МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра мережних технологій

Магістерська робота
освітній ступінь – магістр
на тему:

«ФРЕЙМВОРК ТА РЕКОМЕНДАЦІЇ ДЛЯ ЗАБЕЗПЕЧЕННЯ БЕЗПЕКИ КОМУНІКАЦІЙНИХ КАНАЛІВ У МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ»

Виконав: студент 2-го року навчання,
освітньо-наукової програми
«Інженерія програмного забезпечення», 121
Лук’яненко Євген Олександрович
Керівник: Волинець Є.А.,
ст. викладач, к.н.
Рецензент: _____

Магістерська робота захищена
з оцінкою _____
Секретар ЕК _____
«____» _____ 2026 р.

Київ — 2026

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри мережних технологій,

проф, доктор фіз.-мат.наук

_____ Малашонок Г.І.

(підпис)

„____”_____2026 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студенту 2 року навчання магістерської програми “Інженерія програмного забезпечення”

Лук’яненку Євгену Олександровичу

на тему: «ФРЕЙМВОРК ТА РЕКОМЕНДАЦІЇ ДЛЯ ЗАБЕЗПЕЧЕННЯ БЕЗПЕКИ КОМУНІКАЦІЙНИХ КАНАЛІВ У МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ»

Зміст текстової частини до кваліфікаційної роботи:

Анотація

Зміст

Вступ

1. Огляд існуючих рішень

2. Проектування структури фреймворку

3. Реалізація компонентів фреймворку

Висновки

Список використаних джерел

Додатки

Дата видачі: “___”_____2026 р.

Керівник: к.н. Волинець Є.А. _____(підпис)

Завдання отримав: Лук’яненко Є.О _____(підпис)

Тема: Фреймворк та рекомендації для забезпечення безпеки комунікаційних каналів у мікросервісній архітектурі.

Календарний план виконання роботи:

п/п	Назва етапу роботи	Термін виконання етапу	Примітка
1.	Отримання теми кваліфікаційної роботи		
2.	Пошук і ознайомлення з джерелами		
3.	Написання вступу		
4.	Написання першого розділу		
5.	Написання другого розділу		
6.	Написання третього розділу		
7.	Розробка демонстраційної бібліотеки		
8.	Захист кваліфікаційної роботи		

Студент _____

Керівник _____

“____”_____

АНОТАЦІЯ

У роботі досліджено проблему забезпечення безпеки комунікаційних каналів у мікросервісній архітектурі в умовах суттєвого розширення поверхні атаки, зростання кількості інцидентів, пов’язаних з API, та переходу до хмарно-орієнтованих платформ. Проаналізовано базові стандарти серії NIST SP 800-204, концепцію архітектури нульової довіри (NIST SP 800-207), специфікації OAuth 2.0, OpenID Connect, JSON Web Token, а також практичні підходи до використання сервісної сітки (Istio, Envoy), ідентифікації робочих навантажень (SPIFFE/SPIRE), Policy-as-Code (OPA), керування секретами та сертифікатами (HashiCorp Vault, cert-manager).

На основі узагальнення результатів аналізу сучасних стандартів, архітектурних підходів і практик забезпечення безпеки запропоновано фреймворк, який охоплює вісім функціональних напрямів забезпечення безпеки: контроль периметра та відкритості API, захищене передавання даних із використанням TLS/mTLS, ідентифікацію робочих навантажень, авторизацію на основі токенів, ухвалення та застосування рішень щодо політик доступу за моделлю ABAC, керування секретами й сертифікатами, спостережуваність і аудит, а також виявлення інцидентів, реагування на них і відновлення після них. Зазначені напрями розглядаються не як лінійна послідовність дій, а як модель архітектурних рішень. Концептуальну модель представлено у вигляді демонстраційної proof-of-concept Java-реалізації, з використанням Java 17 і Maven без зовнішніх залежностей часу виконання. Реалізація містить пакети для роботи з ідентифікацією робочих навантажень, перевіркою токенів, точкою ухвалення рішень щодо політик доступу, керуванням секретами, журналюванням подій аудиту та фільтрами на периметрі системи. Фреймворк доповнено матрицею відповідності загроз STRIDE визначеним функціональним напрямам безпеки, чотирирівневою моделлю зрілості, набором ключових показників ефективності та практичними рекомендаціями щодо впровадження.

Робота містить три розділи, висновки, перелік умовних скорочень, список використаних джерел, ілюстрації і таблиці. Практичним результатом Java proof-of-concept фреймворк, що демонструє запропоновану модель у вигляді коду з чотирма демо-сценаріями та unit-тестами.

Ключові слова: мікросервіси, безпека комунікацій, Zero Trust, mTLS, service mesh, SPIFFE/SPIRE, OAuth 2.0, JWT, OPA, NIST SP 800-204, NIST SP 800-207, HashiCorp Vault, cert-manager, Java 17.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ABAC — Attribute-Based Access Control — атрибут-базоване керування доступом

API — Application Programming Interface — програмний інтерфейс застосунку

CA — Certificate Authority — центр сертифікації

CI/CD — Continuous Integration / Continuous Delivery — безперервна інтеграція й доставка

CNCF — Cloud Native Computing Foundation

CNI — Container Network Interface — інтерфейс контейнерних мереж

CVE — Common Vulnerabilities and Exposures

DBIR — Data Breach Investigations Report

DDoS — Distributed Denial of Service

DevSecOps — інтеграція безпеки у процеси DevOps

IAM — Identity and Access Management

JWT — JSON Web Token (RFC 7519)

JWKS — JSON Web Key Set

mTLS — Mutual TLS — взаємна автентифікація на рівні TLS

NIST — National Institute of Standards and Technology (США)

OAuth 2.0 — Authorization framework (RFC 6749)

OIDC — OpenID Connect

OPA — Open Policy Agent

OWASP — Open Worldwide Application Security Project

PDP — Policy Decision Point

PEP — Policy Enforcement Point

PKI — Public Key Infrastructure — інфраструктура відкритих ключів

RBAC — Role-Based Access Control

SBOM — Software Bill of Materials

SIEM — Security Information and Event Management

SPIFFE — Secure Production Identity Framework For Everyone

SPIRE — SPIFFE Runtime Environment

SP — Special Publication (стандарти NIST)

STRIDE — Spoofing, Tampering, Repudiation, Information disclosure, DoS, Elevation of privilege

SVID — SPIFFE Verifiable Identity Document

TLS — Transport Layer Security

WAF — Web Application Firewall

ZTA — Zero Trust Architecture

ВСТУП

Актуальність теми

Перехід індустрії програмного забезпечення на мікросервісну архітектуру став визначальним технологічним зрушенням останнього десятиліття. Замість одного монолітного процесу застосунок реалізується як набір невеликих, незалежно розгортуваних сервісів, що спілкуються через мережу — переважно за допомогою HTTP, gRPC та черг повідомлень. Подібний підхід забезпечує гнучкість, незалежність команд та масштабованість, але одночасно радикально розширює поверхню атаки: кожен сервіс має власні API-точки входу, кожен виклик є мережевою взаємодією, що потенційно проходить через ненадійне середовище.

Згідно зі звітом IBM Security Cost of a Data Breach Report 2024, середня вартість витоку даних у 2024 році становила 4,88 млн дол. США, а в розподілених і мультихмарних середовищах перевищувала 5,27 млн дол. США. При цьому середній час виявлення інциденту становив 283 дні [32]. У звіті Verizon Data Breach Investigations Report 2024 зафіксовано зростання використання вразливостей як початкового вектора атаки на 180 % [33]. Дослідження Akamai засвідчує, що 84 % опитаних фахівців із безпеки повідомили щонайменше про один інцидент, пов’язаний з API, протягом останнього року [36]. Водночас дані Traceable AI свідчать, що лише 13 % організацій здатні запобігти більш ніж половині атак на API [35].

Окремим напрямом проблематики є безпека контейнерної інфраструктури. За даними Sysdig, 87 % образів контейнерів містять критичні або високі за рівнем небезпеки вразливості, 90 % наданих дозволів фактично не використовуються, а 72 % контейнерів існують менше ніж п’ять хвилин. Така коротка тривалість їхнього життєвого циклу суттєво ускладнює цифрове розслідування інцидентів і реагування на них [34].

У відповідь на ці виклики Національний інститут стандартів і технологій США (NIST) опублікував серію документів SP 800-204, SP 800-204A/B/C/D,

присвячених безпеці мікросервісів і підходам DevSecOps [1], [2], [3], [4], [5], а також документ SP 800-207, у якому викладено концепцію архітектури нульової довіри [6]. Крім того, Cloud Native Computing Foundation опублікувала «Cloud Native Security Whitepaper v2» [9], а OWASP у 2023 році оновила перелік основних ризиків безпеки API — API Security Top 10 [8].

Попри значну кількість стандартів і галузевих рекомендацій, залишається потреба в цілісному інженерному фреймворку, який узгоджував би різні рівні безпеки комунікацій — від контролю периметра до цифрового розслідування інцидентів — у вигляді практичної моделі впровадження для організацій із різним рівнем зрілості. Саме це визначає актуальність роботи.

Мета та завдання дослідження

Мета роботи полягає в розробленні цілісного фреймворку забезпечення безпеки комунікаційних каналів у мікросервісній архітектурі у вигляді набору архітектурних можливостей, створенні демонстраційної Java-бібліотеки для підтвердження практичної застосовності запропонованої моделі та формулюванні практичних рекомендацій щодо її впровадження.

Для досягнення мети поставлено такі завдання:

1. Проаналізувати поверхню атаки мікросервісних систем та сучасні класифікації загроз, зокрема модель STRIDE та статистику кіберінцидентів за 2023–2025 роки.
2. Дослідити сучасні документи й рекомендації з безпеки мікросервісів, зокрема серію NIST SP 800-204 та принципи архітектури нульової довіри, викладені в NIST SP 800-207.
3. Порівняти ключові механізми захисту комунікаційних каналів: TLS, mTLS, OAuth 2.0, JWT, OIDC, SPIFFE/SPIRE, сервісну сітку, API Gateway, OPA, Vault і cert-manager.
4. Розробити фреймворк як набір восьми функціональних напрямів забезпечення безпеки, що поєднують розглянуті механізми захисту в межах моделі архітектури нульової довіри.

5. Реалізувати фреймворк у вигляді демонстраційної Java-бібліотеки для підтвердження практичної застосовності запропонованої моделі з використанням Java 17 і Maven, без зовнішніх залежностей часу виконання. Бібліотека має містити модулі ідентифікації, політик доступу, керування секретами та аудиту, які відповідають функціональним напрямам концептуальної моделі.
6. Сформувати матрицю відповідності між загрозами STRIDE та функціональними напрямами фреймворку, а також розробити модель зрілості з визначеними ключовими показниками ефективності.
7. Продемонструвати роботу фреймворку на типових сценаріях: коректний запит, ненадійний видавець токена, прострочений токен, а також заборона доступу на рівні політики. Перевірку реалізації виконати за допомогою модульних тестів і консольної демонстрації.
8. Сформулювати практичні рекомендації щодо поетапного впровадження фреймворку в організаціях різного масштабу, зокрема з урахуванням інтеграції зі Spring Boot та Kubernetes.

Об’єкт і предмет дослідження

Об’єкт — процеси забезпечення безпеки комунікаційних каналів між компонентами мікросервісних систем у хмарних і гібридних середовищах.

Предмет — моделі, механізми, протоколи та архітектурні рішення, що застосовуються для захисту міжсервісної взаємодії, зокрема для забезпечення її конфіденційності, цілісності, доступності й автентичності.

Методи дослідження

У роботі застосовано: системний аналіз — для дослідження архітектури мікросервісних рішень; метод аналогії та порівняльний аналіз — для зіставлення механізмів TLS/mTLS, OAuth 2.0/SPIFFE, Istio та Linkerd; моделювання загроз за моделлю STRIDE — для класифікації основних загроз безпеці комунікаційних каналів; синтез та формалізацію — для побудови архітектурної моделі, що охоплює вісім функціональних напрямів

забезпечення безпеки; аналіз наукових, нормативних і галузевих джерел — для узагальнення положень стандартів NIST, RFC IETF, академічних публікацій і галузевих звітів.

Наукова новизна одержаних результатів

У межах дослідження отримано такі результати:

1. Запропоновано багаторівневу архітектурну модель, яка структурує захист комунікаційних каналів у мікросервісній архітектурі як набір восьми функціональних напрямів забезпечення безпеки: від контролю периметра та доступності API до виявлення інцидентів, реагування на них і відновлення після них. Модель побудовано з урахуванням принципів архітектури нульової довіри. На відміну від розрізненого застосування окремих стандартів і механізмів захисту, запропонована модель подає ці напрями як систему архітектурних рішень, що дає змогу організації добирати конкретні засоби відповідно до власної моделі загроз і рівня зрілості.
2. Уточнено та систематизовано матрицю відповідності між шістьма категоріями загроз STRIDE і функціональними напрямами фреймворку із зазначенням конкретних механізмів захисту. Це дає змогу більш послідовно виконувати моделювання загроз для комунікаційних каналів мікросервісних систем.
3. Запропоновано чотирирівневу модель зрілості (M1–M4) з визначеними критеріями переходу між рівнями та набором ключових показників ефективності. Така модель уможливлює поетапне впровадження функціональних напрямів без потреби в одночасній повній трансформації архітектури.
4. Розроблено демонстраційну Java-реалізацію фреймворку з використанням Java 17 і Maven, без зовнішніх залежностей часу виконання. У реалізації структура пакетів узгоджена з функціональними напрямами фреймворку, а її працездатність

підтверджена компіляцією та модульними тестами. Реалізація розглядається як підтвердження концепції, що ілюструє запропонований архітектурний підхід і взаємодію між компонентами, а не як готова платформа безпеки.

5. Уточнено набір ключових показників ефективності для оцінювання безпеки комунікацій у мікросервісних системах. Запропоновані показники узгоджено з метриками, наведеними в академічних роботах U. Zdun та співавторів, а також із підходами, представленими в галузевих звітах CNCF.

Практичне значення одержаних результатів

Запропонований фреймворк може бути використаний інженерами з інформаційної безпеки, архітекторами хмарних рішень і DevSecOps-командами для розроблення плану захисту мікросервісних застосунків. Модель зрілості та ключові показники ефективності дають змогу обґрунтовувати бюджет, етапи виконання робіт і критерії успішності проєктів, спрямованих на посилення безпеки. Додатково демонстраційна Java-реалізація може бути використана як основа для подальшої інтеграції із виробничими мікросервісами, зокрема в середовищі Spring Boot через механізм OncePerRequestFilter. Її конфігурації та програмні контракти можуть слугувати відправною точкою для подальших впроваджень.

РОЗДІЛ 1 ТЕОРЕТИЧНІ ОСНОВИ БЕЗПЕКИ КОМУНІКАЦІЙ У МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

1.1 Мікросервісна архітектура: принципи, переваги та поверхня атаки

Мікросервісна архітектура — це підхід до проєктування програмних систем, за якого застосунок подається як сукупність невеликих незалежних сервісів, кожен із яких відповідає за окрему бізнес-функцію та взаємодіє з іншими сервісами через стандартизовані мережеві інтерфейси. Основу цього підходу становлять принципи обмеженого контексту (bounded context), високої зв’язності всередині окремого сервісу та слабкої зв’язності між сервісами [1].

До типових переваг мікросервісної архітектури належать незалежне розгортання й масштабування окремих сервісів, можливість використання різних мов програмування та технологічних стеків, ізоляція збоїв, а також спрощення наскрізних процесів CI/CD. Водночас перехід від монолітної архітектури до мікросервісної істотно змінює модель загроз: внутрішні виклики методів замінюються мережевою взаємодією, яка може проходити через недовірені сегменти інфраструктури [29].

Рисунок 1.1 — Порівняння монолітної та мікросервісної архітектур

У монолітних застосунках більшість взаємодій між компонентами відбувається через прямі виклики функцій у межах одного процесу. Такі виклики зазвичай недоступні для зловмисника без попереднього проникнення на рівень операційної системи або середовища виконання. У мікросервісній архітектурі аналогічна взаємодія перетворюється на мережевий виклик, наприклад REST-запит через HTTP або gRPC-виклик поверх HTTP/2, який потенційно проходить через кілька мережевих сегментів, кешів, балансувальників навантаження та сервісних проксі.

Таблиця 1.1 — Порівняння моделей загроз монолітної та мікросервісної архітектур

Критерій	**Моноліт**	**Мікросервіси**
Характер взаємодії між компонентами	Переважно прямі виклики функцій або методів у межах одного процесу	Мережева взаємодія між окремими сервісами через API, REST, gRPC або повідомлення
Межі довіри	Більшість компонентів працює в межах одного застосунку та спільного середовища виконання	Кожен сервіс є окремою одиницею, тому взаємодія між ними потребує окремої перевірки довіри
Поверхня атаки	Порівняно менша, оскільки внутрішня логіка не завжди доступна через мережу	Значно більша через велику кількість API, сервісів, мережевих маршрутів і точок інтеграції
Захист комунікацій	Часто зосереджений на зовнішньому периметрі застосунку	Потребує захисту як зовнішніх, так і внутрішніх міжсервісних комунікацій
Автентифікація та авторизація	Події простіше пов’язати з єдиним застосунком	Потрібна централізована кореляція журналів, трасування запитів і аудит міжсервісної взаємодії

Критерій	Моноліт	Мікросервіси
Реагування на інциденти	Аналіз інциденту часто обмежується одним застосунком або сервером	Аналіз складніший через розподіленість сервісів, короткий життєвий цикл контейнерів і складні маршрути запитів

Серед основних класів загроз, характерних для мікросервісних систем, виокремлюють: горизонтальне поширення атаки після компрометації одного із сервісів (lateral movement), підроблення ідентичності сервісу (service impersonation), незадокументовані API у production-середовищі (shadow APIs), атаки на ланцюг постачання програмного забезпечення (supply chain attacks), зокрема компрометацію базових образів або залежностей, а також виснаження ресурсів унаслідок надмірної кількості викликів [34], [35].

Аналіз ринку свідчить, що API-виклики становлять близько 71 % усього вебтрафіку, а середня організація керує понад 400 API [35]. Це ускладнює підтримання актуального переліку API та контроль їх використання: за даними Cloudflare, понад 30 % API-ендпоінтів виявляються лише за допомогою моделей машинного навчання, тобто належать до незадокументованих або тіньових API [39].

Огляд літератури з безпеки мікросервісів за період 2014–2020 років, що охоплює 290 публікацій, показав, що серед найменш досліджених напрямів залишаються походження та відстежуваність даних, урахування безпеки на ранніх етапах проєктування, а також інтеграція безпекових практик у процеси DevSecOps [29].

1.2 Класифікація загроз STRIDE для мікросервісів

Модель STRIDE, запропонована Microsoft, залишається одним із найпоширеніших підходів до структурованого моделювання загроз. Вона охоплює шість категорій загроз: Spoofing — підроблення ідентичності, Tampering — несанкціонована зміна даних або стану системи, Repudiation — заперечення виконаних дій, Information Disclosure — розкриття інформації,

Denial of Service — відмова в обслуговуванні та Elevation of Privilege — підвищення привілеїв. У контексті мікросервісної архітектури кожна з цих категорій має специфічні прояви, які розглядаються, зокрема, у працях Berardi та ін. [29], а також Volkov і Liubchenko [39].

Рисунок 1.2 — Класифікація загроз STRIDE з прикладами механізмів захисту

Для систематизації загроз безпеці міжсервісної взаємодії доцільно зіставити категорії STRIDE з типовими проявами в мікросервісній архітектурі та відповідними механізмами захисту.

Таблиця 1.2 — STRIDE-загрози у мікросервісах і базові механізми захисту

Категорія	Прояв у мікросервісах	Механізми захисту
Spoofing	Зловмисник або скомпрометований сервіс видає себе за довірений мікросервіс, користувача або системний компонент	mTLS, ідентифікація робочих навантажень, SPIFFE/SPIRE, перевірка токенів, короткострокові сертифікати
Tampering	Зміна запитів, відповідей, токенів, конфігурацій або повідомлень під час передавання між сервісами	TLS/mTLS, підписані токени JWT, контроль цілісності повідомлень, контроль конфігурацій, політики доступу
Repudiation	Сервіс або користувач може заперечити виконання	Централізоване журналювання, аудит,

Категорія	Прояв у мікросервісах	Механізми захисту
	певної дії через відсутність надійних журналів подій	correlation ID, трасування запитів, незмінні журнали подій
Information Disclosure	Несанкціонований доступ до даних через відкриті API, помилки конфігурації, надмірні права або витік секретів	Шифрування трафіку, керування секретами, принцип мінімальних привілеїв, маскування чутливих даних, контроль доступу
Denial of Service	Перевантаження окремих сервісів, API Gateway або внутрішніх залежностей великою кількістю запитів	Rate limiting, circuit breaker, timeout, retry policy, autoscaling, обмеження ресурсів
Elevation of Privilege	Сервіс або користувач отримує доступ до функцій чи даних, на які не має прав	RBAC/ABAC, перевірка claims у токенах, OPA, політики доступу, принцип найменших привілеїв

У мікросервісних системах загрози підроблення ідентичності (Spoofing) та підвищення привілеїв (Elevation of Privilege) часто поєднуються: після компрометації одного сервісу зловмисник може спробувати видати його за інший сервіс і поширити атаку горизонтально [39]. Захист від такого сценарію потребує поєднання ідентифікації робочих навантажень із контролем авторизації на прикладному рівні, тобто з перевіркою прав доступу до конкретних API-запитів, методів і ресурсів.

1.3 Концепція Zero Trust Architecture: принципи й архітектурні компоненти

NIST SP 800-207 визначає архітектуру нульової довіри (Zero Trust Architecture, ZTA) як підхід, за якого довіра не надається неявно, а має постійно перевірятися та переоцінюватися [6]. Цей принцип протиставляється традиційній моделі «безпечного периметра», у межах якої компоненти, розташовані всередині корпоративної мережі, апріорі вважалися довіреними.

Відповідно до NIST SP 800-207, архітектура нульової довіри ґрунтується на таких основних положеннях:

1. Усі джерела даних і обчислювальні сервіси вважаються ресурсами.
2. Усі комунікації захищені незалежно від місцезнаходження мережі.
3. Доступ до окремих корпоративних ресурсів надається для конкретної сесії.
4. Доступ до ресурсу визначається динамічною політикою, що враховує атрибути клієнта, навантаження, поточний контекст.
5. Організація здійснює моніторинг і оцінювання стану цілісності та безпеки всіх власних і пов’язаних активів.
6. Автентифікація та авторизація є динамічними, застосовуються примусово та виконуються до надання доступу.
7. Організація збирає максимум інформації про поточний стан активів, інфраструктури й комунікацій та використовує її для покращення стану безпеки.

Логічна архітектура ZTA охоплює кілька ключових компонентів: Policy Engine — компонент, що ухвалює рішення щодо доступу; Policy Administrator — компонент, який реалізує ухвалене рішення через керування станом доступу; Policy Enforcement Point — точка застосування політики доступу; а також Subject — суб’єкт, який ініціює запит. У мікросервісному середовищі функції точки застосування політики найчастіше може виконувати sidecar-проксі сервісної сітки, тоді як функції ухвалення рішень щодо політик можуть реалізовуватися за допомогою OPA або вбудованих механізмів сервісної сітки [7], [20].

Особливістю NIST SP 800-207A є адаптація принципів архітектури нульової довіри до хмарно-орієнтованих, мультихмарних і гібридних середовищ [7]. Документ акцентує увагу на необхідності використання політик, що враховують ідентичність користувачів, сервісів і робочих навантажень незалежно від їхнього розташування. У мікросервісних системах

такий підхід може бути реалізований через федерацію доменів між кластерами, зокрема за допомогою механізмів SPIFFE/SPIRE.

Таблиця 1.3 — Зіставлення компонентів NIST SP 800-207 та технічних реалізацій у мікросервісному середовищі

Компонент ZTA	**Функція згідно NIST 800-207**	**Приклад реалізації у мікросервісах**
Policy Engine (PE)	Ухвалює рішення щодо надання або заборони доступу на основі політик і контексту	OPA, вбудований механізм політик сервісної сітки, власний модуль ухвалення рішень
Policy Administrator (PA)	Реалізує рішення Policy Engine, створюючи або змінюючи умови доступу	Control plane сервісної сітки, компонент керування конфігураціями, система видачі токенів або сертифікатів
Policy Enforcement Point (PEP)	Застосовує політику доступу безпосередньо під час звернення до ресурсу	API Gateway, sidecar-проксі, ingress controller, фільтр у Spring Boot
Subject	Ініціює запит до ресурсу та проходить автентифікацію й авторизацію	Workload із SPIFFE ID або користувач з OIDC сесією
Policy Information Point	Надає контекст для ухвалення рішення: атрибути суб’єкта, ресурсу, середовища або ризику	IAM, каталог сервісів, Kubernetes API, система моніторингу, сховище атрибутів
Continuous Diagnostics and Mitigation	Збирає дані про стан активів, конфігурацій, вразливостей і подій безпеки	SIEM, система журналювання, Prometheus/Grafana, сканери вразливостей, засоби аудиту

Компонент ZTA	Функція згідно NIST 800-207	Приклад реалізації у мікросервісах
Data Access Policies	Визначають правила, за якими суб’єкт отримує або не отримує доступ до ресурсу. Політики можуть враховувати ідентичність суб’єкта, атрибути ресурсу, контекст запиту, стан пристрою або сервісу та рівень ризику.	Визначають правила, за якими суб’єкт отримує або не отримує доступ до ресурсу. Політики можуть враховувати ідентичність суб’єкта, атрибути ресурсу, контекст запиту, стан пристрою або сервісу та рівень ризику.

Важливою рисою архітектури нульової довіри є можливість поетапного впровадження та поступового посилення контролю доступу без повної перебудови всієї інфраструктури. NIST SP 800-207 описує три типові підходи до впровадження ZTA: посилене керування ідентичностями (enhanced identity governance), мікросегментацію (micro-segmentation) та програмно визначений периметр (software-defined perimeter) [6]. У мікросервісному середовищі найчастіше застосовується поєднання перших двох підходів: посиленої ідентифікації сервісів і робочих навантажень, а також мікросегментації взаємодії між сервісами за допомогою сервісної сітки, OPA та механізмів ідентифікації робочих навантажень [7], [31].

1.4 Огляд існуючих підходів та стандартів

Сучасна нормативна й методична база безпеки мікросервісів формується на перетині документів і рекомендацій кількох організацій: NIST, IETF, OpenID Foundation, OWASP, CNCF, а також результатів академічних досліджень. NIST розробляє спеціальні публікації з питань безпеки мікросервісів, IETF стандартизує відповідні інтернет-протоколи та формати через документи RFC, OpenID Foundation підтримує специфікації у сфері автентифікації та федеративної ідентичності, OWASP систематизує прикладні

ризики безпеки, а CNCF формує рекомендації для хмарно-орієнтованої екосистеми. Найбільш цілісно питання безпеки мікросервісів розглядаються в серії документів NIST SP 800-204 [1], [2], [3], [4], [5].

Таблиця 1.4 — Порівняння стандартів серії NIST SP 800-204

Стандарт	Рік	Тема	Ключові механізми
SP 800-204	2019	Базові стратегії безпеки мікросервісів	API Gateway, service mesh, threat model
SP 800-204A	2020	Service mesh як інструмент безпеки	mTLS, sidecar proxy, control plane
SP 800-204B	2021	ABAC у service mesh	JWT, claims, attribute-based policies
SP 800-204C	2022	DevSecOps + service mesh	CI/CD pipeline, secure deployment
SP 800-204D	2024	Software supply chain security	SBOM, image signing, provenance
SP 800-207	2020	Zero Trust Architecture	PE/PA/PEP, dynamic policy
SP 800-207A	2023	ZTA для cloud-native multi-cloud	Федерація trust domains, identity

OWASP API Security Top 10 версії 2023 року виокремлює низку провідних ризиків безпеки API, зокрема порушення авторизації на рівні об’єктів (Broken Object Level Authorization, BOLA), порушення автентифікації (Broken Authentication), порушення авторизації на рівні властивостей об’єкта (Broken Object Property Level Authorization), необмежене споживання ресурсів (Unrestricted Resource Consumption), порушення авторизації на рівні функцій (Broken Function Level Authorization) та підроблення серверних запитів (Server-Side Request Forgery, SSRF) [8]. Значна частка API-інцидентів пов’язана саме з BOLA, що підкреслює необхідність перевірки прав доступу на рівні окремих об’єктів. Такий контроль може реалізовуватися за допомогою атрибутно-орієнтованої моделі авторизації ABAC та інструментів керування політиками доступу, зокрема OPA.

CNCF Cloud Native Security Whitepaper v2 пропонує життєвий цикл безпеки хмарно-орієнтованих систем, який охоплює етапи розроблення, поширення, розгортання та виконання (Develop → Distribute → Deploy → Runtime) із застосуванням принципів архітектури нульової довіри на кожному з них [9]. У цій роботі зазначений документ використовується як орієнтир для функціональних напрямів розробленого фреймворку, пов'язаних із безпекою ланцюга постачання програмного забезпечення та спостережуваністю під час виконання.

Серед академічних робіт особливий інтерес становить дослідження U. Zdun та співавторів, у якому формалізовано метрики безпеки мікросервісів за трьома вимірами: захищена комунікація, керування ідентичностями та спостережуваність [30]. Ці метрики покладено в основу системи ключових показників ефективності, розглянутої в розділі 3.

Таблиця 1.5 — Зіставлення ризиків OWASP API Security Top 10 (2023) із функціональними напрямками розробленого фреймворку

ID	Назва ризику	Функціональний напрямок	Ключовий механізм захисту
API1	Broken Object Level Authorization (BOLA)	Токен-орієнтована авторизація; ухвалення та застосування політик доступу	ABAC/RBAC, перевірка прав доступу до конкретного об'єкта, OPA/Rego
API2	Broken Authentication	Ідентифікація робочих навантажень; токен-орієнтована авторизація	OIDC/OAuth 2.0, перевірка JWT, короткострокові токени, mTLS
API3	Broken Object Property Level Authorization	Ухвалення та застосування політик доступу	Перевірка доступу до окремих полів об'єкта, фільтрація відповіді, політики ABAC

ID	Назва ризику	Функціональний напрямок	Ключовий механізм захисту
API4	Unrestricted Resource Consumption	Контроль периметра та відкритості API	Обмеження частоти запитів, квоти, тайм-аути, circuit breaker
API5	Broken Function Level Authorization	Ухвалення та застосування політик доступу	RBAC/ABAC, перевірка прав на рівні API-методів
API6	Unrestricted Access to Sensitive Business Flows	Контроль периметра та відкритості API; спостережуваність і аудит	Обмеження частоти операцій, поведінковий аналіз, аудит критичних бізнес-операцій
API7	Server-Side Request Forgery (SSRF)	Контроль периметра та відкритості API	Валідація URL, перелік дозволених зовнішніх адрес, сегментація мережі, обмеження вихідних запитів
API8	Security Misconfiguration	Керування секретами й сертифікатами; контроль периметра	Безпечні конфігурації, сканування налаштувань, централізоване керування секретами
API9	Improper Inventory Management	Спостережуваність і аудит	Інвентаризація API, каталог сервісів, виявлення тіньових API, моніторинг змін
API10	Unsafe Consumption of APIs	Виявлення, реагування та	Перевірка сторонніх API,

ID	Назва ризику	Функціональний напрямок	Ключовий механізм захисту
		відновлення; контроль периметра	валідація відповідей, обмеження довіри до зовнішніх сервісів

Як видно з таблиці, значна частина ризиків OWASP API Security Top 10 пов’язана з порушеннями автентифікації, авторизації та контролю доступу. У межах розробленого фреймворку ці ризики переважно покриваються напрямками, що відповідають за авторизацію на основі токенів і використання політик доступу у вигляді коду. Це свідчить про доцільність пріоритетного впровадження зазначених рівнів, оскільки вони безпосередньо впливають на зменшення ключових ризиків для безпеки API [8].

Розглянуті теоретичні основи — модель загроз STRIDE, принципи архітектури нульової довіри, а також рекомендації NIST, OWASP і CNCF — формують підґрунтя для подальшого аналізу окремих механізмів захисту комунікаційних каналів. У наступному розділі послідовно розглянуто транспортну безпеку на основі TLS/mTLS, ідентифікацію робочих навантажень із використанням SPIFFE/SPIRE, авторизацію на основі токенів із використанням OAuth 2.0, OIDC і JWT, архітектуру сервісної сітки та роль API Gateway. Кожен із цих механізмів оцінюється з погляду забезпечення конфіденційності, цілісності й автентичності комунікаційного каналу, а також з урахуванням накладних витрат і можливостей автоматизації.

РОЗДІЛ 2 АНАЛІЗ МЕХАНІЗМІВ АВТЕНТИФІКАЦІЇ, АВТОРИЗАЦІЇ ТА ЗАХИСТУ КАНАЛІВ

2.1 Транспортна безпека: TLS та mTLS у мікросервісах

TLS (Transport Layer Security) забезпечує конфіденційність і цілісність даних, що передаються між двома учасниками комунікації. У стандартній моделі TLS сервер надає клієнту сертифікат X.509 для підтвердження своєї ідентичності, тоді як клієнт зазвичай не проходить сертифікатну автентифікацію перед сервером. Для мікросервісного середовища цього часто недостатньо, оскільки клієнтом у такій взаємодії може бути інший сервіс, ідентичність якого також має бути перевірена.

Mutual TLS (mTLS) розширює цю модель: обидві сторони надають сертифікати X.509 і перевіряють ідентичність одна одної на основі довіреного набору сертифікатів або відповідного центру сертифікації. У мікросервісних системах mTLS використовується як один із ключових механізмів автентифікації між сервісами, зокрема в рекомендаціях NIST SP 800-204A [2], а також є базовим елементом багатьох рішень сервісної сітки [20], [22].

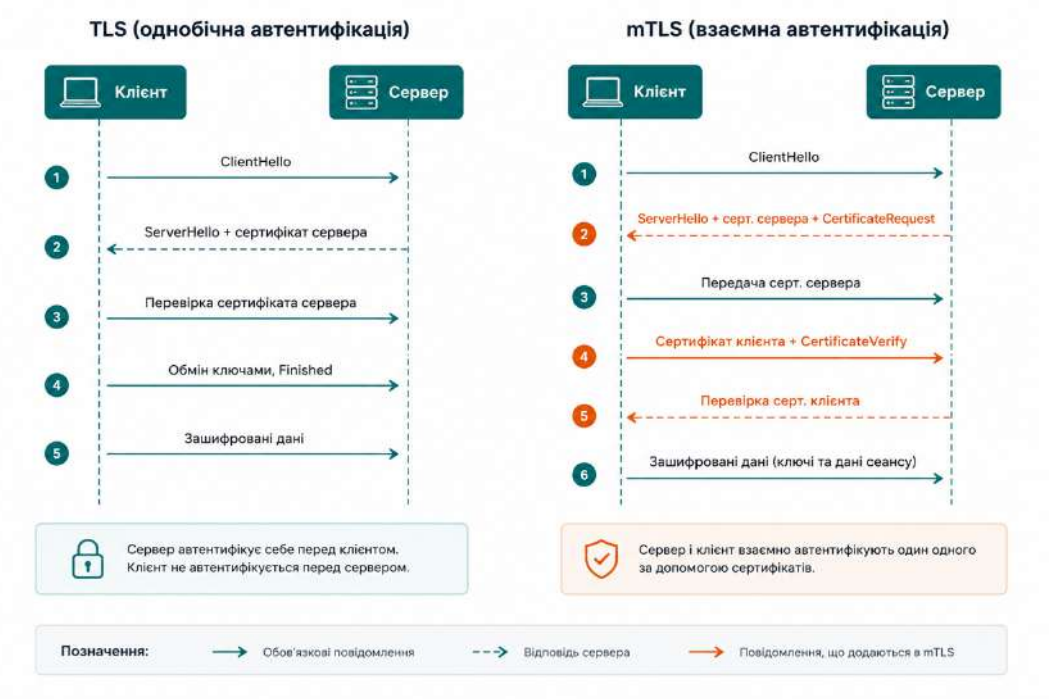

Рисунок 2.1 — Порівняння TLS і mTLS handshake

Згідно з опитуванням CNCF, 79 % організацій, які використовують сервісну сітку у виробничому середовищі, покладаються на mTLS для зниження ризиків, пов’язаних із внутрішнім міжсервісним трафіком (East–West traffic) [18].

Таблиця 2.1 — Порівняння TLS, mTLS та mTLS із використанням SPIFFE у мікросервісній архітектурі

Характеристика	**TLS**	**mTLS**	**mTLS + SPIFFE**
Основна мета	Захист каналу між клієнтом і сервером	Взаємна автентифікація сторін і захист каналу	Взаємна автентифікація сервісів із використанням стандартизованої ідентичності робочих навантажень
Хто підтверджує ідентичність	Зазвичай лише сервер	Сервер і клієнт	Сервер і клієнт, але ідентичність прив’язана не лише до сертифіката, а й до SPIFFE ID
Тип ідентичності	Доменне ім’я або серверний сертифікат	Сертифікати X.509 для обох сторін	SPIFFE ID
Придатність для мікросервісів	Обмежена, бо не підтверджує ідентичність сервісу-клієнта	Вища, бо дає змогу перевіряти обидві сторони взаємодії	Найвища, бо забезпечує єдиний підхід до ідентифікації сервісів у розподіленому середовищі
Керування сертифікатами	Переважно ручне або через PKI	Складніше через потребу видачі	Автоматизоване через SPIRE або іншу

Характеристика	TLS	mTLS	mTLS + SPIFFE
		сертифікатів кожному сервісу	інфраструктуру ідентичностей
Підтримка Zero Trust	Часткова	Висока	Найбільш повна для міжсервісної взаємодії

Для практичного впровадження mTLS у мікросервісному середовищі доцільно використовувати сервісну сітку з поступовим переходом до режиму обов'язкової взаємної автентифікації. На етапі міграції може застосовуватися режим PERMISSIVE, який дозволяє одночасно приймати як зашифрований, так і незашифрований трафік. Після завершення міграції доцільно перейти до режиму STRICT, за якого міжсервісна взаємодія без mTLS відхиляється [21]. Криптографічні параметри мають відповідати TLS 1.2 як мінімально допустимому варіанту або TLS 1.3 як рекомендованому варіанту. Крім того, необхідно заборонити використання застарілих наборів шифрів.

2.2 Ідентифікація workload: SPIFFE/SPIRE та SVID

Проблема ідентифікації робочих навантажень є фундаментальною для архітектури нульової довіри. Класичні сертифікати X.509 зазвичай пов'язані з DNS-іменами, однак не завжди відображають семантичний контекст на кшталт «це сервіс A у виробничому кластері». SPIFFE (Secure Production Identity Framework For Everyone) пропонує універсальний URI-формат ідентифікатора виду spiffe://trust-domain/path [10], [17].

SPIRE є референсною реалізацією SPIFFE, яка видає SVID (SPIFFE Verifiable Identity Document) у двох формах: X.509-SVID та JWT-SVID. Видача ідентичностей спирається на процедуру атестації: SPIRE Agent спочатку атестує вузол, наприклад через kubelet або AWS Instance Identity Document, а потім — окремі робочі навантаження, зокрема на основі UID, простору імен або projected service account token у Kubernetes [11], [19].

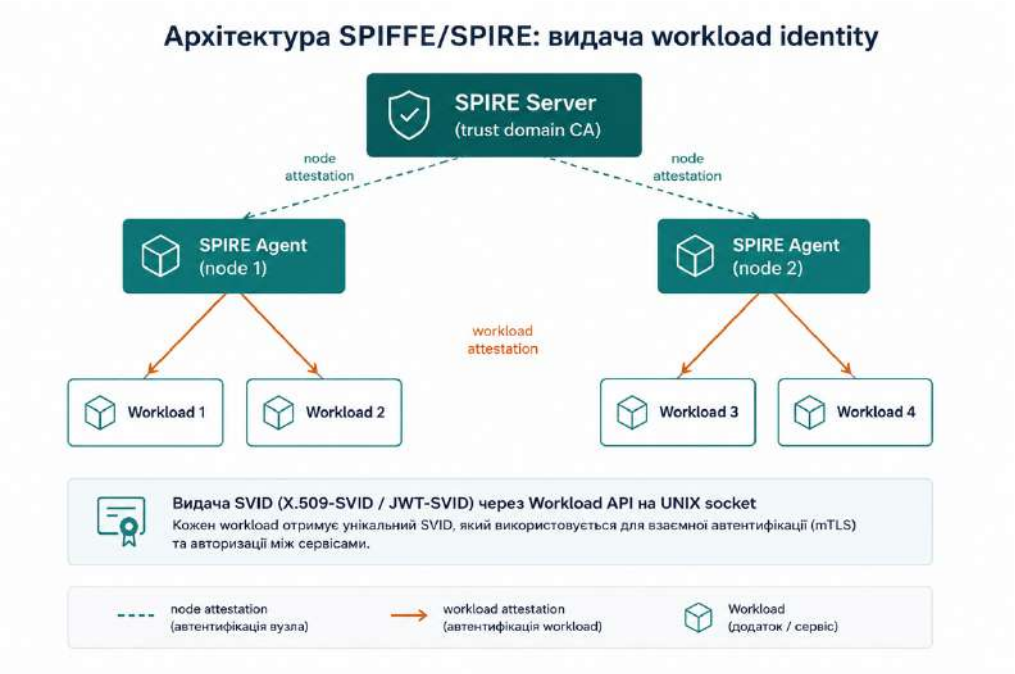

Рисунок 2.2 — Архітектура SPIFFE/SPIRE та видача workload identity

Ключова перевага SPIFFE/SPIRE полягає в тому, що робоче навантаження не потребує попередньо встановлених секретів. Воно отримує SVID через локальний UNIX-сокет за допомогою Workload API, а SPIRE Agent видає сертифікат лише після успішної атестації [17]. Такий підхід розв'язує класичну проблему початкового безпечного поширення секретів, коли для отримання секрету вже потрібен інший довірений секрет.

У роботі [37] експериментально показано, що поєднання моделі нульової довіри зі SPIFFE/SPIRE та OIDC дає змогу досягти високого рівня ізоляції робочих навантажень без істотного впливу на продуктивність. Подібний результат отримано в [38] для фреймворків сервісної сітки. Приклад SPIFFE ID для сервісу обробки замовлень у виробничому кластері: spiffe://prod.example.com/ns/orders/sa/orders-service

Домен prod.example.com визначає межі однієї довіреної зони. SPIRE підтримує федерацію між доменами довіри: довірений набір сертифікатів одного SPIRE Server може експортуватися іншому, що дає змогу сервісам із різних кластерів взаємно автентифікуватися без використання спільного центру сертифікації. Архітектурна перевага SPIFFE полягає у відмові від мережевих ідентифікаторів, таких як IP-адреси або DNS-імена, на користь

криптографічно підтвердженої ідентичності робочого навантаження. Це відповідає принципу NIST SP 800-207, згідно з яким доступ має визначатися динамічно з урахуванням атрибутів суб'єкта, ресурсу та контексту запиту, а не лише мережевого розташування [6].

2.3 Авторизація на основі токену: OAuth 2.0, OIDC, JWT

OAuth 2.0, визначений у RFC 6749, є базовим фреймворком авторизації, який описує кілька способів отримання токенів доступу: Authorization Code, Implicit, Resource Owner Password Credentials, Client Credentials, Refresh Token і Device Code [12]. Водночас деякі з них сьогодні мають обмежене застосування: наприклад, Implicit Grant вважається застарілим, а Resource Owner Password Credentials Grant не рекомендовано використовувати в сучасних мікросервісних системах. Для міжсервісної взаємодії основним варіантом зазвичай є Client Credentials Grant. У цьому сценарії сервіс-клієнт отримує access_token, пред'являючи власні облікові дані серверу авторизації. Отриманий токен надалі використовується для звернення до іншого сервісу або API.

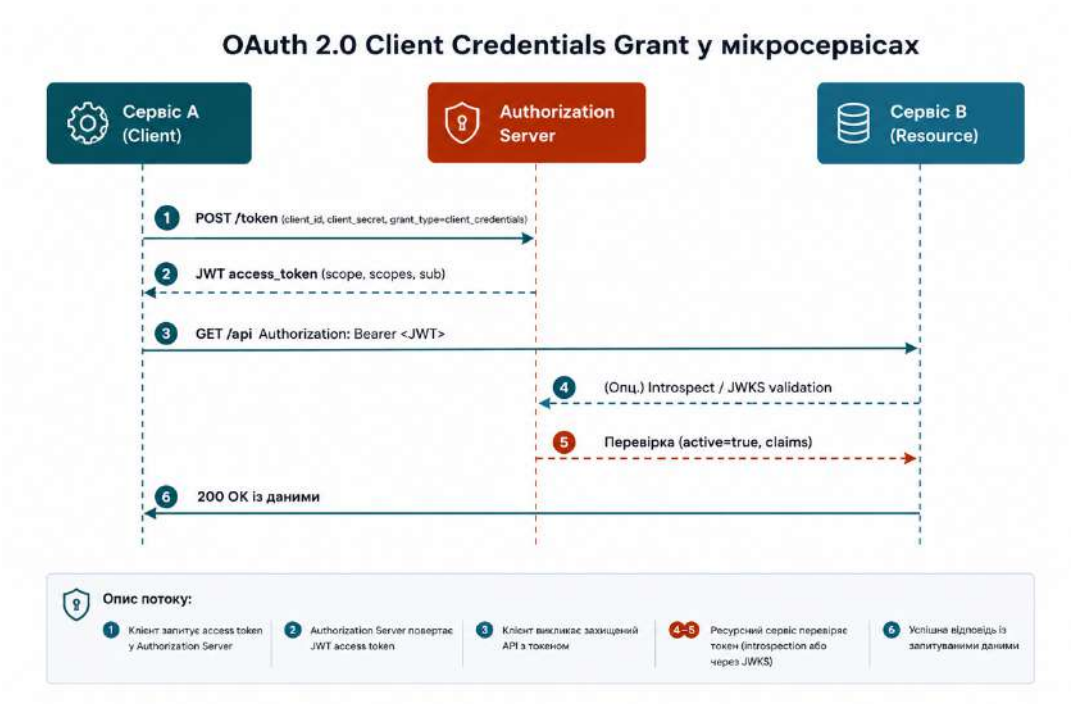

Рисунок 2.3 — Послідовність кроків OAuth 2.0 Client Credentials Grant

JWT (JSON Web Token), визначений у RFC 7519, є компактним форматом токена, що містить набір тверджень (claims) про суб'єкта, видавця, аудиторію та час дії токена. До типових полів JWT належать iss, sub, aud, exp, iat, jti, scope, а також додаткові користувацькі поля [13]. JWT може бути підписаний за допомогою JWS або зашифрований із використанням JWE. У практиці OAuth 2.0 JWT часто використовується як токен доступу, а в OpenID Connect — як ID Token.

OpenID Connect (OIDC) є шаром автентифікації поверх OAuth 2.0, який формалізує поняття ID Token та ендпоінта UserInfo [15]. Через механізм OIDC Discovery, доступний за шляхом /.well-known/openid-configuration, клієнти отримують метадані сервера авторизації, зокрема адресу JWKS (JSON Web Key Set), підтримувані алгоритми підпису та інші параметри конфігурації [14], [16].

Таблиця 2.2 — OAuth 2.0 grant types у мікросервісах

Grant type	Призначення	Рекомендації
Authorization Code + PKCE	Автентифікація користувача та отримання токена доступу клієнтським застосунком	Рекомендовано для вебзастосунків, мобільних застосунків і SPA; PKCE слід використовувати як обов'язковий механізм захисту
Client Credentials	Отримання токена доступу сервісом без участі користувача	Рекомендовано для міжсервісної взаємодії; слід поєднувати з mTLS або SPIFFE/SPIRE для підтвердження ідентичності сервісу
Device Code	Автентифікація пристроїв з обмеженим інтерфейсом введення	Доречно для CLI, Smart TV, IoT або пристроїв без повноцінного браузера

Grant type	Призначення	Рекомендації
Refresh Token	Оновлення токена доступу без повторної автентифікації користувача	Використовувати з ротацією refresh-токенів, обмеженим строком дії та можливістю відкликання
Resource Owner Password Credentials	Передавання логіна й пароля користувача безпосередньо клієнтському застосунку	Не рекомендовано; порушує принцип мінімізації довіри до клієнта
Implicit	Історично використовувався для браузерних застосунків	Не рекомендовано для нових систем; замінювати на Authorization Code + PKCE

До поширених антипатернів використання JWT належать: відсутність перевірки цифрового підпису токена, прийняття токенів із параметром alg: none, надмірно тривалий строк дії токенів без механізму відкликання, а також зберігання спільних секретів безпосередньо в образах контейнерів [8], [13]. Рекомендованою практикою є використання короткого строку дії токена доступу, наприклад 5–15 хвилин, окремого refresh-токена з ротацією, а також перевірка тверджень iss, aud і exp під час кожного звернення до захищеного ресурсу.

Структура JWT складається з трьох частин: заголовка, корисного навантаження та підпису:

header.payload.signature

Приклад заголовка JWT:

```
{
"alg": "RS256",
"kid": "2026-q2",
"typ": "JWT"
}
```

Приклад корисного навантаження JWT:

```
{
"iss": "https://auth.example.com",
"sub": "svc-orders",
"aud": "svc-billing",
"exp": 1747043200,
"iat": 1747042600,
"jti": "01HXZ9...",
"scope": "orders:read billing:write"
}
```

Кожне твердження (claim) виконує окрему функцію. Поле iss визначає сервер авторизації та дає змогу перевірити, чи належить він до переліку довірених видавців токенів. Поле aud обмежує призначення токена конкретним сервісом або API та зменшує ризик атаки типу confused deputy. Поля exp та iat обмежують строк дії токена, а jti дає змогу відстежувати конкретний токен, зокрема для реалізації списку відкликаних токенів або виявлення повторного використання. Поле scope визначає дозволені операції відповідно до принципу найменших привілеїв.

2.4 Сервісна сітка як інфраструктурний шар безпеки

Сервісна сітка — це інфраструктурний шар, який відокремлює міжсервісну взаємодію від коду застосунку та переносить частину функцій комунікації й безпеки на рівень спеціалізованих проксі або мережевих компонентів. До найпоширеніших реалізацій сервісної сітки належать Istio, Linkerd, Cilium Service Mesh і Consul Connect. У NIST SP 800-204A сервісна сітка розглядається як один із важливих інфраструктурних засобів захисту мікросервісної взаємодії, оскільки вона може забезпечувати взаємну TLS-автентифікацію, політики авторизації, обмеження частоти запитів і спостережуваність [2].

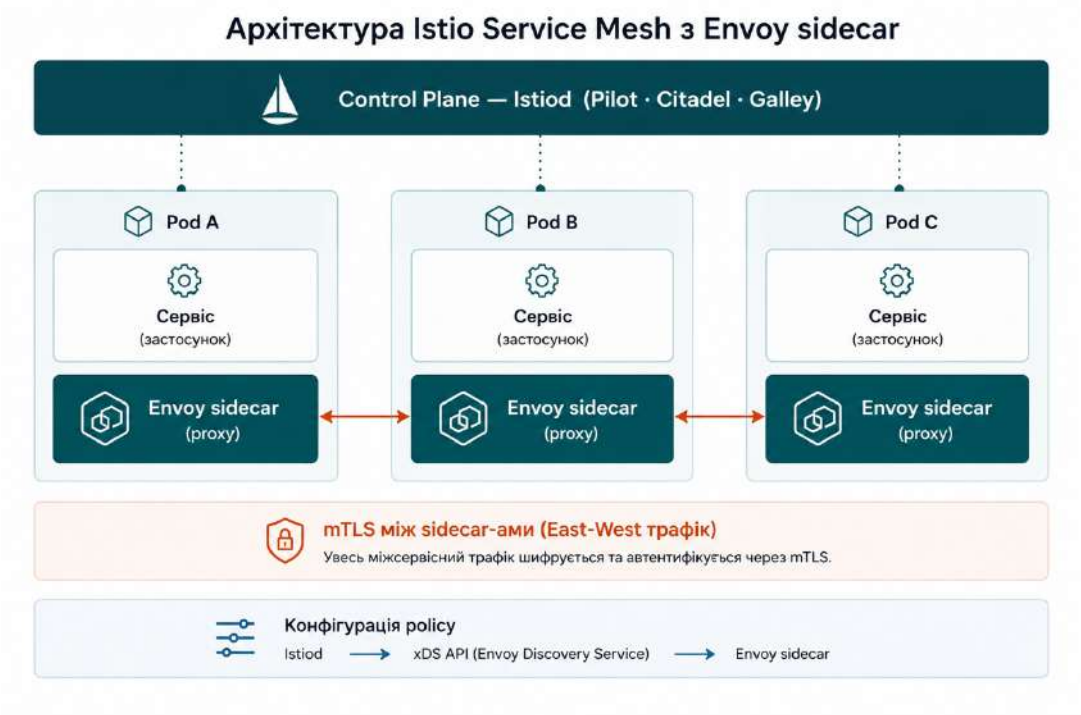

Рисунок 2.4 — Архітектура Istio service mesh з Envoy sidecar

Envoy Proxy — це високопродуктивний проксі прикладного рівня, який широко використовується як sidecar-компонент у рішеннях сервісної сітки [22], [23]. Envoy підтримує завершення TLS/mTLS-з'єднань, перевірку JWT, фільтри рольового контролю доступу, обмеження частоти запитів, механізми запобігання каскадним відмовам, політики повторних спроб і розподілене трасування.

Таблиця 2.3 — Порівняння провідних рішень сервісної сітки

Критерій	**Istio**	**Linkerd**	**Cilium Servie Mesh**	**Consul Connect**
Площина передавання даних	Envoy	Linkerd2-proxy, написаний Rust	eBPF, Cilium Agent, Envoy для L7-функцій	Envoy
Підтримка mTLS	Так, режими STRICT і PERMISSIVE	Так, автоматично	Так	Так
Підтримка SPIFFE	Так	Часткова	Так	Так

Критерій	Istio	Linkerd	Cilium Servie Mesh	Consul Connect
Складність упровадження	Висока	Низька	Середня	Середня
CNCF статус	Graduated	Graduated	Graduated	Не є проєктом CNCF
Політики рівня L7	AuthorizationPolicy	ServerAuthorization / HTTPRoute-політики	Cilium Network Policy / Cilium L7 Policy	Intentions

Отже, Istio забезпечує найбільш широкий набір можливостей для керування трафіком і політиками безпеки, однак має вищу складність упровадження. Linkerd орієнтований на простоту та автоматизоване застосування mTLS. Cilium Service Mesh поєднує механізми сервісної сітки з eBPF-підходом до мережевої безпеки, а Consul Connect є доцільним варіантом для середовищ, де вже використовується HashiCorp Consul.

Дослідження [18] показало, що накладні витрати, пов’язані з використанням mTLS, у розглянутих рішеннях сервісної сітки є порівняно низькими та залежать від конкретної реалізації, конфігурації й характеру навантаження. За результатами вимірювань, різниця в затримці між окремими рішеннями в типових сценаріях не є критичною для більшості мікросервісних систем. Це дає підстави розглядати mTLS як практично прийнятний механізм захисту міжсервісної взаємодії за умови належного тестування продуктивності в конкретному середовищі.

Водночас у роботі [31] звертається увага на проєктні недоліки окремих реалізацій сервісної сітки, зокрема на недостатній захист площини керування. Тому розгортання сервісної сітки має супроводжуватися окремими заходами безпеки для її керувальних компонентів: ізоляцією в окремому просторі імен,

налаштуванням RBAC, захистом взаємодії через mTLS та обмеженням доступу до адміністративних інтерфейсів.

2.5 API Gateway та захист периметра

API Gateway виконує роль єдиної точки входу для зовнішнього трафіку до мікросервісної системи. На цьому рівні можуть реалізовуватися завершення TLS-з'єднань, автентифікація, обмеження частоти запитів, кешування, маршрутизація та первинний контроль доступу. У термінах NIST SP 800-204 API Gateway можна розглядати як один із перших рубежів захисту мікросервісної архітектури [1].

До типових функцій API Gateway з погляду безпеки належать:

- перевірка токенів доступу OAuth 2.0 на рівні платформи, а не окремо в кожному сервісі;
- перевірка цифрового підпису JWT з використанням JWKS;
- обмеження частоти запитів на основі IP-адреси, токена, користувача або клієнтського застосунку;
- захист від ризиків OWASP API Security Top 10 через інтеграцію з WAF та іншими засобами прикладного захисту;
- облік і виявлення API, зокрема незадокументованих або тіньових кінцевих точок;
- журналювання й аудит усіх вхідних запитів для подальшої кореляції подій у SIEM.

За даними Akamai, 84 % опитаних фахівців повідомили щонайменше про один інцидент безпеки API протягом останнього року, тоді як лише 27 % організацій мають повну інвентаризацію власних API [36]. Це підкреслює важливість API Gateway не лише як механізму маршрутизації, а й як засобу контролю, обліку та виявлення доступних API.

У звіті Traceable AI 2025 State of API Security зазначено, що лише 21 % організацій оцінюють власну здатність виявляти API-атаки як високу, а лише 13 % здатні запобігати більш ніж половині таких атак [35]. Це свідчить про

потребу в посиленні захисту периметра API. У цьому контексті API Gateway у поєднанні з WAF, засобами спостережуваності та механізмами аналізу поведінки може бути одним із першочергових напрямів удосконалення безпеки мікросервісної архітектури.

Таблиця 2.4 — Порівняння поширених рішень API Gateway

Можливість	Kong Gateway	NGINX Plus	Envoy Gateway	AWS API Gateway
OAuth 2.0 / OIDC	Так, через плагіни	Так, через JWT/OIDC-модулі та інтеграції	Так, через політики та фільтри Envoy	Так, через JWT Authorizer / Cognito
JWT validation	Так	Так	Так	Так
Обмеження частоти запитів	Так	Так	Так	Так
Інтеграція з WAF	Так, через плагіни або зовнішні рішення	Так, через NGINX App Protect WAF	Можлива через зовнішні рішення	Так, через AWS WAF
Інвентаризація API	Так	Так	Частково	Так
Mutual TLS	Так	Так	Так	Так

Вибір конкретного API Gateway має ґрунтуватися на оцінці наявного технологічного стеку, середовища розгортання, необхідного набору розширень і операційної моделі команди. Для хмарно-орієнтованого розгортання в Kubernetes доцільними варіантами можуть бути Kong Gateway або Envoy Gateway. Для систем, побудованих переважно на інфраструктурі AWS, природним вибором є AWS API Gateway. Для локальних або гібридних середовищ із різнорідним технологічним стеком може бути доречним використання NGINX Plus. Окремі механізми, розглянуті в попередньому розділі, ефективно розв’язують часткові завдання: mTLS забезпечує конфіденційність і автентичність каналу, SPIFFE — криптографічно підтверджену ідентичність робочих навантажень, OAuth 2.0 і JWT — авторизацію на основі токенів, а OPA — централізоване ухвалення рішень

щодо доступу. Водночас жоден із цих механізмів окремо не дає цілісної відповіді на питання, як спроєктувати безпеку комунікацій у мікросервісній системі на всіх рівнях — від захисту периметра до аналізу інцидентів після їх виникнення. У наступному розділі ці механізми об’єднано у фреймворк , поданий як набір восьми функціональних напрямів забезпечення безпеки. Фреймворк доповнено матрицею відповідності загроз, моделлю зрілості, набором ключових показників ефективності та демонстраційною Java-реалізацією для підтвердження практичної застосовності запропонованої моделі.

РОЗДІЛ 3 ФРЕЙМВОРК ТА РЕКОМЕНДАЦІЇ ДЛЯ ВПРОВАДЖЕННЯ

3.1 Концепція й архітектура фреймворку

Фреймворк, запропонований у межах цієї роботи, поєднує практики забезпечення безпеки комунікацій між мікросервісами в багаторівневу модель, побудовану з урахуванням принципів архітектури нульової довіри. Базовим для фреймворку є принцип NIST SP 800-207, згідно з яким довіра не надається неявно, а має постійно перевірятися та переоцінюватися [6]. На відміну від підходів, що зосереджуються на окремих аспектах захисту, наприклад лише на mTLS або лише на ідентичності сервісів, запропонований фреймворк охоплює повний цикл захисту міжсервісної взаємодії — від точки входу через API Gateway до аналізу інцидентів після їх виникнення. Фреймворк подано як модель архітектурних рішень, що складається з восьми функціональних напрямів забезпечення безпеки. Ці напрями не утворюють жорсткої лінійної послідовності або піраміди, а розглядаються як взаємопов’язані архітектурні рішення. Організація може комбінувати їх відповідно до власної моделі загроз, доступних ресурсів і поточного рівня зрілості.

Такий підхід відрізняється від традиційних багаторівневих моделей тим, що жоден функціональний напрям не розглядається як єдина обов’язкова основа для реалізації всіх інших. Натомість напрями взаємно посилюють один одного, але можуть упроваджуватися поетапно або окремо залежно від потреб конкретної системи.

Рисунок 3.1 — Функціональні напрями фреймворку

Нижче подано стислу характеристику кожного функціонального напрямку із прив’язкою до відповідних стандартів і галузевих практик.

Функціональний напрям 1. Контроль периметра та відкритості API

Цей напрям охоплює контроль зовнішнього трафіку до мікросервісної системи, тобто трафіку типу North–South. Основними засобами реалізації є API Gateway, наприклад Kong, NGINX, Envoy або AWS API Gateway, а також WAF, обмеження частоти запитів і засоби захисту від DDoS-атак. Ключовими функціями цього напряму є централізована автентифікація точок входу API, завершення TLS-з’єднань, захист від ризиків OWASP API Security Top 10 та облік доступних кінцевих точок API. На практичному рівні це може передбачати примусове використання TLS 1.2 або TLS 1.3, перевірку токенів доступу, перевірку цифрового підпису JWT, а також використання списків дозволених або заблокованих IP-адрес.

Функціональний напрям 2. Захищений транспортний рівень TLS/mTLS

Цей напрям передбачає шифрування трафіку між сервісами у виробничому середовищі. Для внутрішнього міжсервісного трафіку, тобто

East–West traffic, доцільним є використання автоматизованого mTLS на рівні sidecar-проксі сервісної сітки, зокрема Istio, Linkerd або Cilium. Такий підхід узгоджується з рекомендаціями NIST SP 800-204A [2] та принципами архітектури нульової довіри [6]. Перевагою цього підходу є те, що шифрування залишається прозорим для коду застосунку й реалізується на інфраструктурному рівні. Другий функціональний напрям розв'язує завдання забезпечення конфіденційності та цілісності комунікаційного каналу. Водночас питання повноцінної перевірки ідентичності сторін потребує додаткових механізмів, які розглядаються в межах третього функціонального напряму.

Функціональний напрям 3. Ідентифікація робочих навантажень із використанням SPIFFE/SPIRE

Цей напрям відповідає за надання робочим навантаженням криптографічно підтвердженої ідентичності, яка не залежить від мережевих ідентифікаторів, таких як IP-адреси або DNS-імена. Замість прив'язки до мережевого розташування кожне робоче навантаження отримує SPIFFE ID та SVID, що дає змогу однозначно ідентифікувати сервіс у межах визначеного домену [10], [11]. Використання короткострокових сертифікатів, строк дії яких може вимірюватися хвилинами або годинами, зменшує можливі наслідки компрометації окремого сервісу. Крім того, федерація SPIFFE дає змогу встановлювати довіру між різними кластерами, середовищами або хмарними провайдерами. На відміну від другого функціонального напряму, який зосереджується на захисті каналу передавання даних, третій напрям уточнює зміст ідентичності сторін взаємодії. Іншими словами, він дає змогу перевірити не лише факт наявності сертифіката, а й те, якому саме сервісу, середовищу або домену відповідає ця ідентичність.

Функціональний напрям 4. Авторизація на основі токенів і сесій із використанням OAuth 2.0, OIDC та JWT

Цей напрям охоплює видачу та перевірку токенів доступу з обмеженим строком дії та визначеною областю дозволів. Такі токени містять твердження, зокрема про видавця, цільову аудиторію, строк дії та дозволені операції, які надалі можуть використовуватися для ухвалення рішень щодо доступу [12], [13], [15]. Для міжсервісної взаємодії типовим сценарієм є Client Credentials Grant, за якого сервіс отримує токен доступу від сервера авторизації без участі користувача. Для взаємодії з користувацькими застосунками доцільно використовувати Authorization Code Grant із PKCE. Важливо, що JWT сам по собі не ухвалює рішення про доступ. Він лише передає набір тверджень (claims), які описують суб'єкта, аудиторію, строк дії токена та дозволені операції. Остаточне рішення про дозвіл або заборону доступу має ухвалюватися на рівні політик доступу, що розглядається в межах п'ятого функціонального напряму.

Функціональний напрям 5. Ухвалення та застосування рішень щодо доступу

Цей напрям відповідає за ухвалення рішень щодо доступу на основі атрибутно-орієнтованої моделі авторизації ABAC. У такій моделі враховуються атрибути суб'єкта, дії, ресурсу та контексту запиту: хто звертається, яку дію виконує, до якого ресурсу намагається отримати доступ і за яких умов це відбувається. Точка застосування політики доступу, наприклад Envoy sidecar, проміжний обробник застосунку або API Gateway, передає запит на ухвалення рішення до окремого компонента — точки ухвалення рішень щодо політик доступу. Таку роль може виконувати Open Policy Agent або інший еквівалентний механізм [24], [25]. NIST SP 800-204B прямо рекомендує використання атрибутно-орієнтованого контролю доступу для мікросервісних систем [3]. Базовим принципом цього функціонального напряму є підхід заборони за замовчуванням: доступ надається лише тоді, коли політика явно дозволяє відповідну дію.

Функціональний напрям 6. Керування життєвим циклом секретів, ключів і сертифікатів

Цей напрям охоплює керування секретами, криптографічними ключами та сертифікатами протягом усього їхнього життєвого циклу: створення, зберігання, видачу, ротацію, відкликання та оновлення. HashiCorp Vault може використовуватися для централізованого керування секретами, зокрема для видачі динамічних секретів і їх автоматичної ротації [26], [27]. У Kubernetes для автоматизованої видачі та оновлення сертифікатів X.509 може застосовуватися cert-manager [28].

Автоматизація цього процесу набуває особливого значення в умовах скорочення строку дії публічних TLS-сертифікатів. У такому середовищі ручне керування сертифікатами стає ризикованим і операційно складним. Окремо має бути організована ротація JWKS, яка узгоджується з ротацією ключів, що використовуються для підпису JWT.

Функціональний напрям 7. Спостережуваність, аудит і телеметрія безпеки

Цей напрям охоплює збирання, кореляцію та аналіз даних про роботу мікросервісної системи й події безпеки. До його складників належать розподілене трасування, зокрема Jaeger, Zipkin або OpenTelemetry, централізоване журналювання за допомогою ELK чи Loki, метрики Prometheus/Grafana, журнали аудиту Kubernetes, а також інтеграція з SIEM-системами.

Поширення ідентифікатора трасування між сервісами дає змогу відновити повний шлях запиту через кілька або навіть десятки мікросервісів. Це важливо не лише для діагностики помилок, а й для аналізу інцидентів безпеки. Сьомий функціональний напрям створює інформаційну основу для восьмого напряму: без належної спостережуваності виявлення інцидентів стає неповним і малоефективним.

Функціональний напрям 8. Виявлення інцидентів, реагування та відновлення

Цей напрям охоплює виявлення інцидентів, реагування на них, відновлення після інцидентів і подальший аналіз причин їх виникнення. Виявлення може здійснюватися за допомогою сповіщень у SIEM, засобів безпеки під час виконання, а також eBPF-інструментів, таких як Falco або Tetragon.

Реагування на інциденти може включати ізоляцію скомпрометованого сервісу, застосування circuit breaker, автоматичне обмеження або блокування підозрілої взаємодії, відкликання сертифікатів і токенів, а також оновлення політик доступу. Окреме значення має цифрове розслідування інцидентів, яке потребує журналювання, придатного для подальшого аналізу, з урахуванням короткого життєвого циклу контейнерів.

Післяінцидентний аналіз має повертати отримані висновки в архітектуру безпеки, зокрема у вигляді нових або уточнених правил політик доступу в межах п’ятого функціонального напряму. Виявлення аномалій на основі машинного навчання може розглядатися як додаткове розширення восьмого напряму на найвищому рівні зрілості.

Безпека ланцюга постачання програмного забезпечення, зокрема підпис образів контейнерів і використання SBOM, у запропонованому фреймворку розглядається як контекст DevSecOps, що підсилює керування секретами, ключами та сертифікатами, але не виділяється в окремий функціональний напрям.

Межі довіри (Trust boundaries): де застосовуються функціональні напрями

Запропонований фреймворк розрізняє три основні класи меж довіри, на яких застосовуються відповідні комбінації функціональних напрямів безпеки.

Такий підхід дає змогу аналізувати захист не в абстрактних рівнях, а в термінах конкретних інтерфейсів і типів взаємодії між компонентами системи.

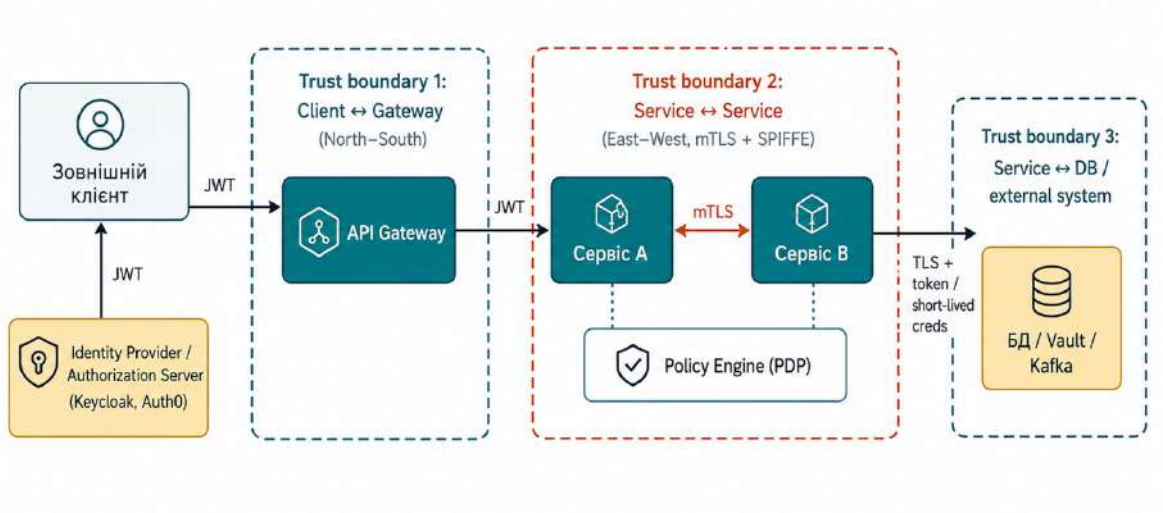

Рисунок 3.2 — Trust boundaries: точки застосування функціональних напрямів фреймворку у мікросервісній системі

Таблиця 3.1 — Trust boundaries та відповідні функціональні напрями фреймворку

Trust boundary	**Типовий трафік**	**Функціональні напрями, що застосовуються**
Клієнт → API Gateway (*North–South*)	Зовнішній HTTP/HTTPS-трафік	C1 — контроль периметра; C4 — авторизація на основі токенів; C7 — аудит і спостережуваність
Сервіс ↔ Сервіс (*East–West*)	Внутрішня взаємодія через HTTP	C2 — TLS/mTLS; C3 — ідентифікація робочих навантажень; C4 — авторизація на основі токенів; C5 — політики доступу

Trust boundary	Типовий трафік	Функціональні напрями, що застосовуються
Сервіс → База даних / зовнішній сервіс	JDBC, Kafka, REST-запити до зовнішніх API	C2 — TLS; C6 — динамічні секрети; C5 — політики вихідного доступу

Зовнішній постачальник ідентичності або сервер авторизації, наприклад Keycloak чи Auth0, видає JWT поза межами фреймворку і передає його клієнтові. Фреймворк не виконує функції постачальника ідентичності, а лише перевіряє отримані токени та використовує їхні твердження (claims) для подальшого контролю доступу. Це дає змогу чітко розмежувати відповідальність: видача токенів належить до окремого компонента архітектури, тоді як фреймворк відповідає за їх перевірку та застосування в межах захисту комунікацій.

Атрибути запиту в моделі ABAC

У запропонованому фреймворку рішення щодо доступу ухвалюються на основі атрибутно-орієнтованого контролю доступу ABAC [3]. Для цього запит до механізму політик описується через чотири групи атрибутів: суб’єкт, дія, ресурс і контекст. Суб’єкт визначає, хто ініціює запит, наприклад користувач, сервіс або робоче навантаження. Дія описує операцію, яку потрібно виконати, наприклад читання, запис або публікацію повідомлення. Ресурс визначає об’єкт доступу, наприклад замовлення, платіж або топік повідомлень. Контекст охоплює умови виконання запиту, зокрема простір імен, рівень ризику, час або IP-адресу.

Такий підхід дає змогу створювати політики доступу, які враховують не лише роль або ідентичність суб’єкта, а й поточні умови виконання запиту. Наприклад, політика може дозволяти читання даних у виробничому середовищі лише в робочі години або забороняти публікацію повідомлень у топік замовлень для сервісів, що не належать до простору імен orders.

Таблиця 3.3 — Групи атрибутів ABAC-запиту

Група атрибутів	Що описує	Приклади
Суб’єкт	Хто ініціює запит. У Java-реалізації SecurityContext. serviceIdentity	spiffe://prod/ns/orders/sa/orders-service, sub=svc-orders
Дія	Яку операцію потрібно виконати. У Java-реалізації це AccessRequest.action	read, write, publish, delete
Ресурс	До якого об’єкта або сервісу здійснюється доступ. У Java-реалізації це AccessRequest.resource	billing-invoice, kafka-topic:orders, payment
Контекст	За яких умов виконується запит. У Java-реалізації це AccessRequest. contextAttributes	namespace=prod, riskLevel=normal, time=09:00–18:00

Асинхронні комунікації: Kafka, RabbitMQ, NATS

Окрім синхронної взаємодії через HTTP, фреймворк може застосовуватися і до асинхронних комунікацій, що здійснюються через брокери повідомлень, зокрема Kafka, RabbitMQ або NATS. У цьому випадку базові принципи безпеки залишаються незмінними, однак конкретні механізми їх реалізації відрізняються від підходів, характерних для HTTP-взаємодії.

Таблиця 3.4 — Застосування функціональних напрямів фреймворку до асинхронної взаємодії

Capability	Адаптація для асинхронних брокерів
C2 — Захищений транспортний рівень	Використання TLS для з’єднань між виробниками повідомлень, споживачами та брокером, а також для взаємодії між брокерами в межах кластера.
C3 — Ідентифікація	Надання робочому навантаженню ідентичності у форматі, який підтримується конкретним брокером: SASL/SCRAM-облікові дані, mTLS-

Capability	Адаптація для асинхронних брокерів
робочих навантажень	сертифікат або IAM-ідентичність для керованих хмарних сервісів Kafka.
C4 — Авторизація на основі токенів	Використання OAuth 2.0 через механізм SASL/OAUTHBEARER у Kafka або токен-орієнтованих розширень у RabbitMQ. Токен доступу може містити область дозволів для конкретних топіків, черг або subject-ів.
C5 — Політики доступу та ACL	Застосування ACL на рівні топіків у Kafka, дозволів для exchange і queue у RabbitMQ, а також правил авторизації в NATS. Механізм політик визначає права на публікацію, споживання та адміністративні дії для кожного ресурсу.
C6 — Керування секретами	Ротація SASL/SCRAM-облікових даних через Vault, автоматизована видача й оновлення mTLS-сертифікатів для брокерів, а також централізоване зберігання секретів доступу.
C7 — Аудит і спостережуваність	Журналювання операцій публікації та споживання повідомлень, аудит адміністративних дій, передавання подій до SIEM, а також створення незмінної копії журналів для подальшого аналізу.

Для виробників і споживачів повідомлень застосовується та сама логіка контролю доступу, що й для синхронних HTTP-запитів. Спочатку виконується автентифікація суб’єкта на основі ідентичності робочого навантаження, далі перевіряються твердження (claims) токена, а остаточне рішення щодо дозволу на публікацію або споживання повідомлень із конкретного топіка ухвалюється механізмом політик доступу.

3.2 Policy-as-Code та управління секретами

OPA (Open Policy Agent) — це універсальний механізм ухвалення рішень щодо політик доступу. Він отримує на вхід структурований документ input і повертає рішення на основі політики, описаної мовою Rego [24]. У

мікросервісному середовищі OPA може розгортатися як sidecar-компонент поруч із сервісом або як централізований сервіс ухвалення рішень, до якого Envoy звертається через ext_authz API [25].

Рисунок 3.3 — OPA як Policy Decision Point у мікросервісах

До переваг підходу політики доступу у вигляді коду»належать декларативність, можливість версіонування політик у Git, їх автоматизоване тестування в CI/CD-процесах, централізоване журналювання ухвалених рішень, а також розділення відповідальності між розробниками політик доступу та розробниками сервісів.

До типових категорій політик, які доцільно винести в OPA, належать:

- політики міжсервісної авторизації — визначають, який сервіс має право звертатися до конкретної кінцевої точки іншого сервісу;
- політики авторизації на рівні об’єктів — знижують ризик BOLA, тобто порушення авторизації на рівні об’єктів, через перевірку відповідності твердження sub ідентифікатору ресурсу;
- політики допуску в Kubernetes — через OPA Gatekeeper можуть забороняти привілейовані контейнери, вимагати обов’язкові мітки та перевіряти наявність мережевих політик;
- політики CI/CD-процесів — можуть перевіряти SBOM, забороняти розгортання образів без підпису cosign або застосовувати інші вимоги до ланцюга постачання програмного забезпечення.

HashiCorp Vault забезпечує централізоване зберігання секретів і підтримує динамічні секрети, тобто облікові дані, які видаються за запитом із обмеженим строком дії [26]. У Kubernetes інтеграція з Vault може здійснюватися за допомогою Vault Secrets Operator, CSI-монтування або Vault Agent Injector, що дає змогу доставляти секрети до застосунків без їхнього жорсткого збереження в образах контейнерів чи конфігураційних файлах [27].

Таблиця 3.5 — Порівняння підходів до керування секретами

Параметр	**Статичні облікові дані**	**Динамічні секрети Vault**
Спосіб видачі	Попередньо задані в конфігурації, Kubernetes Secrets або змінних середовища	Видаються за запитом з обмеженим строком дії
Ротація	Здебільшого ручна та нерегулярна	Автоматизована, з періодичністю від кількох годин до кількох днів
Відкликання	Ускладнене, часто потребує оновлення конфігурації або повторного розгортання сервісу	Може виконуватися централізовано через Vault
Аудит	Обмежений або фрагментарний	Централізований журнал доступу до секретів
Наслідки компрометації	Значні, оскільки секрет залишається чинним до ручної заміни	Обмежені завдяки короткому строку дії секрету
Інтеграція з CI/CD	Часто через змінні середовища або секрети платформи	Через короткострокові токени та автоматизовану видачу секретів

cert-manager — це контролер для Kubernetes, який автоматизує видачу, оновлення та видалення сертифікатів X.509. Він може інтегруватися з

різними джерелами сертифікатів, зокрема ACME-серверами, такими як Let’s Encrypt або Buypass, HashiCorp Vault PKI, самопідписаними центрами сертифікації та внутрішньою PKI організації [28]. У поєднанні з Istio та Vault PKI cert-manager дає змогу побудувати автоматизовану інфраструктуру керування сертифікатами для сервісної сітки.

3.3 Спостережуваність та реагування на інциденти

Розподілене трасування є одним із ключових інструментів для відстеження шляху запиту через множину мікросервісів. Стандарт OpenTelemetry дає змогу уніфікувати збирання трас, метрик і журналів подій [40]. У середовищах із сервісною сіткою поширення ідентифікатора трасування між сервісами може забезпечуватися на рівні проксі, зокрема Envoy. Водночас ручне додавання засобів трасування може бути потрібне для фіксації бізнес-подій або операцій, специфічних для конкретного застосунку.

Журнали аудиту Kubernetes фіксують звернення до Kubernetes API, зокрема операції створення, оновлення, видалення та часткової зміни ресурсів. Такі записи містять контекст виконання операції: суб’єкта, ресурс, дію та відповідь API-сервера. Політика аудиту дає змогу налаштовувати рівень деталізації журналювання для різних класів ресурсів.

Для виявлення аномалій у потоці метрик і подій безпеки доцільно використовувати SIEM-системи, зокрема з компонентами аналізу поведінки користувачів і сутностей. Для захисту контейнерів під час виконання ефективними є рішення на основі eBPF, наприклад Falco або Cilium Tetragon [34].

Таблиця 3.6 — Стек спостережуваності у фреймворку

Шар	Стандарт / інструмент	Призначення
Розподілене трасування	OpenTelemetry, Jaeger, Zipkin	Відстеження шляху запиту через мікросервіси
Метрики	Prometheus, Grafana	Моніторинг показників продуктивності та безпеки: затримки, частки помилок,

Шар	Стандарт / інструмент	Призначення
		стану mTLS, доступності сервісів
Журнали подій	Loki, Elasticsearch, Fluent Bit	Збирання журналів застосунків, sidecar-проксі та інфраструктурних компонентів
Журнали аудиту	Kubernetes Audit Logs, OPA Decision Logs	Фіксація дій у кластері та рішень політик доступу: хто, що, коли і щодо якого ресурсу виконав
Безпека під час виконання	Falco, Tetragon, eBPF	Виявлення підозрілих системних викликів, поведінкових аномалій і порушень у контейнерах
SIEM/SOAR	Splunk, Elastic Security, OpenSearch	Кореляція подій безпеки, формування сповіщень і запуск сценаріїв реагування

Реагування на інциденти в мікросервісному середовищі ускладнюється коротким життєвим циклом контейнерів. Через це артефакти, необхідні для подальшого аналізу інцидентів, зокрема журнали подій, траси та знімки стану, мають зберігатися поза межами контейнера в централізованих сховищах. Для таких сховищ доцільно використовувати режим незмінного зберігання, наприклад підхід write-once або механізми на кшталт S3 Object Lock чи їхні аналоги.

3.4 Модель зрілості й ключові показники ефективності

Рисунок 3.4 — Чотирирівнева модель зрілості впровадження фреймворку

Запропонована чотирирівнева модель зрілості (M1–M4) дозволяє організаціям упроваджувати фреймворк поетапно, з урахуванням наявної інфраструктури, ресурсів команди та поточної моделі загроз:

1. M1 — базовий рівень. Передбачає використання TLS на периметрі, базову автентифікацію на рівні API Gateway, Kubernetes RBAC та базові мережеві політики.
2. M2 — проміжний рівень. Охоплює впровадження mTLS у сервісній сітці для внутрішнього міжсервісного трафіку, використання JWT/OAuth 2.0 для міжсервісної авторизації, застосування HashiCorp Vault для керування секретами та cert-manager для автоматизованої ротації сертифікатів.
3. M3 — розвинутий рівень. Передбачає використання SPIFFE/SPIRE для ідентифікації робочих навантажень, застосування політик доступу у вигляді коду за допомогою OPA, повне розподілене трасування, журнали аудиту Kubernetes та автоматизовані сценарії реагування на інциденти.
4. M4 — оптимізований рівень. Охоплює федерацію довіри між кластерами, безперервну перевірку політик у CI/CD-процесах,

виявлення аномалій на основі машинного навчання та автоматизовану підготовку даних для цифрового розслідування інцидентів.

Таблиця 3.7 — Відповідність загроз STRIDE функціональним напрямам фреймворку і механізмам захисту

Загроза	STRIDE	Функціональні напрями	Механізм захисту
Підроблення ідентичності сервісу	Spoofing	C2, C3	mTLS, SPIFFE ID, SVID
Перехоплення трафіку	Information Disclosure	C2	Шифрування трафіку за допомогою TLS/mTLS
Несанкціонована зміна повідомлень або трафіку	Tampering	C2, C5	mTLS, перевірка цілісності повідомлень, політики доступу
Витік даних або секретів	Information Disclosure	C2, C6	mTLS, централізоване керування секретами, шифрування секретів у Vault
Повторне використання токена	Spoofing	C4	Короткий строк дії JWT, перевірка exp і jti, механізми відкликання токенів
Несанкціонований доступ до API	Несанкціонований доступ до API	C1, C4, C5	Автентифікація на рівні API Gateway, перевірка JWT, scopes, OPA/ABAC

Загроза	STRIDE	Функціональні напрями	Механізм захисту
DDoS або виснаження ресурсів	Denial of Service	C1	Обмеження частоти запитів, WAF, квоти, масштабування
Експлуатація незадокументованих API	Information Disclosure / Elevation of Privilege	C1, C7	Облік API, виявлення тіньових кінцевих точок, аудит, аналіз поведінки
Витік секретів	Information Disclosure	C6	Динамічні секрети Vault, ротація секретів, обмеження строку дії
Атака на ланцюг постачання ПЗ	Tampering	C6, C8	SBOM, підпис образів, Sigstore/cosign, перевірка артефактів у CI/CD
Підвищення привілеїв	Elevation of Privilege	C3, C5	RBAC/ABAC, ідентичність робочих навантажень, політики доступу
Порушення авторизації на рівні об’єктів	Elevation of Privilege	C4, C5	Перевірка тверджень JWT, OPA/ABAC, перевірка доступу до конкретного об’єкта

Таблиця 3.8 — Ключові показники ефективності

KPI	Базова ціль (M2)	Оптимальна ціль (M4)
Частка сервісів з активним mTLS	> 80 %	100 %
Середній час ротації сертифікатів	< 30 днів	автоматизовано, < 24 год
Частка секретів, що зберігаються в централізованому сховищі	> 90 %	100 %
Час виявлення аномалій, MTTD	< 60 хв	< 15 хв
Час реагування на інциденти, MTTR	< 4 год	< 1 год
Частка кінцевих точок API, внесених до інвентарю	> 95 %	100 %
Частка політик доступу, що зберігаються в системі контролю версій	> 80 %	100 %
Частка робочих навантажень зі SPIFFE-ідентичністю	не вимагається	100 %
Частка сервісів із розподіленим трасуванням	> 70 %	100 %

3.5 Практичні сценарії застосування фреймворку

Сценарій 1. Перехід від монолітної архітектури до мікросервісної в новому проєкті (greenfield)

Для нового проєкту доцільно одразу орієнтуватися на рівень зрілості M2, що передбачає використання сервісної сітки, cert-manager і Vault. Такий підхід дає змогу сформувати надійну основу для захисту міжсервісної взаємодії та зменшити ризик накопичення технічного боргу. Авторизацію варто будувати на основі OAuth 2.0 з початкових етапів проєктування.

Орієнтовна дорожня карта впровадження може бути такою:

1. Розгорнути Kubernetes-кластер із налаштованим RBAC, базовими мережевими політиками та ввімкненим аудитом.
2. Обрати сервісну сітку, наприклад Istio або Linkerd, і спочатку розгорнути mTLS у режимі PERMISSIVE для безпечного перехідного етапу.
3. Впровадити cert-manager і Vault PKI для автоматизованої видачі, оновлення та керування сертифікатами.
4. Поступово перевести міжсервісну взаємодію до режиму STRICT mTLS після перевірки сумісності сервісів.
5. Розгорнути сервер авторизації, наприклад Keycloak, і перевести міжсервісну авторизацію на сценарій OAuth 2.0 Client Credentials.
6. Впровадити API Gateway із засобами захисту від основних ризиків OWASP API Security Top 10 та сформувати актуальний перелік доступних API.
7. Налаштувати розподілене трасування за допомогою OpenTelemetry для основних сервісів.
8. Підключити базову SIEM-систему та підготувати інструкції реагування на критичні інциденти.

Сценарій 2. Зрілий мікросервісний застосунок (brownfield)

Для вже наявного мікросервісного застосунку основним викликом є впровадження додаткових механізмів безпеки без значних простоїв і без порушення сумісності зі сторонніми клієнтами. У такому випадку доцільно застосовувати поетапний підхід, за якого сервісна сітка вмикається

поступово для окремих просторів імен після попереднього тестування в staging-середовищі.

На початковому етапі варто розгорнути сервісну сітку лише для обмеженої групи сервісів, які мають найменше зовнішніх залежностей. Після перевірки коректності маршрутизації, затримок, журналювання та політик доступу можна поступово розширювати охоплення на інші простори імен.

Окрему увагу необхідно приділити застарілим клієнтам або стороннім інтеграціям, які ще не підтримують mTLS. Для них на перехідному етапі може застосовуватися режим PERMISSIVE, що дозволяє приймати як зашифрований, так і незашифрований трафік. Надалі такі клієнти слід поступово мігрувати до обов'язкового використання mTLS, контролюючи метрики взаємної автентифікації, частку незашифрованих з'єднань і кількість відхилених запитів.

Сценарій 3. DevSecOps інтеграція

NIST SP 800-204C і NIST SP 800-204D описують застосування контрольних точок безпеки в CI/CD-процесах, що дає змогу виявляти ризики ще до розгортання сервісів у виробничому середовищі [4], [5].

Для інтеграції фреймворку у DevSecOps-процеси доцільно використовувати послідовний набір перевірок: статичний аналіз коду, перевірку залежностей, сканування образів контейнерів, підпис образів за допомогою Sigstore або cosign, формування SBOM за допомогою Syft, перевірку політик через OPA і conftest, а також контроль допуску ресурсів у Kubernetes за допомогою OPA Gatekeeper.

Такий підхід дає змогу поєднати безпеку комунікацій із безпекою ланцюга постачання програмного забезпечення. У результаті політики фреймворку застосовуються не лише під час виконання сервісів, а й на етапах розроблення, перевірки, збирання та розгортання.

Сценарій 4. FinOps та оцінка вартості впровадження

Впровадження фреймворку потребує попереднього оцінювання витрат, пов’язаних із ліцензіями, інфраструктурними ресурсами та роботою команди. До можливих статей витрат належать використання комерційних редакцій або підтримуваних дистрибутивів окремих інструментів, додаткові обчислювальні ресурси для sidecar-проксі, OPA, SPIRE Agent та компонентів площини керування, а також витрати на роботу DevSecOps- і SRE-команд.

У середовищах із великою кількістю сервісів сервісна сітка може створювати додаткове навантаження на CPU та пам’ять кластера через використання sidecar-компонентів і керувальних сервісів. Тому перед промисловим упровадженням доцільно провести навантажувальне тестування та оцінити вплив обраної архітектури на продуктивність. Частину цих накладних витрат можна зменшити завдяки використанню підходів без sidecar-компонентів, зокрема Cilium Service Mesh на основі eBPF або Istio Ambient.

Орієнтовна структура витрат може охоплювати три основні категорії: людський ресурс на впровадження й подальшу експлуатацію, інфраструктурні витрати, а також навчання команди й зовнішню експертизу. Економічний ефект від упровадження фреймворку може проявлятися через скорочення середнього часу реагування на інциденти, зменшення ризику витоків даних, підвищення керованості доступу та зниження операційних витрат на ручне керування сертифікатами, секретами й політиками безпеки.

Узагальнення практичних рекомендацій

На підставі викладеного в цьому розділі можна сформулювати такі узагальнені практичні рекомендації:

1. Розпочинати впровадження захисту комунікацій із API Gateway та OAuth 2.0, оскільки ці механізми дають змогу швидко посилити контроль доступу до зовнішніх точок входу API.

2. Якомога раніше переходити до використання сервісної сітки з режимом STRICT mTLS, оскільки впровадження взаємної автентифікації на ранніх етапах є простішим, ніж подальша міграція вже сформованого застарілого середовища.
3. Описувати політики безпеки у вигляді коду з використанням Git, OPA та автоматизованих перевірок у CI/CD-процесах. Це підвищує відстежуваність змін, відтворюваність конфігурацій і зменшує ризик інцидентів, спричинених помилками налаштування.
4. Використовувати динамічні секрети та короткострокові сертифікати, оскільки це зменшує можливі наслідки компрометації окремого секрету або сервісу.
5. Інтегрувати засоби спостережуваності з початкових етапів упровадження: поширення ідентифікатора трасування між сервісами, журнали аудиту, метрики взаємної автентифікації та централізоване збирання подій безпеки. Без цього ускладнюються як налагодження системи, так і реагування на інциденти.
6. Підтримувати актуальний перелік API та регулярно зіставляти його з фактичним трафіком. Це дає змогу виявляти незадокументовані або тіньові API, які залишаються одним із суттєвих джерел ризику.
7. Регулярно виконувати моделювання загроз за моделлю STRIDE з використанням матриці відповідності, наведеної в таблиці 3.7.
8. Відстежувати ключові показники ефективності з таблиці 3.8 під час регулярного перегляду операційного стану системи та використовувати ці дані для планування подальшого розвитку безпеки.

Концептуальна модель фреймворку, розглянута в підрозділах 3.1–3.5, охоплює вісім функціональних напрямів забезпечення безпеки, матрицю відповідності загроз STRIDE, модель зрілості, ключові показники ефективності та практичні сценарії впровадження. Вона описує архітектурні рішення на концептуальному рівні. Для переходу від архітектурної моделі до

виконуваних інженерних артефактів у наступному підрозділі подано демонстраційну Java-реалізацію фреймворку

Java-проєкт ілюструє програмні контракти між функціональними напрямами фреймворку у вигляді інтерфейсів і класів, демонструє роботу на типових сценаріях авторизації та містить порівняння між можливостями, реалізованими в proof-of-concept, і вимогами, необхідними для промислового середовища.

3.6 Реалізація Java-фреймворку

Для переходу від концептуальної моделі фреймворку до виконуваного інженерного артефакту розроблено демонстраційну Java-бібліотеку MSSCF Java Framework. Вибір Java як платформи реалізації зумовлений її поширеністю в промислових мікросервісних системах, зокрема в екосистемах Spring Boot, Quarkus і Micronaut, зрілістю стандартних механізмів безпеки платформи, таких як java.security, JAAS і JCE, а також можливістю реалізувати основну логіку фреймворку на базі стандартної бібліотеки Java 17 без зовнішніх залежностей.

3.6.1 Архітектура та структура модулів

Фреймворк організовано як Maven-проєкт з ідентифікатором групи ua.edu.msscf та ідентифікатором артефакту msscf-java-framework. Пакетна структура проєкту узгоджена з концептуальними напрямами фреймворку, що дає змогу простежити зв’язок між архітектурною моделлю та її програмною реалізацією. Кожен пакет виконує роль окремого модуля з відкритим інтерфейсом, демонстраційною реалізацією та можливістю подальшої заміни на промисловий варіант. Наприклад, in-memory постачальник секретів може бути замінений клієнтом до HashiCorp Vault, а консольний журнал аудиту — інтеграцією з Kafka або централізованою системою журналювання.

Рисунок 3.5 — Архітектура Java-фреймворку: пакети, ключові класи та потік обробки запиту

Таблиця 3.10 — Відповідність функціональних напрямків модулям Java-фреймворку

Рівень	Концептуальна роль	Java-пакет	Ключові класи і інтерфейси
1	Perimeter Security & API Gateway	ua.edu.msscf.gateway	CommunicationSecurityFilter, MsscfGatewayFilter
2	Service-to-Service mTLS	ua.edu.msscf.gateway (filter chain)	MsscfGatewayFilter (перевірка mTLS як фільтр)
3	Workload Identity (SPIFFE)	ua.edu.msscf.identity	WorkloadIdentity
4	Token-based AuthZ (OAuth 2.0/JWT)	ua.edu.msscf.identity	AccessToken, TokenValidator
5	Policy-as-Code (PDP)	ua.edu.msscf.policy	PolicyEngine, MsscfPolicyEngine, Rule, PolicyDecision
6	Secrets Lifecycle Management	ua.edu.msscf.secrets	SecretProvider, InMemorySecretProvider, DynamicSecret
7	Observability & Audit	ua.edu.msscf.audit	AuditLogger, AuditEvent, ConsoleAuditLogger
8	Incident Response & Forensics	ua.edu.msscf.audit + integrators	AuditLogger (append-only) + зовнішні SIEM/runbooks

Рівень	Концептуальна роль	Java-пакет	Ключові класи і інтерфейси
—	Корінь фреймворку	ua.edu.msscf.core	MsscfFramework, MsscfConfig, SecurityContext
—	Демо та тести	ua.edu.msscf.demo	DemoApplication, MsscfFrameworkTest

Свідомим архітектурним рішенням є відсутність зовнішніх залежностей під час виконання. Це спрощує повторне збирання проєкту, зменшує поверхню атаки, пов’язану з ланцюгом постачання програмного забезпечення, і дає змогу інтегрувати MSSCF у різні Java-стеки без ризику конфліктів версій сторонніх бібліотек. Для промислового використання криптографічних механізмів, зокрема підпису JWT або роботи з X.509-SVID, демонстраційні реалізації можуть бути замінені на рішення, побудовані на базі java.security або спеціалізованих бібліотек, таких як Nimbus JOSE + JWT.

3.6.2 Потік обробки запиту

Центральним елементом реалізації є клас MsscfFramework, який надає єдиний публічний метод для перевірки доступу. Обробка запиту відбувається за визначеною послідовністю: спочатку виконуються фільтри безпеки комунікації CommunicationSecurityFilter, що відповідають за перевірки на рівнях 1, 3 і 4; далі запит передається до PolicyEngine для ухвалення рішення щодо доступу на рівні 5; після цього формується та зберігається AuditEvent, який відповідає за аудит на рівні 7.

Псевдокод основного методу authorize:

```
public PolicyDecision authorize(SecurityContext context) {
    // Рівні 1–4: послідовність фільтрів.
    for (CommunicationSecurityFilter filter : config.getFilters()) {
        FilterResult result = filter.apply(context);
        if (!result.pass()) {
            PolicyDecision denied = PolicyDecision.deny(
                    filter.name(), result.reason());
            writeAudit(context, denied);
            return denied;
        }
    }
```

```
    // Рівень 5: оцінка policy.
    PolicyDecision decision = config.getPolicyEngine().evaluate(context);
    writeAudit(context, decision);  // Рівень 7
    return decision;
}
```

У реалізації застосовано принципи швидкої відмови та заборони за замовчуванням. Принцип швидкої відмови означає, що обробка запиту припиняється одразу після виявлення першої критичної помилки, наприклад недійсного токена, ненадійного видавця або відсутньої ідентичності сервісу. Принцип заборони за замовчуванням передбачає, що доступ не надається, якщо жодне правило політики явно не дозволяє відповідну дію. Такий підхід узгоджується з принципами архітектури нульової довіри, викладеними в NIST SP 800-207, а також із рекомендаціями NIST SP 800-204B щодо контролю доступу в мікросервісних системах.

3.6.3 Конфігурація фреймворку

Конфігурація фреймворку задається декларативно за допомогою MsscfConfig.Builder. Через цей об’єкт визначаються домен, перелік довірених видавців токенів, очікувана аудиторія токена, механізм ухвалення рішень щодо політик доступу, сховище секретів і компонент аудиторського журналювання:

```
MsscfPolicyEngine pdp = new MsscfPolicyEngine()
    .addRule(MsscfPolicyEngine.Rule.allowWhen(
        "orders-can-read-billing",
        ctx -> "read".equals(ctx.getAction())
            && ctx.getAccessToken()
                  .map(t -> t.hasScope("billing:read"))
                  .orElse(false),
        "orders-service has billing:read scope"))
    .addRule(MsscfPolicyEngine.Rule.denyWhen(
        "no-external-write-to-billing",
        ctx -> "billing-invoice".equals(ctx.getResource())
            && "write".equals(ctx.getAction()),
        "external write to billing not permitted"));

MsscfConfig config = MsscfConfig.builder()
    .trustDomain("prod.example.com")
    .trustedIssuers(Set.of("https://auth.example.com"))
    .expectedAudience("svc-billing")
    .policyEngine(pdp)
    .secretProvider(new InMemorySecretProvider())
    .auditLogger(new ConsoleAuditLogger())
```

```
    .buildWithDefaults();

MsscfFramework framework = new MsscfFramework(config);
```

Правила PolicyEngine подано як пари з умови перевірки та фабрики рішення PolicyDecision. За своєю логікою цей підхід наближений до політик Rego в OPA: правила перевіряються послідовно, а перше правило, умова якого виконується, визначає результат. Якщо жодне правило не спрацювало, застосовується принцип заборони за замовчуванням, тобто доступ не надається.

3.6.4 Приклад виклику та формування SecurityContext

Для кожного вхідного запиту формується незмінний об’єкт SecurityContext, який містить ідентичність робочого навантаження, токен доступу та бізнес-атрибути, необхідні для ухвалення рішення щодо доступу. Нижче наведено приклад синхронної перевірки доступу до ресурсу billing-invoice:

```
SecurityContext ctx = SecurityContext.builder()
    .requestId("req-001")
    .callerIdentity(WorkloadIdentity.of(
        "prod.example.com", "orders", "orders-service",
        Instant.now().minusSeconds(60),
        Instant.now().plusSeconds(3600)))
    .accessToken(AccessToken.builder()
        .issuer("https://auth.example.com")
        .subject("svc-orders")
        .audience("svc-billing")
        .issuedAt(Instant.now())
        .expiresAt(Instant.now().plusSeconds(600))
        .tokenId(UUID.randomUUID().toString())
        .scopes(Set.of("billing:read"))
        .build())
    .resource("billing-invoice")
    .action("read")
    .build();

PolicyDecision decision = framework.authorize(ctx);
if (decision.isAllowed()) {
    // дозволити доступ до ресурсу
} else {
    throw new ForbiddenException(decision.getReason());
}
```

3.6.5 Результати виконання DemoApplication

Демонстраційний застосунок виконує чотири сценарії перевірки доступу та фіксує кожне ухвалене рішення в журналі аудиту. Нижче наведено скорочений фрагмент фактичного виводу програми:

```
[AUDIT] {"requestId":"req-001","effect":"ALLOW",
         "policyId":"orders-can-read-billing",
         "reason":"orders-service has billing:read scope"}
--- Сценарій 1 (валідний read) ---
Decision: ALLOW(orders-can-read-billing): ...

[AUDIT] {"requestId":"req-002","effect":"DENY",
         "policyId":"no-external-write-to-billing", ...}
--- Сценарій 2 (write — заборонено policy) ---
Decision: DENY(no-external-write-to-billing): ...

[AUDIT] {"requestId":"req-003","effect":"DENY",
         "policyId":"msscf-gateway-filter",
         "reason":"token rejected: untrusted issuer: ..."}

[AUDIT] {"requestId":"req-004","effect":"DENY",
         "reason":"token rejected: token expired at ..."}

Level 6 — видано dynamic secret: role=db-readonly, ttl=15m
Усього зафіксовано audit-подій: 4
```

Крім того, клас MsscfFrameworkTest виконує чотири модульні тести: testValidReadAllowed, testUntrustedIssuerRejected, testExpiredTokenRejected і testDefaultDeny. Вони перевіряють базову логіку фреймворку для чотирьох типових сценаріїв: дозволений коректний запит, відхилення токена від недовіреного видавця, відхилення простроченого токена та заборона доступу за замовчуванням. Тести реалізовано без використання зовнішніх фреймворків тестування, що узгоджується з обраним підходом мінімізації зовнішніх залежностей.

Таблиця 3.11 — Типові демонстраційні сценарії фреймворку

Сценарій	**Очікуваний результат**	**Capability, що визначає рішення**
1. Коректний запит	ALLOW із зазначенням policyId	C5 — механізм політик доступу: спрацювало правило дозволу
2. Недовірений видавець токена	DENY на рівні gateway-фільтра	C4 — перевірка токена: видавець токена відсутній у переліку довірених видавців

Сценарій	Очікуваний результат	Capability, що визначає рішення
3. Прострочений токен	DENY на рівні gateway-фільтра	C4 — перевірка токена: значення exp вказує на завершення строку дії токена
4. Заборона на рівні політики	DENY на рівні PolicyEngine	C5 — механізм політик доступу: спрацювало правило заборони

3.6.6 Інтеграція у Spring Boot та Kubernetes

Фреймворк не залежить від конкретного HTTP-фреймворку, тому може бути інтегрований у різні Java-стеки. Для Spring Boot типовим варіантом інтеграції є використання OncePerRequestFilter. Такий фільтр може отримувати JWT із заголовка Authorization, а ідентичність робочого навантаження — із mTLS-сертифіката або через SPIFFE Workload API:

```
@Component
public class MsscfSecurityFilter extends OncePerRequestFilter {
    private final MsscfFramework framework;

    public MsscfSecurityFilter(MsscfFramework framework) {
        this.framework = framework;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest req,
                                    HttpServletResponse res,
                                    FilterChain chain) throws IOException {
        SecurityContext ctx = SecurityContextMapper.fromRequest(req);
        PolicyDecision decision = framework.authorize(ctx);
        if (!decision.isAllowed()) {
            res.sendError(HttpServletResponse.SC_FORBIDDEN,
                          decision.getReason());
            return;
        }
        chain.doFilter(req, res);
    }
}
```

У Kubernetes-середовищі SecurityContextMapper може формувати SPIFFE ID на основі projected service account token або отримувати SVID через SPIFFE Workload API [10], [17]. Правила PolicyEngine можуть зберігатися в зовнішніх ConfigMap або бути винесені в окрему реалізацію

механізму політик доступу, яка звертається до OPA через його API [24]. Компонент SecretProvider у промисловому середовищі може бути замінений клієнтом HashiCorp Vault, що взаємодіє зі сховищем секретів через відповідний HTTP API [26], [27].

3.6.7 Безпекові обмеження референсної реалізації

Наведений Java-фреймворк є академічною демонстраційною реалізацією, призначеною для підтвердження концепції фреймворку. Тому він має низку свідомих обмежень, пов’язаних із рівнем деталізації окремих модулів.

Компонент TokenValidator перевіряє лише основні твердження токена, зокрема iss, aud та exp, але не виконує криптографічну перевірку підпису JWS. У промисловому середовищі така перевірка має бути обов’язково доповнена валідацією через JWKS із використанням java.security або спеціалізованих бібліотек, наприклад Nimbus JOSE + JWT.

Компонент WorkloadIdentity зберігає лише метадані ідентичності робочого навантаження, тоді як видача та перевірка X.509-SVID у межах демонстраційної реалізації не виконуються. Компонент InMemorySecretProvider зберігає секрети в оперативній пам’яті, що є прийнятним для тестування, але в промисловому середовищі має бути замінено інтеграцією з HashiCorp Vault або іншим захищеним сховищем секретів. Аналогічно, ConsoleAuditLogger записує події аудиту у стандартний потік виводу, тоді як для промислового використання доцільною є інтеграція з Kafka, SIEM або централізованою системою журналювання.

Зазначені обмеження не змінюють архітектурну логіку фреймворку, а лише визначають межі демонстраційної реалізації. Вони показують, які компоненти мають бути замінені або розширені під час переходу від proof-of-concept до промислового впровадження.

3.6.8 Порівняння демонстраційної та промислової реалізації

Щоб уникнути неправильної інтерпретації демонстраційної реалізації, у таблиці нижче узагальнено, які можливості вже реалізовано в поточній версії Java-фреймворку, а які компоненти потребують доопрацювання для використання в промисловому середовищі. Це дає змогу чітко визначити межі застосовності розробленого артефакту та відокремити перевірку архітектурної концепції від вимог до повноцінного виробничого впровадження.

Таблиця 3.12 — Порівняння демонстраційної Java-реалізації з вимогами до промислового впровадження

Компонент / Функціональний напрям	**У PoC реалізовано**	**Потрібно для промислового впровадження**
mTLS, C2	Моделювання контексту запиту та перевірка наявності ідентичності робочого навантаження за принципом швидкої відмови	Сервісна сітка, наприклад Istio, Linkerd або Cilium, або завершення TLS-з'єднань на рівні API Gateway з використанням реальних сертифікатів
SPIFFE-ідентичність, C3	WorkloadIdentity як об'єкт значення з перевіркою строку дії	Клієнт SPIFFE Workload API, видача X.509-SVID, SPIRE Agent на вузлах кластера
JWT, C4	Перевірка тверджень iss, aud та exp	Повна перевірка підпису JWS через JWKS, ротація ключів, використання спеціалізованої бібліотеки, наприклад Nimbus JOSE + JWT
Механізм політик доступу, C5	Java-реалізація PDP на основі Predicate, яка моделює логіку послідовного виконання правил	OPA/Rego або зовнішній PDP, завантаження правил із ConfigMap чи іншого централізованого джерела конфігурації
Секрети, C6	InMemorySecretProvider, який видає динамічний	HashiCorp Vault, AWS KMS або GCP Secret Manager із підтримкою ротації секретів

Компонент / Функціональний напрям	У PoC реалізовано	Потрібно для промислового впровадження
	секрет в оперативній пам’яті	
Аудит, C7	ConsoleAuditLogger, який записує події у стандартний потік виводу та зберігає знімок подій у пам’яті	Інтеграція із SIEM, наприклад Splunk або Elastic Security, системами журналювання на кшталт Loki, брокером Kafka та сховищем із підтримкою незмінного зберігання, наприклад S3 Object Lock
Реагування на інциденти, C8	Формування подій аудиту, які можуть використовуватися як вхідні дані для подальшого аналізу	SOAR-система, інструкції реагування, процес цифрового розслідування інцидентів, автоматизована ізоляція скомпрометованих компонентів
Асинхронна взаємодія	У демонстраційній версії не реалізовано	Інтеграція з Kafka, RabbitMQ або NATS; SASL/SCRAM, SASL/OAUTHBEARER, ACL на рівні топіків, черг або subject-ів

Наведена таблиця окреслюють межі деталізації демонстраційної реалізації. Усі зазначені компоненти, необхідні для промислового впровадження, можуть бути інтегровані через наявну систему інтерфейсів MSSCF Java Framework, зокрема SecretProvider, PolicyEngine, AuditLogger і CommunicationSecurityFilter. Це дає змогу розширювати функціональність окремих модулів без перепроєктування ядра фреймворку..

ВИСНОВКИ

У роботі досліджено комплекс проблем, пов’язаних із забезпеченням безпеки комунікаційних каналів у мікросервісній архітектурі, та запропоновано фреймворк із демонстраційною Java-реалізацією для підтвердження практичної застосовності запропонованої моделі. Мету дослідження, що полягала в розробленні цілісного фреймворку та формулюванні практичних рекомендацій щодо його впровадження, досягнуто. Основні результати дослідження наведено нижче.

1. Систематизовано основні складники поверхні атаки мікросервісних систем. Показано, що перехід від монолітної архітектури до мікросервісної істотно збільшує кількість мережевих інтерфейсів і точок взаємодії між компонентами. Крім того, короткий життєвий цикл контейнерів, значна частина яких існує менше ніж п’ять хвилин, змінює вимоги до збирання журналів, збереження артефактів і цифрового розслідування інцидентів. Проаналізована статистика IBM, Verizon, Sysdig, Akamai та Traceable AI за 2023–2025 роки підтверджує актуальність проблеми: середня вартість витоку даних у мультихмарних середовищах сягає 5,27 млн дол. США, а значна частка організацій повідомляє про інциденти, пов’язані з безпекою API.
2. Класифіковано загрози для мікросервісних систем за моделлю STRIDE та виокремлено вектори, особливо характерні для такої архітектури: горизонтальне поширення атаки, підроблення ідентичності сервісу, експлуатація незадокументованих API, атаки на ланцюг постачання програмного забезпечення та порушення авторизації на рівні об’єктів. Для кожної категорії загроз визначено базові механізми захисту.
3. Систематизовано нормативну й методичну базу дослідження. Проаналізовано серію документів NIST SP 800-204 за 2019–2024 роки, а також NIST SP 800-207 і NIST SP 800-207A, присвячені архітектурі нульової довіри. Крім того, розглянуто специфікації OAuth 2.0, JWT,

OAuth 2.0 Authorization Server Metadata, OpenID Connect Core 1.0, документи CNCF, матеріали SPIFFE/SPIRE, а також документацію Kubernetes, Istio та Envoy.

4. Порівняно ключові механізми захисту комунікаційних каналів: TLS і mTLS, рішення сервісної сітки, зокрема Istio, Linkerd, Cilium і Consul, ідентифікацію робочих навантажень за допомогою SPIFFE/SPIRE, авторизацію на основі OAuth 2.0, OIDC і JWT, а також використання API Gateway. На основі звітів CNCF та академічних досліджень встановлено, що mTLS є практично доцільним механізмом захисту міжсервісного трафіку, однак його вплив на продуктивність залежить від конкретної реалізації, конфігурації сервісної сітки та характеру навантаження.
5. Розроблено фреймворк, який подано як набір восьми функціональних напрямів забезпечення безпеки комунікацій у мікросервісній архітектурі: C1 — контроль периметра та відкритості API; C2 — захищений транспортний рівень; C3 — ідентифікація робочих навантажень; C4 — авторизація на основі токенів і сесій; C5 — ухвалення та застосування рішень щодо доступу на основі ABAC; C6 — керування життєвим циклом секретів, ключів і сертифікатів; C7 — спостережуваність, аудит і телеметрія безпеки; C8 — виявлення інцидентів, реагування та відновлення. Функціональні напрями фреймворку подано як модель архітектурних рішень, що ґрунтується на принципах архітектури нульової довіри та узгоджується з рекомендаціями NIST SP 800-204A/B/C і NIST SP 800-207..
6. Запропоновано матрицю відповідності між загрозами STRIDE, функціональними напрямами фреймворку та конкретними механізмами захисту. Матриця охоплює дванадцять типових загроз, характерних для мікросервісної архітектури, і дає змогу інженеру з безпеки швидко визначити, які напрями фреймворку можуть бути застосовані для зниження відповідного ризику.

7. Розроблено чотирирівневу модель зрілості фреймворку від базового рівня M1 до оптимізованого рівня M4 та визначено набір ключових показників ефективності. До них належать частка сервісів з активним mTLS, середній час ротації сертифікатів, час виявлення інцидентів і реагування на них, а також частка секретів, що зберігаються в централізованому сховищі. Це дає змогу впроваджувати фреймворк поступово, без потреби в одноразовій повній трансформації всієї архітектури.
8. Розроблено демонстраційну Java-реалізацію фреймворку для підтвердження практичної застосовності запропонованої моделі. Реалізацію створено з використанням Java 17 і Maven без зовнішніх залежностей під час виконання. Фреймворк містить пакети core, identity, policy, secrets, audit, gateway та demo, структура яких узгоджена з функціональними напрямами концептуальної моделі. Базову логіку реалізації перевірено чотирма модульними тестами та консольною демонстрацією, що охоплює типові сценарії: коректний запит на читання, заборонений запит на запис, токен від недовіреного видавця та прострочений токен.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Chandramouli R. Security Strategies for Microservices-based Application Systems: NIST Special Publication 800-204. Gaithersburg : National Institute of Standards and Technology, 2019. 60 p. DOI: 10.6028/NIST.SP.800-204. URL: https://csrc.nist.gov/pubs/sp/800/204/final (дата звернення: 19.04.2026).
2. Chandramouli R. Building Secure Microservices-based Applications Using Service-Mesh Architecture: NIST Special Publication 800-204A. Gaithersburg : National Institute of Standards and Technology, 2020. DOI: 10.6028/NIST.SP.800-204A. URL: https://csrc.nist.gov/pubs/sp/800/204/a/final (дата звернення: 19.04.2026).
3. Chandramouli R., Butcher Z., Chetal A. Attribute-based Access Control for Microservices-based Applications Using a Service Mesh: NIST Special Publication 800-204B. Gaithersburg : NIST, 2021. DOI: 10.6028/NIST.SP.800-204B. URL: https://csrc.nist.gov/pubs/sp/800/204/b/final (дата звернення: 19.04.2026).
4. Chandramouli R. Implementation of DevSecOps for a Microservices-based Application with Service Mesh: NIST Special Publication 800-204C. Gaithersburg : NIST, 2022. DOI: 10.6028/NIST.SP.800-204C. URL: https://csrc.nist.gov/pubs/sp/800/204/c/final (дата звернення: 19.04.2026).
5. Chandramouli R. Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines: NIST Special Publication 800-204D (Initial Public Draft). Gaithersburg : NIST, 2024. URL: https://csrc.nist.gov/pubs/sp/800/204/d/ipd (дата звернення: 19.04.2026).
6. Rose S., Borchert O., Mitchell S., Connelly S. Zero Trust Architecture: NIST Special Publication 800-207. Gaithersburg : NIST, 2020. 50 p. URL: https://csrc.nist.gov/pubs/sp/800/207/final (дата звернення: 19.04.2026).
7. A Zero Trust Architecture Model for Access Control in Cloud-Native Applications in Multi-Cloud Environments: NIST Special Publication 800-207A. Gaithersburg : NIST, 2023. URL: https://csrc.nist.gov/pubs/sp/800/207/a/final (дата звернення: 19.04.2026).
8. OWASP API Security Top 10 — 2023 [Електронний ресурс] / OWASP Foundation. — 2023. — URL: https://owasp.org/API-Security/editions/2023/en/0x11-t10/ (дата звернення: 21.04.2026).
9. Cloud Native Security Whitepaper v2 [Електронний ресурс] / CNCF Security Technical Advisory Group. — San Francisco : Cloud Native Computing

Foundation, 2022. — URL: https://www.cncf.io/reports/cloud-native-security-whitepaper/ (дата звернення: 21.04.2026).

10. SPIFFE [Електронний ресурс] / Cloud Native Computing Foundation. — URL: https://www.cncf.io/projects/spiffe/ (дата звернення: 22.04.2026).
11. SPIRE [Електронний ресурс] / Cloud Native Computing Foundation. — URL: https://www.cncf.io/projects/spire/ (дата звернення: 22.04.2026).
12. Hardt D. (Ed.) The OAuth 2.0 Authorization Framework: RFC 6749 [Електронний ресурс] / Internet Engineering Task Force. — 2012. — URL: https://datatracker.ietf.org/doc/html/rfc6749 (дата звернення: 22.04.2026).
13. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT): RFC 7519 [Електронний ресурс] / IETF. — 2015. — URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 25.04.2026).
14. Jones M., Sakimura N., Bradley J. OAuth 2.0 Authorization Server Metadata: RFC 8414 [Електронний ресурс] / IETF. — 2018. — URL: https://datatracker.ietf.org/doc/html/rfc8414 (дата звернення: 25.04.2026).
15. OpenID Connect Core 1.0 incorporating errata set 2 [Електронний ресурс] / OpenID Foundation. — 2023. — URL: https://openid.net/specs/openid-connect-core-1_0.html (дата звернення: 19.04.2026).
16. OpenID Connect Discovery 1.0 [Електронний ресурс] / OpenID Foundation. — 2023. — URL: https://openid.net/specs/openid-connect-discovery-1_0.html (дата звернення: 19.04.2026).
17. SPIFFE Concepts [Електронний ресурс] / SPIFFE Project. — URL: https://spiffe.io/docs/latest/spiffe-about/spiffe-concepts/ (дата звернення: 22.04.2026).
18. Naor Y. Performance Comparison of Service Mesh Frameworks: the MTLS Test Case [Електронний ресурс] / Deepness Lab. — 2024. — URL: https://deepness-lab.org/wp-content/uploads/2024/05/Service_Mesh_Performance_Project_Report.pdf (дата звернення: 19.04.2026).
19. Security [Електронний ресурс] / Kubernetes Project. — 2025. — URL: https://kubernetes.io/docs/concepts/security/ (дата звернення: 28.04.2026).
20. Security Concepts [Електронний ресурс] / Istio Project. — URL: https://istio.io/latest/docs/concepts/security/ (дата звернення: 28.04.2026).

21. Mutual TLS Migration [Електронний ресурс] / Istio Project. — URL: https://istio.io/latest/docs/tasks/security/authentication/mtls-migration/ (дата звернення: 25.04.2026).

22. Security Architecture Overview [Електронний ресурс] / Envoy Proxy. — URL: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/security/security (дата звернення: 25.04.2026).

23. TLS [Електронний ресурс] / Envoy Proxy. — URL: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/security/ssl (дата звернення: 25.04.2026).

24. Open Policy Agent Documentation [Електронний ресурс] / Open Policy Agent Project. — URL: https://openpolicyagent.org/docs (дата звернення: 29.04.2026).

25. Introducing Policy As Code: The Open Policy Agent (OPA) [Електронний ресурс] / CNCF Blog. — 2020. — URL: https://www.cncf.io/blog/2020/08/13/introducing-policy-as-code-the-open-policy-agent-opa/ (дата звернення: 29.04.2026).

26. Vault Secrets Management Use Cases [Електронний ресурс] / HashiCorp. — URL: https://www.hashicorp.com/en/products/vault/use-cases/secrets-management (дата звернення: 29.04.2026).

27. Vault Secrets Operator for Kubernetes [Електронний ресурс] / HashiCorp Developer. — 2023. — URL: https://developer.hashicorp.com/vault/tutorials/kubernetes-introduction/vault-secrets-operator (дата звернення: 29.04.2026).

28. cert-manager — Official Documentation [Електронний ресурс] / cert-manager Project. — URL: https://cert-manager.io (дата звернення: 29.04.2026).

29. Berardi D., Giallorenzo S., Mauro J., Melis A., Montesi F., Prandini M. Microservice security: a systematic literature review. PeerJ Computer Science. 2022. DOI: 10.7717/peerj-cs.779. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC8771803/ (дата звернення: 01.05.2026).

30. Zdun U., Queval P.-J., Simhandl G., Scandariato R., Chakravarty S., Jelic M., Jovanovic A. Microservice Security Metrics for Secure Communication, Identity Management, and Observability. ACM Transactions on Software Engineering and Methodology. 2022. DOI: 10.1145/3532183. URL: https://dl.acm.org/doi/10.1145/3532183 (дата звернення: 01.05.2026).

31. Stojkovic F. et al. Security Issues and Challenges in Service Meshes : arXiv:2010.11079. 2020. URL: https://arxiv.org/abs/2010.11079 (дата звернення: 01.05.2026).

32. Cost of a Data Breach Report 2024 [Електронний ресурс] / IBM Security ; Ponemon Institute. — Armonk : IBM, 2024. — URL: https://www.ibm.com/think/insights/whats-new-2024-cost-of-a-data-breach-report (дата звернення: 01.05.2026).

33. 2024 Data Breach Investigations Report [Електронний ресурс] / Verizon Business. — New York : Verizon, 2024. — URL: https://www.verizon.com/business/resources/reports/2024-dbir-data-breach-investigations-report.pdf (дата звернення: 12.05.2026).

34. 2023 Cloud-Native Security and Usage Report [Електронний ресурс] / Sysdig. — San Francisco : Sysdig Inc., 2023. — URL: https://www.sysdig.com/press-releases/sysdig-2023-usage-report (дата звернення: 12.05.2026).

35. 2025 State of API Security [Електронний ресурс] / Traceable AI. — 2025. — URL: https://www.traceable.ai/2025-state-of-api-security (дата звернення: 12.05.2026).

36. New Study: 84 % of Security Professionals Experienced an API Security Incident in the Past Year [Електронний ресурс] / Akamai Technologies. — 2024. — URL: https://www.akamai.com/newsroom/press-release/new-study-finds-84-of-security-professionals-experienced-an-api-security-incident-in-the-past-year (дата звернення: 12.05.2026).

37. Zero Trust Security Model Implementation in Microservices Architectures Using Identity Federation: arXiv:2511.04925. 2025. URL: https://arxiv.org/pdf/2511.04925 (дата звернення: 12.05.2026).

38. Technical Report: Performance Comparison of Service Mesh Frameworks: the MTLS Test Case: arXiv:2411.02267. 2024. URL: https://arxiv.org/html/2411.02267v1 (дата звернення: 12.05.2026).

39. Volkov D., Liubchenko V. Securing Microservices: Challenges and Best Practices. CEUR Workshop Proceedings. 2024. Vol. 3790. Paper 02. URL: https://ceur-ws.org/Vol-3790/paper02.pdf (дата звернення: 12.05.2026).

40. OpenTelemetry Documentation [Електронний ресурс]. OpenTelemetry. URL: https://opentelemetry.io/docs/ (дата звернення: 15.05.2026).