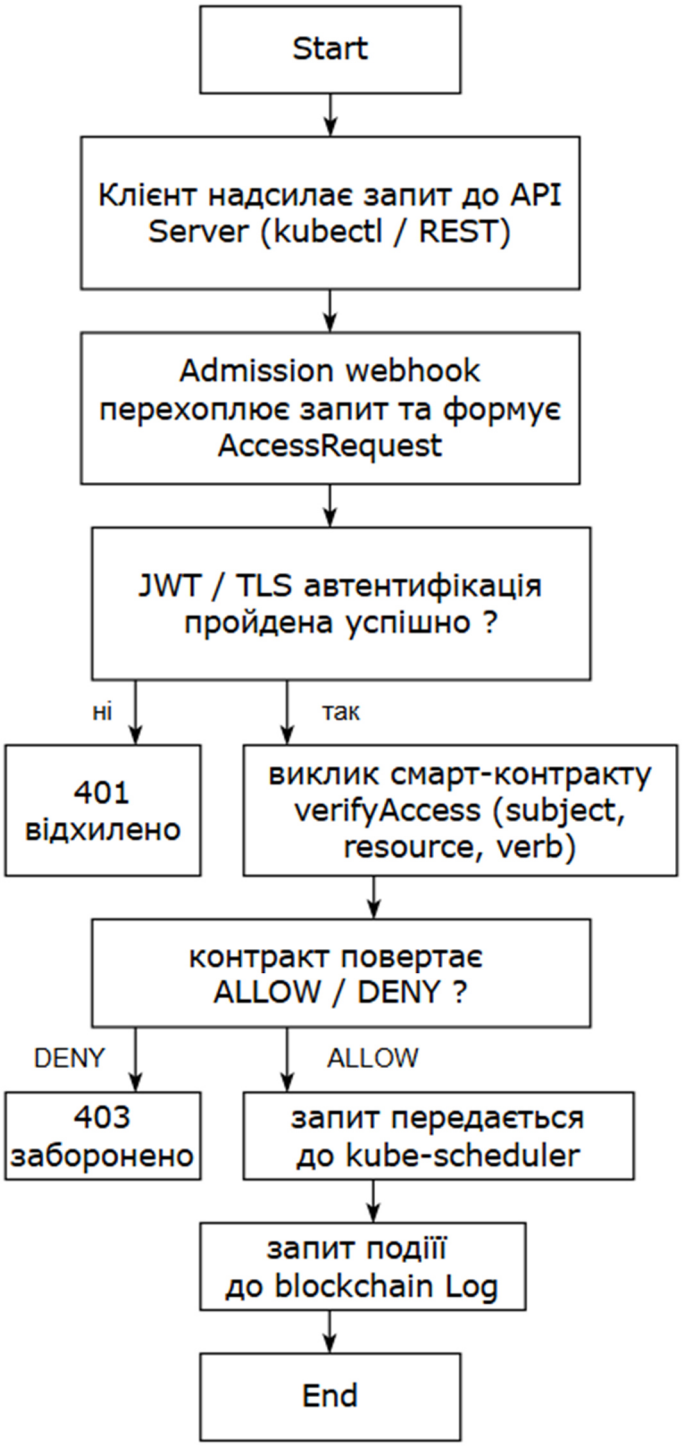

Рисунок 2.2 – Блок-схема алгоритму обробки запиту на доступ до ресурсів Kubernetes

Критичним є те, що запис до аудит-логу відбувається у будь-якому разі – як при дозволі, так і при забороні доступу. Це забезпечує повну трасування всіх спроб доступу та є обов'язковою вимогою для відповідності стандартам безпеки SOC 2 та ISO/IEC 27001.

Алгоритм виконання смарт-контракту деталізує внутрішню логіку функції verifyAccess, що виконується в середовищі EVM (Ethereum Virtual Machine) або Hyperledger Fabric Chaincode.

Контракт отримує вхідні параметри: subject (адреса суб'єкта), namespace, resource, verb та необов'язковий масив labels для атрибутивного контролю доступу (ABAC). Першим кроком перевіряється наявність суб'єкта у whitelist – списку привілейованих адміністративних адрес, яким завжди дозволений повний доступ. Якщо суб'єкт не входить до whitelist, контракт звертається до PolicyRegistry та завантажує актуальну Policy.

Policy являє собою структуру даних у сховищі контракту (mapping), що містить масив PolicyRule. Кожне правило специфікує дозволені verbs для певних resources у певних namespaces. Якщо жодне правило не збігається, контракт випромінює подію Denied та повертає false. Важливою властивістю алгоритму є атомарність: запис до AuditLog та повернення результату відбуваються в одній транзакції блокчейну. Це виключає можливість розбіжності між фактично прийнятим рішенням та його записом в журналі.

UML-діаграма класів описує об'єктну модель системи на рівні програмного коду. В основі архітектури знаходиться інтерфейс IAccessController, який визначає контракт взаємодії для всіх реалізацій контролера доступу.

Клас SmartContractController реалізує IAccessController та інкапсулює всю логіку взаємодії з блокчейном: адресу розгорнутого контракту, ABI-специфікацію для кодування/декодування викликів та Web3Provider для підписання транзакцій. Альтернативна реалізація KubernetesRBACFallback використовується у випадках деградованого режиму роботи, коли блокчейн-вузол недоступний, і делегує перевірку стандартному механізму RBAC

Kubernetes. На рисунку 2.3 проілюстровано блок-схема алгоритму виконання смарт-контракту verifyAccess. Лістинг функції verifyAccess смарт-контракту наведено в додатку Б.

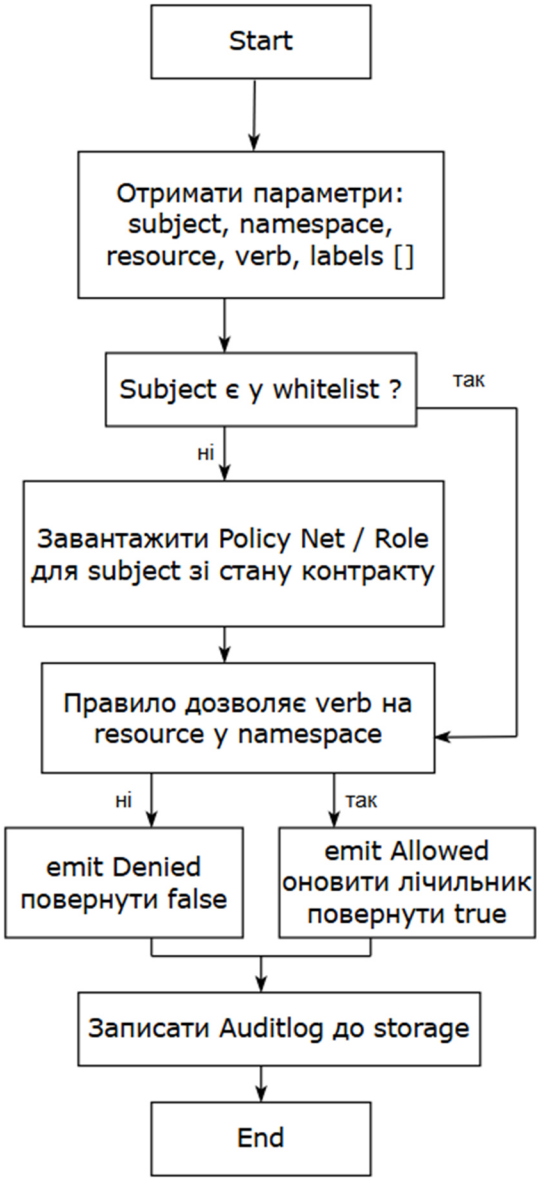

Рисунок. 2.3 – Блок-схема алгоритму виконання смарт-контракту verifyAccess

PolicyRegistry відповідає за управління реєстром політик. Клас агрегує об’єкти Policy, кожен з яких містить унікальний ідентифікатор (bytes32), адресу суб’єкта, масив правил PolicyRule, термін дії (expiry) та адресу видавця (issuer). Клас AuditLog є незмінною структурою, що записується в блокчейн і містить хеш транзакції, ідентифікатор суб’єкта, прийняте рішення та номер блоку.

Принцип розподілу відповідальності (Single Responsibility Principle) реалізовано шляхом виділення PolicyRegistry як окремого компонента, що не бере участі безпосередньо в авторизаційних рішеннях, а лише забезпечує CRUD-операції над сховищем політик. На рисунку 2.4 наведено UML-діаграма класів системи контролю доступу.

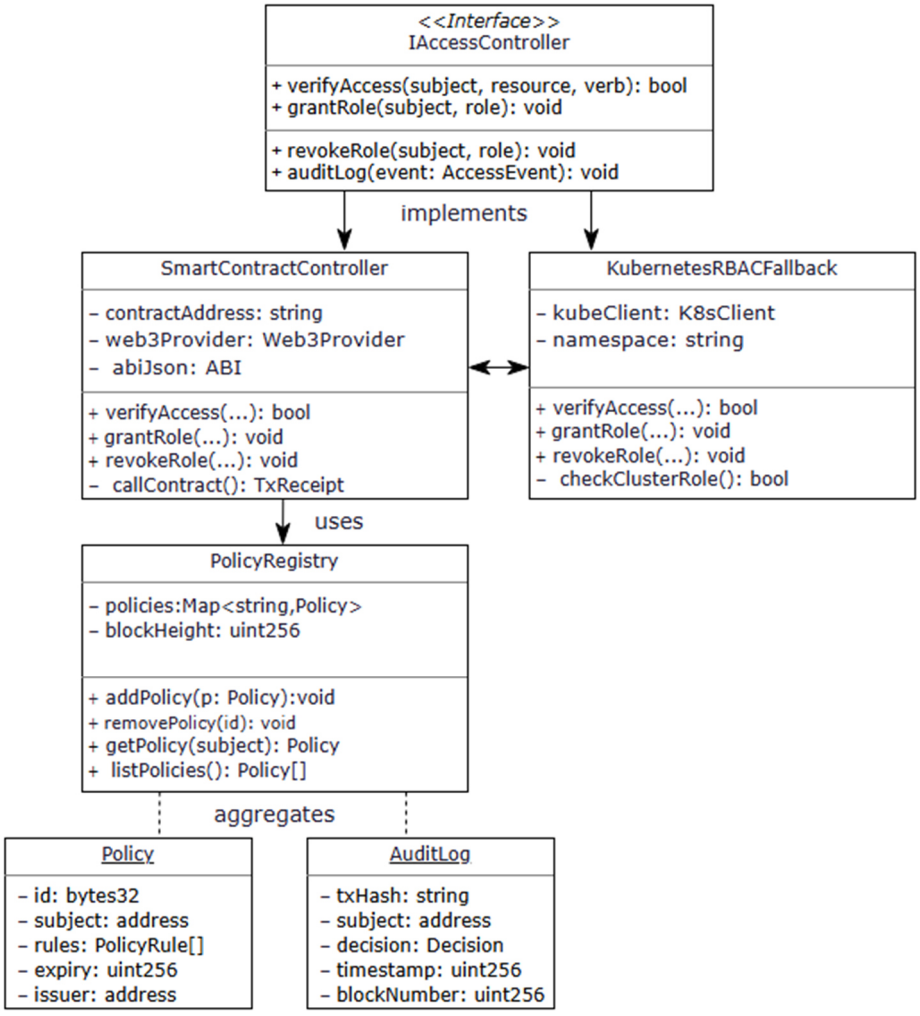

Рисунок 2.4 – UML-діаграма класів системи контролю доступу

Діаграма компонентів відображає фізичну декомпозицію системи на розгортувані артефакти та їх залежності. Система складається з трьох рівнів: зовнішніх клієнтів, Kubernetes Control Plane та Blockchain Layer.

На рівні Kubernetes Control Plane розміщуються такі компоненти: kube-apiserver (стандартний компонент Kubernetes), ValidatingWebhookConfiguration

(конфігурація вебхуку у форматі YAML-маніфесту), та AuthZ Plugin (бінарний плагін авторизації, що реалізує інтерфейс Kubernetes Authorizer). Плагін взаємодіє з Access Gateway Service через внутрішній gRPC-виклик з mTLS-шифруванням.

Access Gateway Service містить три підкомпоненти: RequestParser (перетворює Kubernetes SubjectAccessReview у параметри виклику контракту), PolicyEvaluator (виконує виклик смарт-контракту та інтерпретує відповідь) та AuditEmitter (асинхронно надсилає додаткові метадані до зовнішніх SIEM-систем).

На рівні Blockchain Layer розгорнуті три смарт-контракти: AccessControl.sol (основна логіка авторизації), PolicyRegistry.sol (сховище політик) та AuditLog.sol (незмінний журнал). Всі три контракти розгортаються в одній мережі та взаємодіють між собою через міжконтрактні виклики (internal contract calls) з ідемпотентними функціями запису.

Розроблена функціонально-структурна схема демонструє, що система контролю доступу в Kubernetes на основі смарт-контрактів є повноцінною багаторівневою архітектурою з чіткою межею між компонентами. Блок-схеми алгоритмів підтверджують детерміновану поведінку системи: кожен запит на доступ проходить фіксований шлях обробки, результат якого завжди фіксується в блокчейні.

UML-діаграма класів розкриває застосування принципів об’єктно-орієнтованого проектування, зокрема програмування на основі інтерфейсів, що забезпечує можливість заміни блокчейн-бекенду без зміни решти системи. Діаграма компонентів визначає межі розгортування та залежності між артефактами, що є необхідною основою для подальшого опису процесу CI/CD та DevSecOps-автоматизації.

Описана архітектура відповідає принципам Zero Trust Security: кожен запит перевіряється незалежно від попередньої історії, а всі авторизаційні рішення є публічно верифікованими через блокчейн-транзакції.

2.3 Створення моделі бази даних

Система контролю доступу в Kubernetes на основі Smart Contracts оперує двома принципово різними рівнями зберігання даних. Перший рівень – децентралізований реєстр блокчейну Ethereum, у якому зберігаються незмінні записи про дозволи доступу, аудит подій та визначення ролей. Другий рівень – реляційна база даних PostgreSQL, що виконує роль операційного кешу: зберігає конфігурації кластерів Kubernetes, кешовані рішення авторизації та журнали запитів для швидкого пошуку.

Такий гібридний підхід забезпечує баланс між незмінністю та прозорістю блокчейну і продуктивністю реляційної СУБД при обслуговуванні тисяч запитів на авторизацію щосекунди в умовах реального Kubernetes-кластера.

Центральна сутність системи, що відповідає структурі смарт-контракту AccessControlRegistry. Кожен запис визначає один дозвіл на виконання певного дієслова Kubernetes для конкретного суб’єкта на визначеному ресурсі, що відображено в таблиці 2.1.

Таблиця 2.1 – AccessPolicy

Атрибут	Тип	Обмеження	Опис
`policy_id`	BYTES32	PK, NOT NULL	Унікальний хеш-ідентифікатор запису
`subject_address`	VARCHAR(42)	NOT NULL, INDEX	Ethereum-адреса суб’єкта доступу
`resource_kind`	VARCHAR(64)	NOT NULL	Тип ресурсу K8s (Pod, Deployment…)
`resource_namespace`	VARCHAR(253)	NOT NULL	Простір імен Kubernetes
`verb`	ENUM	NOT NULL	get, list, create, update, delete
`is_active`	BOOLEAN	DEFAULT TRUE	Статус активності дозволу
`block_number`	BIGINT	NOT NULL	Блок фіксації в блокчейні
`tx_hash`	BYTES32	UNIQUE, NOT NULL	Хеш транзакції Ethereum
`created_at`	TIMESTAMP	NOT NULL	Час створення запису

Зберігає визначення ролей, що агрегують набори дозволів аналогічно до ClusterRole у Kubernetes RBAC, проте з прив'язкою до адреси смарт-контракту в мережі Ethereum, що продемонстровано в таблиці 2.2.

Таблиця 2.2 – RoleDefinition

Атрибут	Тип	Обмеження	Опис
role_id	BYTES32	PK, NOT NULL	Унікальний ідентифікатор ролі
role_name	VARCHAR(128)	UNIQUE, NOT NULL	Людиночитана назва ролі
owner_address	VARCHAR(42)	NOT NULL, INDEX	Адреса власника/адміністратора
contract_address	VARCHAR(42)	NOT NULL	Адреса смарт-контракту ролі
metadata_ipfs	VARCHAR(64)	NULLABLE	CID-посилання на IPFS-метадані
created_at	TIMESTAMP	NOT NULL	Час реєстрації ролі

Незмінний журнал усіх подій авторизації, синхронізований із подіями смарт-контракту. Слугує джерелом правди для аудиту відповідності вимогам безпеки (compliance), які наведені в таблиці 2.3.

Таблиця 2.3 – AuditLog

Атрибут	Тип	Обмеження	Опис
log_id	UUID	PK, NOT NULL	Унікальний ідентифікатор запису
policy_id	BYTES32	FK→AccessPolicy	Посилання на застосований дозвіл
requester	VARCHAR(42)	NOT NULL, INDEX	Ethereum-адреса запитувача
decision	ENUM	NOT NULL	ALLOW або DENY
resource_path	TEXT	NOT NULL	Повний шлях до ресурсу K8s
response_time_ms	INTEGER	NOT NULL	Час відповіді (мілісекунди)
event_timestamp	TIMESTAMP	NOT NULL, INDEX	Час події авторизації

Між таблицями визначено такі відношення: AccessPolicy → AuditLog (один-до-багатьох); RoleDefinition → AccessPolicy (один-до-багатьох через зв’язкову таблицю RolePolicyBinding). Для забезпечення продуктивності запитів авторизації застосовано трирівневу індексну стратегію:

– складений індекс (subject_address, resource_kind, resource_namespace, verb) на таблиці AccessPolicy – покриває 0,95 запитів авторизації за одне сканування індексу;

– частковий індекс WHERE is_active = TRUE – скорочує розмір робочого набору вдвічі, обмежуючи вибірку лише активними дозволами;

– індекс (event_timestamp DESC) на AuditLog – забезпечує ефективне хронологічне вибіркове читання журналів для звітів безпеки.

Цілісність між PostgreSQL та Ethereum досягається через сервіс Event Listener, який підписується на події смарт-контракту (PolicyGranted, PolicyRevoked, RoleCreated) та відображає їх у реляційні записи. Поле block_number у кожній сутності гарантує ідемпотентність повторної синхронізації: якщо блок вже оброблено, запис пропускається без дублювання. Таблиця SyncCheckpoint зберігає останній оброблений номер блоку, що забезпечує коректне відновлення після збоїв мережі або рестарту сервісу.

Спроектована модель дозволяє системі масштабуватись до десятків тисяч дозволів без деградації продуктивності завдяки поєднанню незмінності блокчейн-реєстру та оптимізованої реляційної схеми кешування.

РОЗДІЛ 3
РОЗРОБКА СИСТЕМИ КОНТРОЛЮ ДОСТУПУ

3.1 Практична реалізація об’єкта проектування. Побудова блок-схем модулів програми

Практична реалізація системи контролю доступу в Kubernetes на основі смарт-контрактів передбачає використання сучасного стеку технологій, який забезпечує не лише функціональність, але й зручність розробки, тестування та розгортання. У даному проєкті ключову роль відіграють такі інструменти: Hardhat для розробки та тестування смарт-контрактів, Go разом із бібліотекою go-ethereum для взаємодії з блокчейном, Kubebuilder для створення Kubernetes-оператора, а також GitHub Actions для автоматизації процесів CI/CD.

Загальна архітектура системи залишається модульною, проте кожен компонент реалізується із використанням зазначених технологій. Це дозволяє досягти високого рівня інтеграції між блокчейн-рівнем та Kubernetes.

Розробка смарт-контрактів виконується у середовищі Hardhat, що забезпечує локальне тестування, деплой та інтеграцію з різними мережами Ethereum. Контракт контролю доступу реалізує базову логіку перевірки прав користувача, що відображено на лістингу 3.1.

Лістинг 3.1 – Логіка перевірки прав користувача

```
       pragma solidity ^0.8.0;
       contract AccessControl {
           mapping(address => mapping(string => mapping(string => bool)))
permissions;
           address public admin;
           constructor() {
               admin = msg.sender;
           }
           modifier onlyAdmin() {
               require(msg.sender == admin, "Only admin");
               _;
           }
           function grantAccess(address user, string memory resource, string
memory action) public onlyAdmin {
               permissions[user][resource][action] = true;
           }

           function revokeAccess(address user, string memory resource, string
memory action) public onlyAdmin {
               permissions[user][resource][action] = false;
```

```
        }
        function checkAccess(address user, string memory resource, string
memory action) public view returns (bool) {
            return permissions[user][resource][action];
        }
    }
```

кінець лістингу 3.1

Для деплою контракту використовується скрипт Hardhat (ліст. 3.2).

Лістинг 3.2 – Скрипт Hardhat

```
    async function main() {
        const AccessControl = await
ethers.getContractFactory("AccessControl");
        const contract = await AccessControl.deploy();

        await contract.deployed();
        console.log("Contract deployed to:", contract.address);
    }
    main();
```

кінець лістингу 3.2

Блок-схема цього модуля включає етапи компіляції, тестування, деплою та взаємодії зі смарт-контрактом.

На відміну від попередніх підходів, middleware реалізується мовою Go, що забезпечує високу продуктивність та ефективність роботи з мережею. Для взаємодії з Ethereum використовується бібліотека go-ethereum (ліст. 3.3).

Лістинг 3.3 – Конфігурація middleware

```
    package main
    import (
        "context"
        "fmt"
        "log"
        "math/big"
        "github.com/ethereum/go-ethereum/ethclient"
        "github.com/ethereum/go-ethereum/accounts/abi/bind"
    )
    func checkAccess(client *ethclient.Client, contract *AccessControl, user
string) bool {
        result, err := contract.CheckAccess(&bind.CallOpts{
            Context: context.Background(),
        }, user, "pod", "create")
        if err != nil {
            log.Fatal(err)
        }
        return result
    }
    func main() {
        client, err := ethclient.Dial("http://localhost:8545")
```

```
        if err != nil {
            log.Fatal(err)
        }
        fmt.Println("Connected to Ethereum node")
        _ = client
    }
```

кінець лістингу 3.3

У блок-схемі middleware чітко визначаються етапи підключення до вузла Ethereum, формування виклику контракту, обробки відповіді та повернення результату до Kubernetes.

Для глибшої інтеграції з Kubernetes використовується Kubebuilder, який дозволяє створити власний оператор для управління політиками доступу.

Опис кастомного ресурсу (CRD)наведено на лістингу 3.4.

Лістинг 3.4 – Налаштування CRD

```
    type AccessPolicySpec struct {
        User     string `json:"user"`
        Resource string `json:"resource"`
        Action   string `json:"action"`
        Allow    bool   `json:"allow"`
    }
```

кінець лістингу 3.4

Контролер синхронізації стан з блокчейном представлено на лістингу 3.5.

Лістинг 3.5 – Контролер синхронізації

```
    func (r *AccessPolicyReconciler) Reconcile(ctx context.Context, req
ctrl.Request) (ctrl.Result, error) {
        var policy AccessPolicy

        if err := r.Get(ctx, req.NamespacedName, &policy); err != nil {
            return ctrl.Result{}, client.IgnoreNotFound(err)
        }

        if policy.Spec.Allow {
            // виклик smart contract
            fmt.Println("Grant access in blockchain")
        } else {
            fmt.Println("Revoke access in blockchain")
        }

        return ctrl.Result{}, nil
    }
```

кінець лістингу 3.5

Webhook залишається ключовим елементом перевірки доступу в реальному часі. Він взаємодіє з Go-based middleware як це проілюстровано на лістингу 3.6.

Лістинг 3.6 – Механізм взаємодії

```
        func validateRequest(user, resource, action string) bool {
            resp, err := http.Post("http://middleware/check", "application/json",
body)
            if err != nil {
                return false
            }
            // парсинг відповіді
            return true
        }
```

кінець лістингу 3.6

Автоматизація процесів розробки реалізується через GitHub Actions. Pipeline включає етапи тестування смарт-контрактів, збірки Go-сервісу та деплою в Kubernetes (ліст. 3.7).

Лістинг 3.7 – Сценарій GitHub Actions

```
        name: CI/CD Pipeline
        on:
          push:
            branches: [ "main" ]
        jobs:
          build:
            runs-on: ubuntu-latest

            steps:
              - uses: actions/checkout@v3
              - name: Setup Node.js
                uses: actions/setup-node@v3
                with:
                  node-version: 18
              - name: Install dependencies
                run: npm install
              - name: Test smart contracts
                run: npx hardhat test
              - name: Build Go service
                run: go build -o app
              - name: Build Docker image
                run: docker build -t access-control .
              - name: Deploy to Kubernetes
                run: kubectl apply -f k8s/
```

кінець лістингу 3.7

У результаті всі компоненти формують єдину систему: Hardhat забезпечує життєвий цикл смарт-контрактів; Go + go-ethereum реалізують швидкий

middleware; Kubebuilder створює Kubernetes-native управління політиками; Admission Controller виконує runtime-контроль; GitHub Actions автоматизує розгортання.

Побудова блок-схем для кожного модуля дозволяє формалізувати логіку їх роботи: від обробки запиту до прийняття рішення на основі даних блокчейну. Такий підхід забезпечує не лише технічну цілісність системи, але й її зрозумілість для подальшого розвитку.

Таким чином, практична реалізація системи контролю доступу в Kubernetes на основі смарт-контрактів із використанням Hardhat, Go, go-ethereum, Kubebuilder та GitHub Actions демонструє сучасний підхід до побудови безпечних розподілених систем. Поєднання цих технологій дозволяє створити ефективну, масштабовану та відмовостійку систему, яка може бути використана в реальних хмарних середовищах.

3.2 Тестування та налагодження системи

Тестування та налагодження системи контролю доступу в Kubernetes, побудованої на основі смарт-контрактів, є критично важливим етапом розробки, який забезпечує її надійність, безпеку та відповідність функціональним вимогам. З огляду на складність архітектури, що поєднує Kubernetes, блокчейн та middleware-компоненти, процес тестування повинен охоплювати всі рівні системи: від окремих модулів до інтеграційної взаємодії та поведінки в реальному середовищі.

Перш за все, тестування починається з рівня смарт-контрактів, оскільки саме вони визначають політики доступу. Використання середовища Hardhat дозволяє проводити модульне тестування контрактів у локальній мережі, що значно спрощує перевірку логіки доступу. Основна увага приділяється перевірці коректності функцій grantAccess, revokeAccess та checkAccess, а також обробці несанкціонованих викликів.

Приклад тесту для смарт-контракту продемонстровано у лістингу 3.8.

Лістинг 3.8 – Тест для смарт-контракту

```
       const { expect } = require("chai");
       describe("AccessControl", function () {
         let contract, owner, user;

         beforeEach(async function () {
           const AccessControl = await
ethers.getContractFactory("AccessControl");
           [owner, user] = await ethers.getSigners();

           contract = await AccessControl.deploy();
           await contract.deployed();
         });

         it("should grant and verify access", async function () {
           await contract.grantAccess(user.address, "pod", "create");

           const result = await contract.checkAccess(user.address, "pod",
"create");
           expect(result).to.equal(true);
         });

         it("should deny access if not granted", async function () {
           const result = await contract.checkAccess(user.address, "pod",
"delete");
           expect(result).to.equal(false);
         });
       });
```

кінець лістингу 3.8

Ці тести дозволяють виявити помилки логіки ще до розгортання контракту в реальній мережі. Важливо також тестувати граничні випадки, такі як повторне призначення прав або їх відкликання.

Наступним етапом є тестування middleware, реалізованого на Go із використанням бібліотеки go-ethereum (ліст. 3.9). Основна мета – перевірити правильність взаємодії з блокчейном, обробку помилок та стабільність при високому навантаженні. Для цього використовуються unit- та integration-тести.

Лістинг 3.9 – Тест middleware

```
       func TestCheckAccess(t *testing.T) {
           client, _ := ethclient.Dial("http://localhost:8545")
           result := checkAccess(client, contractInstance, "0x123")
           if result != true {
               t.Errorf("Expected true, got false")
        }
```

кінець лістингу 3.9

Окрім цього, важливим аспектом є налагодження підключення до вузла Ethereum. Часто проблеми виникають через неправильну конфігурацію RPC-ендпоінтів або ABI контракту. Для діагностики таких ситуацій застосовується детальне логування. Паралельно виконується тестування Kubernetes-оператора, створеного за допомогою Kubebuilder. Тут перевіряється правильність обробки подій, пов’язаних із кастомними ресурсами (CRD), а також синхронізація стану з блокчейном. Особлива увага приділяється сценаріям, коли зміна політики доступу повинна негайно відобразитися у смарт-контракті (ліст. 3.10).

Лістинг 3.10 – Тестування зміни політики

```
       func (r *AccessPolicyReconciler) Reconcile(ctx context.Context, req
ctrl.Request) (ctrl.Result, error) {
            log.Println("Reconciling access policy")
            // перевірка стану ресурсу
            // виклик blockchain
            return ctrl.Result{}, nil
```

кінець лістингу 3.10

Тестування Admission Controller є ключовим етапом, оскільки саме він відповідає за прийняття рішень у реальному часі (ліст. 3.11). Тут важливо перевірити, як система реагує на різні типи запитів: дозволені, заборонені та некоректні.

Лістинг 3.11 – Тестування Admission Controller

```
       def test_validate_allow(client):
           response = client.post("/validate", json={
               "request": {
                   "userInfo": {"username": "user1"},
                   "resource": {"resource": "pods"},
                   "operation": "CREATE"
                assert response.json["response"]["allowed"] is True
```

кінець лістингу 3.11

Окрім функціонального тестування, проводиться інтеграційне тестування всієї системи. Воно включає розгортання локального Kubernetes-кластера (наприклад, за допомогою Minikube), запуск смарт-контрактів у тестовій мережі та перевірку повного циклу обробки запиту: від користувача до блокчейну і назад.

Для автоматизації тестування використовується GitHub Actions (ліст. 3.12), що дозволяє запускати тести при кожному оновленні коду. Це забезпечує безперервний контроль якості та швидке виявлення помилок.

Лістинг 3.12 – Тестування GitHub Actions

```
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Run smart contract tests
        run: npx hardhat test
      - name: Run Go tests
        run: go test ./...
      - name: Run Python tests
        run: pytest
```

кінець лістингу 3.12

Важливим аспектом є також тестування відмовостійкості, що відображено на лістингу 3.13. Система повинна коректно працювати навіть у випадку тимчасової недоступності блокчейну. Для цього реалізуються механізми повторних запитів та fallback-логіка.

Лістинг 3.13 – Тестування відмовостійкості

```
for i := 0; i < 3; i++ {
    result, err := callContract()
    if err == nil {
        return result
       time.Sleep(time.Second)
}
```

кінець лістингу 3.13

Таким чином, тестування та налагодження системи контролю доступу в Kubernetes на основі смарт-контрактів є багаторівневим процесом, що охоплює всі компоненти системи. Використання сучасних інструментів, таких як Hardhat, go-ethereum, Kubebuilder та GitHub Actions, дозволяє забезпечити високу якість програмного продукту та його готовність до використання в реальних умовах. Комплексний підхід до тестування гарантує не лише коректність роботи системи, але й її стійкість до помилок та атак, що є критично важливим для систем безпеки.

РОЗДІЛ 4
SEO ІНФОРМАЦІЙНО-КОМП'ЮТЕРНОЇ СИСТЕМИ

Пошукова оптимізація (Search Engine Optimization, SEO) є невід'ємною складовою просування сучасних інформаційно-комп'ютерних систем у мережі Інтернет. Для системи контролю доступу в Kubernetes на основі смарт-контрактів SEO відіграє важливу роль у забезпеченні видимості продукту серед цільової аудиторії – розробників, DevOps-інженерів, системних адміністраторів та фахівців у галузі інформаційної безпеки.

Основна мета SEO-оптимізації даної системи полягає у досягненні високих позицій у пошукових системах Google, Bing та DuckDuckGo за відповідними ключовими запитами, пов'язаними із захистом хмарної інфраструктури, управлінням доступом у контейнерних середовищах та технологією смарт-контрактів.

Ефективна SEO-стратегія дозволяє не лише залучити нових користувачів, але й сформувати навколо системи активну спільноту розробників, що сприяє підвищенню якості продукту та його постійному вдосконаленню. Враховуючи специфіку технічної аудиторії, SEO-оптимізація повинна поєднувати технічні аспекти з якісним контентом, орієнтованим на вирішення конкретних завдань у сфері безпеки Kubernetes-середовищ.

Формування семантичного ядра є фундаментальним етапом SEO-оптимізації. Для системи контролю доступу в Kubernetes на основі смарт-контрактів семантичне ядро охоплює три основні категорії запитів: високочастотні, середньочастотні та низькочастотні (LSI-запити).

До групи високочастотних запитів відносяться широкі пошукові терміни із значним обсягом щомісячних пошуків. Для даної системи визначено такі пріоритетні ключові слова:

– kubernetes access control – базовий запит щодо управління доступом у Kubernetes;

– smart contract security – ключовий запит у сфері безпеки смарт-контрактів;

– blockchain access management – управління доступом на основі блокчейну;

– kubernetes RBAC alternative – альтернативи стандартному RBAC у Kubernetes;

– decentralized access control system – децентралізовані системи контролю доступу.

Середньочастотні запити характеризуються більшою конкретизацією та вищим відсотком конверсії завдяки чіткій вираженості наміру користувача. Визначено такі середньочастотні ключові слова:

– Kubernetes smart contract access control – інтеграція смарт-контрактів у систему доступу Kubernetes;

– ethereum-based role management Kubernetes – управління ролями Kubernetes на основі Ethereum;

– blockchain Kubernetes security policy – безпекова політика Kubernetes на основі блокчейну;

– smart contract pod security – захист подів Kubernetes через смарт-контракти;

– web3 Kubernetes authentication – автентифікація у Kubernetes засобами Web3;

– decentralized RBAC implementation – реалізація децентралізованого RBAC.

Низькочастотні та LSI-запити (Latent Semantic Indexing) дозволяють охопити вузькоспеціалізовану аудиторію та покращити семантичну релевантність контенту. До цієї групи належать:

– solidity smart contract for Kubernetes namespace access – смарт-контракти Solidity для управління доступом до просторів імен Kubernetes;

– immutable audit log Kubernetes blockchain – незмінний журнал аудиту Kubernetes на блокчейні;

– zero-trust architecture smart contracts microservices – архітектура нульової довіри зі смарт-контрактами;

– on-chain access policy enforcement – примусове виконання політик доступу в мережі;

– kubernetes admission webhook blockchain verification – верифікація через блокчейн за допомогою Kubernetes admission webhook.

Коректне налаштування мета-тегів є ключовим чинником забезпечення видимості системи у пошукових результатах. Мета-теги формують перше враження про продукт у SERP (Search Engine Results Page) та безпосередньо впливають на показник клікабельності (CTR). В таблиці 4.1 наведено рекомендовані мета-теги для основних сторінок системи.

Таблиця 4.1 – Мета-теги для основних сторінок системи

Сторінка	Title / Meta Description
Головна сторінка (Title)	Kubernetes Access Control via Smart Contracts Secure & Decentralized
Головна сторінка (Description)	Централізована система керування доступом у Kubernetes з використанням смарт-контрактів Ethereum. Забезпечуємо незмінний аудит, автоматичне застосування політик та децентралізовану авторизацію.
Документація (Title)	Smart Contract Kubernetes RBAC Docs | Integration Guide & API Reference
Документація (Description)	Повна документація для інтеграції смарт-контрактів з Kubernetes. Посібники з розгортання, API-референс та приклади конфігурацій безпекових політик.
Блог (Title)	Kubernetes Security Blog | Smart Contracts & Blockchain Access Control
Блог (Description)	Актуальні статті про безпеку Kubernetes, застосування смарт-контрактів у хмарній інфраструктурі та кращі практики управління доступом у контейнерних середовищах.

Для підвищення видимості у пошукових системах рекомендується впровадження структурованих даних згідно зі специфікацією Schema.org. Для даної системи доцільно використовувати типи розмітки: SoftwareApplication (для основного продукту), TechArticle (для документації та блогу), FAQPage (для

сторінок з відповідями на поширені запитання) та HowTo (для покрокових інструкцій з налаштування).

Впровадження JSON-LD розмітки типу SoftwareApplication дозволяє передати пошуковим системам ключові характеристики продукту: назву, категорію, операційну систему, мову програмування, ліцензію та посилання на репозиторій. Це суттєво підвищує шанси на отримання розширених сніпетів у результатах пошуку.

Технічна SEO-оптимізація охоплює аспекти, що безпосередньо впливають на здатність пошукових роботів індексувати та ранжувати сторінки системи. Для документаційного порталу та лендингу системи контролю доступу в Kubernetes необхідно забезпечити виконання низки технічних вимог.

Google використовує метрики Core Web Vitals як фактор ранжування. Для системи необхідно досягти таких цільових показників: Largest Contentful Paint (LCP) – менше 2.5 секунди, що відповідає оцінці «Добре»; First Input Delay (FID) / Interaction to Next Paint (INP) – менше 100 мілісекунд; Cumulative Layout Shift (CLS) – менше 0.1. Для досягнення цих показників рекомендується використання CDN для статичних ресурсів, мінімізація JavaScript та CSS, lazy loading зображень, а також Server-Side Rendering (SSR) або Static Site Generation (SSG) для документаційних сторінок.

Структура URL-адрес повинна відповідати принципам читабельності та семантичної зрозумілості. Рекомендована ієрархія URL для системи: /docs/getting-started, /docs/smart-contracts/overview, /docs/kubernetes/integration, /blog/kubernetes-security, /blog/smart-contracts-access-control. Усі URL мають бути у нижньому регістрі, слова розділяються дефісами, кириличні символи замінюються транслітерацією або англійськими відповідниками.

Для ефективної індексації необхідно забезпечити коректно налаштований файл sitemap.xml, що містить посилання на всі ключові сторінки системи, та robots.txt із відповідними директивами для пошукових роботів. Окрема увага приділяється налаштуванню канонічних URL для уникнення дублювання контенту.

Враховуючи принцип Mobile-First Indexing від Google, документаційний портал та лендинг системи повинні бути повністю адаптовані для мобільних пристроїв. Це передбачає використання адаптивного дизайну (responsive design), коректне відображення блоків коду на малих екранах, зручну навігацію з тач-управлінням та оптимізований розмір шрифтів для читабельності на смартфонах.

Контент є ключовим фактором SEO-успіху для технічних продуктів. Для системи контролю доступу в Kubernetes рекомендується багаторівнева контент-стратегія, що охоплює документацію, технічний блог, навчальні матеріали та case studies.

Технічна документація є одним із найефективніших SEO-активів для систем такого типу. Розробники активно шукають відповіді на конкретні технічні запитання, тому якісна, добре структурована документація природно залучає органічний трафік. Кожна сторінка документації повинна бути оптимізована під конкретний ключовий запит і мати чітку заголовкову ієрархію H1-H6.

Рекомендується створення окремих розділів документації для таких тем: архітектура системи та принципи роботи; інструкція з встановлення та налаштування; API-референс для інтеграції смарт-контрактів; приклади конфігурацій для різних хмарних провайдерів (AWS EKS, Google GKE, Azure AKS); FAQ та усунення типових проблем.

Регулярне публікування технічних статей та туторіалів є ефективним методом нарощування органічного трафіку та формування авторитетності домену. Рекомендований графік публікацій – 2-4 матеріали на місяць. Пріоритетні теми для блогу:

– порівняння традиційного RBAC Kubernetes та підходу на основі смарт-контрактів;

– покрокові туторіали з розгортання системи в різних середовищах;

– аналіз реальних кейсів використання системи в продакшн-оточеннях;

– огляди вразливостей Kubernetes та методи їх усунення засобами смарт-контрактів;

– порівняльні тести продуктивності та безпеки різних підходів до контролю доступу.

Грамотна внутрішня перелінковка підвищує поведінкові показники та допомагає пошуковим роботам ефективніше індексувати контент. Для системи рекомендується реалізація перехресних посилань між документацією та відповідними статтями блогу, використання хлібних крихт (breadcrumbs) для навігації, зв’язування API-референсу з практичними прикладами використання, а також розміщення блоків «Пов’язані матеріали» наприкінці кожної сторінки.

Зовнішня SEO-оптимізація спрямована на нарощування авторитетності домену через отримання якісних зворотних посилань із релевантних ресурсів. Для системи контролю доступу в Kubernetes визначено такі пріоритетні канали отримання посилань:

– GitHub та GitLab – розміщення відкритого вихідного коду сприяє природньому отриманню посилань від розробників, які використовують систему у своїх проєктах;

– публікації на платформах Medium, Dev.to та Habr – технічні статті з посиланнями на документацію системи;

– відповіді на Stack Overflow та Kubernetes Forum із посиланнями на відповідні розділи документації;

– участь у конференціях KubeCon, CloudNativeCon та публікація матеріалів доповідей;

– реєстрація в каталогах CNCF Landscape та Artifact Hub для підвищення видимості серед цільової аудиторії.

Важливим аспектом зовнішньої оптимізації є моніторинг репутації бренду та відстеження згадок системи в мережі. Для цього рекомендується використання інструментів Google Alerts, Mention та Brand24, що дозволяють оперативно реагувати на публікації та перетворювати незалінковані згадки на повноцінні зворотні посилання.

Ефективне SEO-просування неможливе без систематичного аналізу та моніторингу ключових показників. Для системи контролю доступу в Kubernetes

рекомендується впровадження комплексної системи відстеження метрик, що охоплює такі інструменти та показники.

Для відстеження позицій за ключовими словами рекомендується використання спеціалізованих SEO-платформ: Ahrefs, SEMrush або Moz Pro. Ці інструменти також дозволяють аналізувати посилальний профіль конкурентів, виявляти можливості для нарощування посилальної маси та проводити регулярний SEO-аудит сайту.

У четвертому розділі розроблено комплексну SEO-стратегію для інформаційно-комп'ютерної системи контролю доступу в Kubernetes на основі смарт-контрактів. Сформоване семантичне ядро охоплює понад двадцять ключових запитів трьох рівнів частотності, що дозволяє ефективно охопити цільову аудиторію на всіх етапах прийняття рішення.

Розроблені рекомендації щодо мета-тегів, структурованих даних та технічної оптимізації забезпечують необхідну базу для успішного просування у пошукових системах. Визначені вимоги до Core Web Vitals та мобільної адаптації відповідають актуальним стандартам Google.

Запропонована контент-стратегія, що поєднує якісну технічну документацію, регулярні публікації в блозі та активне залучення до спільноти розробників, є оптимальною для нарощування органічного трафіку та формування авторитетності системи серед фахівців у сфері безпеки Kubernetes-середовищ. Систематичний моніторинг визначених SEO-метрик дозволить своєчасно вносити корективи у стратегію та досягти поставлених цілей просування.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне науково-практичне завдання розробки системи контролю доступу в середовищі Kubernetes на основі смарт-контрактів. Актуальність теми обумовлена стрімким розвитком хмарних обчислень, широким використанням контейнерних технологій та зростанням вимог до безпеки розподілених інформаційних систем. У сучасних Kubernetes-кластерах традиційні механізми контролю доступу, зокрема RBAC, мають низку обмежень, пов’язаних із централізованим управлінням політиками безпеки, недостатньою прозорістю аудиту та залежністю від привілейованих адміністраторів. Використання технології блокчейн і смарт-контрактів дозволяє реалізувати децентралізований підхід до управління політиками доступу, забезпечити незмінність журналу аудиту та підвищити рівень довіри до процесів авторизації в Kubernetes-середовищі. У процесі виконання кваліфікаційної роботи було досягнуто поставленої мети та успішно виконано всі визначені завдання дослідження.

У межах аналітичної частини роботи здійснено комплексний огляд існуючих підходів до контролю доступу в Kubernetes: детально досліджено принципи функціонування механізмів RBAC та ABAC, розглянуто архітектуру Admission Webhook як точки розширення авторизаційного конвеєра API Server, а також визначено ключові структурні недоліки традиційних підходів – зокрема централізацію сховища політик, обмежену прозорість аудиту, відсутність криптографічних гарантій незмінності конфігурацій та вразливість до компрометації etcd. Паралельно проведено огляд технології смарт-контрактів на платформі Ethereum, досліджено принципи роботи віртуальної машини EVM, мову Solidity та патерни управління доступом на основі бібліотеки OpenZeppelin, що сформувало теоретичне підґрунтя для подальшого проектування системи.

На етапі проектування розроблено повноцінну архітектуру системи контролю доступу в Kubernetes на основі смарт-контрактів. Визначено склад та відповідальність основних компонентів: ValidatingAdmissionWebhook як точки

перехоплення запитів, Go-сервісу як посередника між Kubernetes API та блокчейн-вузлом, а також смарт-контракту як незмінного сховища політик доступу. Описано логіку обробки запитів авторизації: від формування AccessRequest на основі AdmissionReview до виклику функції verifyAccess(subject, resource, verb) смарт-контракту та повернення рішення Allow або Deny у форматі, сумісному з Kubernetes. Архітектурні рішення зафіксовано у вигляді UML-діаграм компонентів, послідовностей та розгортання.

У ході реалізації розроблено прототип системи, що охоплює три взаємопов’язані складові. Перша – смарт-контракт мовою Solidity, який зберігає ролі, суб'єктів та правила доступу у вигляді незмінних записів у блокчейні, підтримує операції додавання та відкликання прав і емітує події для формування аудиторського сліду. Друга – Webhook-сервіс на мові Go, який перехоплює запити до Kubernetes API Server, виконує JWT-автентифікацію, звертається до смарт-контракту через бібліотеку go-ethereum та повертає рішення авторизації з відповідним HTTP-статусом. Третя – адміністративний CLI-інтерфейс для управління політиками доступу через взаємодію зі смарт-контрактом без прямого редагування конфігурацій кластера. Розроблене рішення забезпечує автоматизовану перевірку політик доступу та незмінну фіксацію результатів у журналі аудиту блокчейну, що унеможливлює ретроактивну модифікацію записів.

На завершальному етапі проведено багаторівневе тестування та оцінювання розробленої системи. Функціональне тестування підтвердило коректність рішень авторизації в усіх передбачених сценаріях: дозвіл легітимних запитів, блокування несанкціонованих звернень, коректна обробка whitelist-суб'єктів та відмовостійка поведінка при недоступності блокчейн-вузла. Навантажувальне тестування дозволило оцінити затримку обробки запитів відносно стандартного механізму RBAC та визначити прийнятність додаткових витрат, зумовлених зверненням до блокчейну. Аналіз безпеки смарт-контракту засобами статичного аналізу підтвердив відсутність критичних вразливостей.

Отримані результати в сукупності підтвердили працездатність запропонованого підходу та його здатність забезпечувати суттєво вищий рівень прозорості, аудитованості й захищеності процесів контролю доступу порівняно зі стандартними механізмами Kubernetes.

Наукова новизна роботи полягає у запропонованому підході до побудови гібридної системи контролю доступу, яка поєднує нативні механізми Kubernetes із децентралізованим управлінням політиками безпеки на основі смарт-контрактів. На відміну від традиційних моделей, запропоноване рішення забезпечує криптографічно верифікований аудит дій, незмінність журналу доступу та усунення залежності від єдиного адміністратора, що створює додатковий рівень захисту для критично важливих хмарних інфраструктур.

Практична цінність отриманих результатів полягає у можливості використання розробленої системи в організаціях, які експлуатують Kubernetes-кластери з підвищеними вимогами до безпеки та відповідності нормативним стандартам. Запропонований webhook-компонент може бути інтегрований у реальні Kubernetes-середовища без модифікації вихідного коду платформи, що спрощує процес впровадження рішення. Розроблена система може бути застосована у фінансовому секторі, медичних інформаційних системах, державних хмарних інфраструктурах та інших критично важливих середовищах, де необхідно забезпечити прозорий аудит дій із привілейованим доступом.

Таким чином, усі поставлені у кваліфікаційній роботі завдання виконано в повному обсязі, а результати дослідження підтверджують доцільність використання смарт-контрактів як механізму децентралізованого контролю доступу в Kubernetes-середовищах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Dynamic Trust Security Architecture for Distributed Service Mesh. URL: https://www.ietf.org/archive/id/draft-si-service-mesh-dta-00.html?utmsource. com (дата звернення: 20.02.2026).

2. Using RBAC Authorization. URL: https://kubernetes.io/ docs/reference/access-authn-authz/rbac/?utm_source (дата звернення: 22.02.2026).

3. Documentation / Solidity. URL: https://docs.soliditylang.org/en/v0.8.35 /?utm_source (дата звернення: 22.02.2026).

4. Ethereum development documentation. URL: https://ethereum.org /developers/docs/?utm_source.com (дата звернення: 25.02.2026).

5. A blockchain based approach for the definition of auditable Access Control systems. URL: https://www.sciencedirect.com/science/article /abs/pii/S0167404818309398?via%3Dihub (дата звернення: 05.03.2026).

6. Open Policy Agent (OPA). URL: https://www.openpolicyagent.org /docs?utm_source.com (дата звернення: 05.03.2026).

7. The X.509 SPIFFE Verifiable Identity Document. URL: https://github.com/spiffe/spiffe/blob/main/standards/X509-SVID.md.com (дата звернення: 10.03.2026).

8. Using RBAC Authorization. URL: https://kubernetes.io/docs/reference /access-authn-authz/rbac/?utm_source.com (дата звернення: 11.03.2026).

9. Kubernetes RBAC. URL: https://www.geeksforgeeks.org/devops/how-to-use-kubernetes-rbac-role-based-access-control/ (дата звернення: 11.03.2026).

10. Enforcing Kubernetes Policies Effortlessly with OPA Gatekeeper: A Step-by-Step Guide. URL: https://mokadisuryaprasad.medium.com/enforcing-policies-in-kubernetes-with-opa-gatekeeper-easy-step-by-step-guide-e21267f1e4ab (дата звернення: 14.03.2026).

11. Kyverno / Installation. URL: https://kyverno.io/docs/installation/ installation/ (дата звернення: 22.03.2026).

12. Захист ваших секретів: Глибоке дослідження HashiCorp Vault. URL: https://dou.ua/forums/topic/46007/ (дата звернення: 28.03.2026).

13. Enhancing identity and access management using Hyperledger Fabric and OAuth 2.0: A block-chain-based approach for security and scalability for healthcare industry. URL: https://www.sciencedirect.com/science/article /pii/S2667345223000470 (дата звернення: 02.04.2026).

14. Що таке SSL/TLS та як проапгрейдитися до версії TLS-протоколу 1.3. URL: https://vps.ua/blog/ukr/ssl-tls-protocols-details/ (дата звернення: 02.04.2026).

15. Ethereum / Transactions. URL: https://ethereum.org/developers /docs/transactions/ (дата звернення: 03.04.2026).

16. Kubernetes Documentation / Webhook Mode. URL: https://kubernetes.io/docs/reference/access-authn-authz/webhook/ (дата звернення: 03.04.2026).

17. Applying CI/CD Using GitHub Actions for Android. URL: https://medium.com/empathyco/applying-ci-cd-using-github-actions-for-android-1231e40cc52f (дата звернення: 04.04.2026).

ДОДАТКИ

ДОДАТОК А

Лістинг Webhook-сервісу на Go

```
package main

import (
	"context"
	"crypto/tls"
	"encoding/json"
	"fmt"
	"log"
	"math/big"
	"net/http"
	"os"

	"github.com/ethereum/go-ethereum/accounts/abi/bind"
	"github.com/ethereum/go-ethereum/common"
	"github.com/ethereum/go-ethereum/ethclient"
	admissionv1 "k8s.io/api/admission/v1"
	metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// --- Конфігурація ---

type Config struct {
	TLSCertFile      string
	TLSKeyFile       string
	EthRPCURL        string
	ContractAddress  string
	ListenAddr       string
}

func loadConfig() Config {
	return Config{
		TLSCertFile:     getEnv("TLS_CERT_FILE", "/certs/tls.crt"),
		TLSKeyFile:      getEnv("TLS_KEY_FILE", "/certs/tls.key"),
		EthRPCURL:       getEnv("ETH_RPC_URL", "http://localhost:8545"),
		ContractAddress: getEnv("CONTRACT_ADDRESS", ""),
		ListenAddr:      getEnv("LISTEN_ADDR", ":8443"),
	}
}
func getEnv(key, fallback string) string {
	if v := os.Getenv(key); v != "" {
		return v
	}
	return fallback
}

// --- Blockchain клієнт ---

type AccessControlContract struct {
	client   *ethclient.Client
	address  common.Address
}
func NewAccessControlContract(rpcURL, contractAddr string) (*AccessControlContract, error) {
	client, err := ethclient.Dial(rpcURL)
	if err != nil {
		return nil, fmt.Errorf("не вдалося підключитися до Ethereum RPC: %w", err)
	}
	return &AccessControlContract{
		client:  client,
		address: common.HexToAddress(contractAddr),
	}, nil
}
// VerifyAccess викликає смарт-контракт verifyAccess(subject, resource, verb)
// Повертає true = ALLOW, false = DENY
func (a *AccessControlContract) VerifyAccess(ctx context.Context, subject, resource, verb string)
(bool, error) {
	// ABI-виклик через низькорівневий call (спрощено для демонстрації)
	// У реальному проєкті використовується згенерований abigen-біндинг
	callData, err := accessControlABI().Pack("verifyAccess", subject, resource, verb)
	if err != nil {
		return false, fmt.Errorf("помилка пакування ABI: %w", err)
	}
	msg := ethereum.CallMsg{
		To:   &a.address,
		Data: callData,
	}

	result, err := a.client.CallContract(ctx, msg, nil)
```

```
	if err != nil {
		return false, fmt.Errorf("помилка виклику контракту: %w", err)
	}

	// Розпаковуємо bool результат
	var allowed bool
	if err := accessControlABI().UnpackIntoInterface(&allowed, "verifyAccess", result); err != nil
{
		return false, fmt.Errorf("помилка розпакування результату: %w", err)
	}

	return allowed, nil
}

// EmitAccessEvent надсилає транзакцію для запису події до blockchain Log
func (a *AccessControlContract) EmitAccessEvent(ctx context.Context, subject, resource, verb string,
allowed bool) {
	// У production реалізації тут — підписана транзакція через auth *bind.TransactOpts
	// Для демонстрації — лише логування
	decision := "DENY"
	if allowed {
		decision = "ALLOW"
	}
	log.Printf("[BlockchainLog] subject=%s resource=%s verb=%s decision=%s",
		subject, resource, verb, decision)
}

// --- Admission Webhook Handler ---

type WebhookServer struct {
	contract *AccessControlContract
}
// AccessRequest — структура, що формується з AdmissionRequest
type AccessRequest struct {
	Subject  string // ім'я користувача з JWT
	Resource string // група/версія/ресурс
	Verb     string // get, create, delete тощо
}
func extractAccessRequest(req *admissionv1.AdmissionRequest) AccessRequest {
	subject := req.UserInfo.Username
	resource := fmt.Sprintf("%s/%s/%s",
		req.Resource.Group,
		req.Resource.Version,
		req.Resource.Resource,
	)
	verb := string(req.Operation)
	return AccessRequest{
		Subject:  subject,
		Resource: resource,
		Verb:     verb,
	}
}
func (ws *WebhookServer) handleAdmission(w http.ResponseWriter, r *http.Request) {
	// --- Крок 1: JWT / TLS автентифікація ---
	// TLS вже перевірено на рівні net/http (mutual TLS або сертифікат сервера).
	// Додатково перевіряємо наявність Bearer-токена.
	if !isAuthenticated(r) {
		log.Println("[AUTH] Автентифікацію не пройдено — повертаємо 401")
		http.Error(w, "Unauthorized", http.StatusUnauthorized) // 401 відхилено
		return
	}
	// --- Крок 2: Декодуємо AdmissionReview ---
	var admissionReview admissionv1.AdmissionReview
	if err := json.NewDecoder(r.Body).Decode(&admissionReview); err != nil {
		http.Error(w, "Bad Request", http.StatusBadRequest)
		return
	}
	req := admissionReview.Request

	// --- Крок 3: Формуємо AccessRequest з AdmissionRequest ---
	ar := extractAccessRequest(req)
	log.Printf("[Webhook] Отримано запит: subject=%s resource=%s verb=%s",
		ar.Subject, ar.Resource, ar.Verb)

	// --- Крок 4: Виклик смарт-контракту verifyAccess(subject, resource, verb) ---
	allowed, err := ws.contract.VerifyAccess(r.Context(), ar.Subject, ar.Resource, ar.Verb)
	if err != nil {
		// Якщо блокчейн недоступний — fail-closed: відхиляємо запит
		log.Printf("[ERROR] Помилка виклику контракту: %v", err)
		allowed = false
	}
```

```
	// --- Крок 5: Запис події до blockchain Log ---
	ws.contract.EmitAccessEvent(r.Context(), ar.Subject, ar.Resource, ar.Verb, allowed)

	// --- Крок 6: Формуємо відповідь AdmissionReview ---
	response := buildAdmissionResponse(req.UID, allowed)
	admissionReview.Response = response

	w.Header().Set("Content-Type", "application/json")
	if err := json.NewEncoder(w).Encode(admissionReview); err != nil {
		log.Printf("[ERROR] Помилка кодування відповіді: %v", err)
	}
}

// buildAdmissionResponse формує Allow або Deny відповідь (403 заборонено)
func buildAdmissionResponse(uid types.UID, allowed bool) *admissionv1.AdmissionResponse {
	resp := &admissionv1.AdmissionResponse{
		UID:     uid,
		Allowed: allowed,
	}
	if !allowed {
		resp.Result = &metav1.Status{
			Code:    403, // 403 заборонено
			Message: "Доступ заборонено смарт-контрактом AccessControl",
		}
	}
	return resp
}
// isAuthenticated перевіряє наявність Bearer JWT у заголовку Authorization
func isAuthenticated(r *http.Request) bool {
	authHeader := r.Header.Get("Authorization")
	if len(authHeader) < 8 || authHeader[:7] != "Bearer " {
		return false
	}
	token := authHeader[7:]
	// У production тут — повноцінна JWT валідація (підпис, exp, iss)
	return len(token) > 0
}

// --- HTTP сервер з TLS ---

func main() {
	cfg := loadConfig()

	contract, err := NewAccessControlContract(cfg.EthRPCURL, cfg.ContractAddress)
	if err != nil {
		log.Fatalf("Помилка ініціалізації контракту: %v", err)
	}

	ws := &WebhookServer{contract: contract}

	mux := http.NewServeMux()
	mux.HandleFunc("/validate", ws.handleAdmission)
	mux.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
		w.WriteHeader(http.StatusOK)
	})

	// TLS конфігурація (обов'язкова для Kubernetes Admission Webhook)
	tlsCert, err := tls.LoadX509KeyPair(cfg.TLSCertFile, cfg.TLSKeyFile)
	if err != nil {
		log.Fatalf("Помилка завантаження TLS-сертифіката: %v", err)
	}

	server := &http.Server{
		Addr:    cfg.ListenAddr,
		Handler: mux,
		TLSConfig: &tls.Config{
			Certificates: []tls.Certificate{tlsCert},
			MinVersion:   tls.VersionTLS13,
		},
	}

	log.Printf("Webhook-сервер запущено на %s", cfg.ListenAddr)
	if err := server.ListenAndServeTLS("", ""); err != nil {
		log.Fatalf("Сервер зупинено: %v", err)
	}
}
```

ДОДАТОК Б

Лістинг функції verifyAccess смарт-контракту

```
package accesscontrol

import (
	"context"
	"fmt"
	"log"
	"sync"
	"time"
)

// --- Типи даних ---

type Verb string
type Labels map[string]string

const (
	VerbGet    Verb = "get"
	VerbCreate Verb = "create"
	VerbUpdate Verb = "update"
	VerbDelete Verb = "delete"
	VerbList   Verb = "list"
)

// PolicyRule описує одне правило дозволу в межах ролі
type PolicyRule struct {
	Resources  []string // ресурси, на які поширюється правило
	Namespaces []string // простори імен ("*" = усі)
	Verbs      []Verb   // дозволені операції
}

// Role — набір правил, прив'язаних до імені ролі
type Role struct {
	Name  string
	Rules []PolicyRule
}

// PolicyNet — прив'язка суб'єкта до ролей у контракті
type PolicyNet struct {
	Subject string
	Roles   []Role
}

// AuditLog — запис події для збереження у storage
type AuditLog struct {
	Timestamp time.Time
	Subject   string
	Namespace string
	Resource  string
	Verb      Verb
	Labels    Labels
	Allowed   bool
	Reason    string
}

// AccessRequest — вхідні параметри з блок-схеми
type AccessRequest struct {
	Subject   string
	Namespace string
	Resource  string
	Verb      Verb
	Labels    Labels
}

// AccessResult — результат перевірки
type AccessResult struct {
	Allowed bool
	Reason  string
}

// --- Інтерфейси залежностей ---

// ContractReader читає стан смарт-контракту (PolicyNet / Role)
type ContractReader interface {
	GetPolicyNet(ctx context.Context, subject string) (*PolicyNet, error)
}

// AuditStorage зберігає записи аудиту
type AuditStorage interface {
	WriteAuditLog(ctx context.Context, entry AuditLog) error
}

// Counter оновлює лічильник дозволених запитів (on-chain або off-chain)
```

```
type Counter interface {
	Increment(ctx context.Context, subject string) error
}

// --- Whitelist ---

type Whitelist struct {
	mu      sync.RWMutex
	entries map[string]struct{}
}

func NewWhitelist(subjects ...string) *Whitelist {
	wl := &Whitelist{entries: make(map[string]struct{})}
	for _, s := range subjects {
		wl.entries[s] = struct{}{}
	}
	return wl
}

func (wl *Whitelist) Contains(subject string) bool {
	wl.mu.RLock()
	defer wl.mu.RUnlock()
	_, ok := wl.entries[subject]
	return ok
}

func (wl *Whitelist) Add(subject string) {
	wl.mu.Lock()
	defer wl.mu.Unlock()
	wl.entries[subject] = struct{}{}
}

// --- Основний верифікатор ---

type AccessVerifier struct {
	whitelist ContractReader  // whitelist як окремий контракт або вбудована логіка
	contract  ContractReader  // основний контракт PolicyNet / Role
	storage   AuditStorage
	counter   Counter
	wl        *Whitelist
}

func NewAccessVerifier(
	contract ContractReader,
	storage AuditStorage,
	counter Counter,
	wl *Whitelist,
) *AccessVerifier {
	return &AccessVerifier{
		contract: contract,
		storage:  storage,
		counter:  counter,
		wl:       wl,
	}
}

// VerifyAccess — головна функція, що реалізує блок-схему
func (v *AccessVerifier) VerifyAccess(ctx context.Context, req AccessRequest) (AccessResult, error) {
	log.Printf("[VerifyAccess] subject=%s namespace=%s resource=%s verb=%s",
		req.Subject, req.Namespace, req.Resource, req.Verb)

	// --- Крок 1: Subject є у whitelist? ---
	if v.wl.Contains(req.Subject) {
		log.Printf("[Whitelist] subject=%s знайдено у whitelist — негайний ALLOW", req.Subject)

		result := AccessResult{Allowed: true, Reason: "subject is whitelisted"}

		// Навіть для whitelist записуємо auditlog
		v.writeAudit(ctx, req, result)
		return result, nil
	}

	// --- Крок 2: Завантажити PolicyNet / Role для subject зі стану контракту ---
	policyNet, err := v.contract.GetPolicyNet(ctx, req.Subject)
	if err != nil {
		log.Printf("[ERROR] Не вдалося отримати PolicyNet для subject=%s: %v", req.Subject,
err)

		// Fail-closed: при помилці контракту — DENY
		result := AccessResult{
			Allowed: false,
			Reason:  fmt.Sprintf("contract read error: %v", err),
```

```
                }
                v.emitDenied(ctx, req, result.Reason)
                v.writeAudit(ctx, req, result)
                return result, nil
        }

        // --- Крок 3: Правило дозволяє verb на resource у namespace? ---
        allowed, matchedRole := evaluatePolicy(policyNet, req)

        if !allowed {
                // --- Гілка НІ: emit Denied, повернути false ---
                reason := fmt.Sprintf(
                        "жодна роль не дозволяє verb=%s на resource=%s у namespace=%s",
                        req.Verb, req.Resource, req.Namespace,
                )
                result := AccessResult{Allowed: false, Reason: reason}

                v.emitDenied(ctx, req, reason)
                v.writeAudit(ctx, req, result)
                return result, nil
        }

        // --- Гілка ТАК: emit Allowed, оновити лічильник, повернути true ---
        reason := fmt.Sprintf("дозволено роллю: %s", matchedRole)
        result := AccessResult{Allowed: true, Reason: reason}

        v.emitAllowed(ctx, req, matchedRole)

        if err := v.counter.Increment(ctx, req.Subject); err != nil {
                // Не критично — логуємо, але не блокуємо
                log.Printf("[WARN] Не вдалося оновити лічильник для subject=%s: %v", req.Subject, err)
        }

        v.writeAudit(ctx, req, result)
        return result, nil
}

// --- Оцінка політики ---

// evaluatePolicy перевіряє всі ролі PolicyNet та повертає
// (true, назва ролі) якщо хоч одне правило дозволяє запит,
// або (false, "") якщо жодне правило не підходить.
func evaluatePolicy(pnet *PolicyNet, req AccessRequest) (bool, string) {
        for  , role := range pnet.Roles {
                for _, rule := range role.Rules {
                        if matchesResource(rule.Resources, req.Resource) &&
                                matchesNamespace(rule.Namespaces, req.Namespace) &&
                                matchesVerb(rule.Verbs, req.Verb) {
                                return true, role.Name
                        }
                }
        }
        return false, ""
}

func matchesResource(resources []string, target string) bool {
        for _, r := range resources {
                if r == "*" || r == target {
                        return true
                }
        }
        return false
}

func matchesNamespace(namespaces []string, target string) bool {
        for _, ns := range namespaces {
                if ns == "*" || ns == target {
                        return true
                }
        }
        return false
}

func matchesVerb(verbs []Verb, target Verb) bool {
        for _, v := range verbs {
                if v == "*" || v == target {
                        return true
                }
        }
        return false
}
```

```
// --- Події (emit) ---

func (v *AccessVerifier) emitAllowed(ctx context.Context, req AccessRequest, roleName string) {
	log.Printf("[Event:Allowed] subject=%s namespace=%s resource=%s verb=%s role=%s",
		req.Subject, req.Namespace, req.Resource, req.Verb, roleName)
	// У on-chain варіанті тут — emit Allowed(subject, resource, verb, roleName)
}

func (v *AccessVerifier) emitDenied(ctx context.Context, req AccessRequest, reason string) {
	log.Printf("[Event:Denied] subject=%s namespace=%s resource=%s verb=%s reason=%s",
		req.Subject, req.Namespace, req.Resource, req.Verb, reason)
	// У on-chain варіанті тут — emit Denied(subject, resource, verb, reason)
}

// --- Запис AuditLog до storage ---

func (v *AccessVerifier) writeAudit(ctx context.Context, req AccessRequest, result AccessResult) {
	entry := AuditLog{
		Timestamp: time.Now().UTC(),
		Subject:   req.Subject,
		Namespace: req.Namespace,
		Resource:  req.Resource,
		Verb:      req.Verb,
		Labels:    req.Labels,
		Allowed:   result.Allowed,
		Reason:    result.Reason,
	}

	if err := v.storage.WriteAuditLog(ctx, entry); err != nil {
		log.Printf("[ERROR] Не вдалося записати AuditLog: %v", err)
	} else {
		log.Printf("[AuditLog] Збережено: subject=%s allowed=%v", req.Subject, result.Allowed)
	}
}
```